

Computer Architecture|

with (MIPS) Assembly

Peter Stallinga

`peter@stallinga.org`

Computer Architecture: with (MIPS) Assembly (paperback)

Peter Stallinga

v. 2.0 (March 25, 2025)

Copyright © 2025, Peter Stallinga. All rights reserved

ISBN: 978-1-31281-319-9

Typefaces: Times Roman, Helvetica, Free Sans

Typesetting: L^AT_EX2e 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 5 (adapted from Wikipedia), 6 (Wikipedia), 91 (output of MARS 4.5 of Sanderson and Vollmar), and 97 (Wikipedia).



Non-profit science organization

Contents

1	Introduction	1
2	Number systems	11
2.1	Binary numbers	16
2.2	Octal, hexadecimal and binary-coded decimal	18
2.3	Arithmetic	19
2.4	Number conversion	25
2.5	Negative numbers	29
2.6	ASCII	32
2.7	Gray code	33
2.8	Floating point	34
2.9	Exercises	37
3	Boolean algebra/logic	41
3.1	Set theory and Boolean algebra	42
3.2	Huntington postulates	44
3.3	Formal derivation of the truth tables	50
3.4	From postulates to truth tables	55
3.5	sum-of-products (SoP) and product-of-sums (PoS)	55
3.6	Karnaugh maps	59
3.7	An alternative to Karnaugh maps	64
3.8	Exercises	68
4	Hardware components	71
4.1	Electronics: from transistors to gates	71
4.1.1	Tri-state	79
4.2	From gates to logic circuits	81
4.3	Karnaugh maps in electronics	85
4.4	Timing	88
4.5	Latches, flip-flops and memory	92
4.6	Finite-state machines (FSM)	97
4.6.1	Moore machine sequencers	98
4.6.2	Mealy machines	102

4.7 Exercises	107
5 Integration	111
5.1 Half-adder/full-adder	111
5.2 Summing and subtracting	114
5.3 Advanced adding and subtracting	116
5.4 Advanced logic circuits	116
6 Computers	127
6.1 Arithmetic and logic unit (ALU)	127
6.2 Central processing unit (CPU)	129
6.3 Control logic	130
6.4 Programming the CPU	133
6.5 Advanced arithmetic	134
6.6 Floating point; IEEE 754	137
6.7 Advanced calculations	144
7 Information and memory	149
7.1 (Quantifying) Information	150
7.2 Information sizes	158
7.3 External memory	159
7.4 Memory speed-up	168
7.5 Software aspects of memory	169
7.5.1 Heap and stack	172
7.5.2 Garbage collection, paging, and overlays	175
7.5.3 Addressing modes	178
8 Hardware/software aspects	179
8.1 Interrupts	180
8.2 Bus	182
8.3 Communication	184
9 Architecture of MIPS	189
10 MARS	195
10.1 I/O (system calls), memory access	198
10.2 Arithmetic	204
10.3 Jump and branch; (goto, if ... then goto)	206
10.4 Loops; (for, while, do-while)	211
10.5 Masking	213
10.6 Variables, arrays and structures	217
10.7 Floating point	223
10.8 Functions and the stack	228
10.9 Macros (pseudo-instructions)	235
10.10 Gauss method	238

10.11	Calculating blockchain	248
10.12	RARS: Evolution of MARS	250
11	Examples of architectures	261
11.1	Difference Engine of Charles Babbage	262
11.2	Intel 4004	263
11.3	MOS 65xx	267
11.4	Atmel AVR	276
11.5	Intel x86	282
11.6	Analog computing	293
11.7	Advanced architectures	294
A	Intel 4004 instruction set	303
B	MOS 65xx instruction set	305
C	AVR Atmel instruction set	309
D	x86 instruction set	313
E	x86 exceptions, H/W interrupts	317
F	x86 BIOS, MS-DOS and API and Linux interrupts	319
G	Linux (Debian) system calls	321
H	MIPS instruction set	323
I	MARS (MIPS) Assembler directives	333
J	(MARS) MIPS system calls	335
K	(RARS) RISC-V base integer instruction set (RV32I)	337
L	(RARS) RISC-V system calls	347
M	ASCII	349
<i>i</i>	Index	351

1 | Introduction

Where does the word computer come from? In spite of what we may think, that it is a very modern word, it actually comes from the 17th century. It derives from the word 'compute' which is from the Latin for 'reckon with' (from the prefix *com-* and the verb *putare* meaning to reckon). The word architecture is then derived from the Greek *archi* + *tekon*, a builder or craftsman. Computer Architecture is then described as "A fundamental underlying design of computer hardware, software, or both". Indeed, that is what we will describe here in this book. The design of a computer hardware and software and the interplay between them. It has three basic parts: Digital systems, computer hardware, and assembly programming.

This book is part of a course of Computer Architecture at the first year of a university. It ranges from the number system and then goes from hardware all the way up to programming it in machine language and finally assembly. To start with this point, assembly: While mainly talking about general concepts, explains a specific assembly 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. Less is more, because we learn how to see the computer as a machine that constantly shoves information around, like a train engineer shunting wagons in a railway depot.

But where does all this fit in? In the great hierarchy of knowledge. Up front it has to be said that it is assumed here that the reader is comfortable with high-level programming languages. Specifically, for the part on Assembly it is assumed that the reader knows the basic concepts of the C programming language, such as:

- Data types
- Variables and constants
- Comment
- Input/output (`printf`, `scanf`)

- Branching: If, if-else and switch; conditional execution
- Loops: for, while, do-while
- Arrays and structures
- Functions (and recursivity)
- Passing by value and passing by reference
- Pointers

Especially the last item — pointers — is very important because basically *everything* in Assembly is pointers, as we will see. We will also see that nearly all concepts of the list above are not part of Assembly. There are no functions. There are no arrays and structures. Variables do not exist. No looping instructions exist. These are all concepts of the level above, high-level programming languages. In the level below, all these concepts have to be implemented by us ourselves, but we will see that MIPS Assembly 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 assembly is quite strong. One can even say that, in a way, C is Assembly with macros.

Now the main question is, *why* should we want to learn to write programs in Assembly, 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 from Assembly that divisions are slower compared to multiplications, we might want to replace a C instruction `a = b/5.0` by `a = 0.2*b`. Or instead of `a = 2*b` we could use `a = b+b`, since multiplications are very slow compared to additions.
2. In cases where 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 Assembly.
3. In case no high-level compiler exists for an architecture from C to machine language we have to write it ourselves, in which case the knowledge of Assembly is essential.

If you want to jump to the 'goodies', because you came here only to learn how to program in Assembly, you can directly jump to Chapter 10. 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 Assembly 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 of 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 discern 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 we have in 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 you 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 will see primarily such modern machines here.

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 physicists 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 in space 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 were a mere mathematical property, forgetting that currents consist of moving electrons. It is possible to get away with this approach, 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 ($L = dI/Vdt$)*. 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 Figure 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 really 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

* R is resistance, V is voltage, Q is charge, I is current, C is capacitance, L = inductance, t is time, and $I = dQ/dt$.

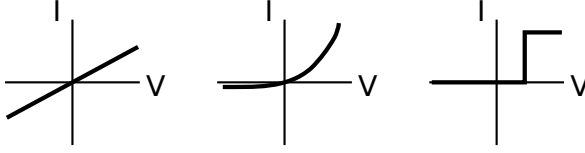


Figure 1: Linear electronics, non-linear electronics and digital electronics (I-V curves).

the Ebers-Moll equation, named after two German scientists,

$$I = I_0 \left[\exp \left(\frac{V}{V_T} \right) - 1 \right].$$

Even more interesting is a transistor, which is a diode with a current-voltage relation between A and B controlled by a third connection, C. The resistance (and current) between A and B is thus controlled by a voltage placed at C, and we thus have a trans-resistor, or 'transistor' for short. 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 can exist, because the voltages at the output can never exceed the supply voltages (see Figure 1). We can call this 'binary' or 'digital electronics'. We enter the realm of modern 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.

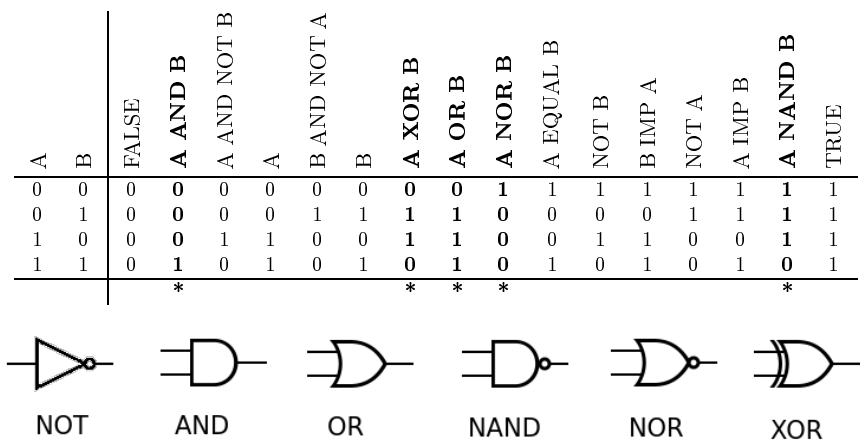


Figure 2: The 16 possible 2-input 1-output logic gates (IMP means material implication). Symbols for the six fundamental gates. The one-input-one-output inverter (NOT) gate and the five basic two-input-one-output logic gates (AND, OR, NAND, NOR, XOR).

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 on only one input (copying it, or inverting it), and some of them are redundant (that is, since any functionality can be implemented by, for instance, only NANDs, some gates are redundant). That leaves basically the five logic circuits mentioned above: AND, NAND, OR, NOR, XOR. They are shown in bold and marked with an asterisk in Figure 2.

Note that the names of the devices are 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 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 and 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 2021. 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 really starts kicking in (at Chapter 5). A central processor has several input lines and output lines. It can be imagined 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 actuate on the data.

In the first approach, in the early days, 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', and this is the second-generation programming, built on top of the hardware layer, which we can call the first generation of computing. A program for calculating the product of two floating point numbers might be

```
0100011000000001
```

```
0001000011000000
```

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 expensive machine do something that can perfectly be done by a cheap 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 an expensive human doing work that can perfectly be done by a cheap 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 code and then ran.

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`, or something like that. This type of programming is called third-generation programming languages, or (macro) 'assembly', exactly the layer of knowledge included in this book, with MIPS chosen for pedagogic reasons (see Chapter 10). 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-assembly, 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 would be 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 in this assembly. It was always things like for-loops, or if-then-else structures. People must have started writing programs in meta-assembly, nearly English, macros written in (macro) Assembly (note that the two words 'macro' have different meanings). 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 Assembly and fed to the computer. Well, they must have thought, if Assembly 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 fourth-generation programming languages, of which FORTRAN (Formula Translator), BASIC (Beginners All-Purpose Symbolic Instruction Code), Pascal (named after French Mathematician Blaise Pascal), FORTH (supposed to have been a FOurTH generation language; at the time commands could only be five letters), and C are the most famous. This is normally the starting level in Informatics of the engineer and scientist alike. It is well possible to never look back at the levels below, but learning Assembly is useful for the reasons given earlier.

Table I: Hierarchical levels in computing; 'generations'.

Generation	Meaning	Example
First	Hardware, logic gates	AND, OR, XOR ...
Second	Machine language	0111011101000...
Third	Assembly	addi \$t0, \$t1, 2
Fourth	Imperative Programming	C: <code>printf("%s", hellow);</code>
Fifth	Object Oriented	C++: <code>myString.print</code>
Sixth	Frameworks	Android Studio

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, which we can call fifth-generation programming languages (see Table I for a summary of the levels of programming). 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 are based on assembly, 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. So the story goes.

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* 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 Figure 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.

In some cultures also a system of tallying with hands is used that can count up to 12 (or 24). Take your right right and count with your thumb placed on the top phalanx of your little finger (1), then on its middle phalanx (2), then on its bottom phalanx (3). Then continue on the top phalanx of the ring finger (4), etc., until you reach the bottom phalanx of your index finger (12). You can now repeat the process on your left hand (from 13 to 24). While not good for counting, it is very useful for tallying and it does not need any external hardware.

The disadvantage of low-number bases such as the hexal system is that numbers get large faster. That is, they rapidly start having many digits. To give an example, nearly 1 million (999 999) in the decimal system has 6

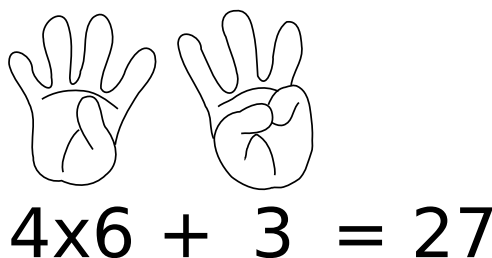


Figure 3: An example of the hexal (base 6) number system with counting on our hands. One hand is used for the units and the other hand for the multiples of the base number 6.

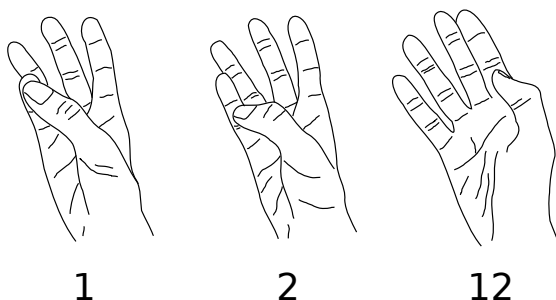


Figure 4: Tallying with the 12 phalanges of the four fingers of your hand, selected by the thumb. Here shown the numbers 1, 2 and 12.

digits while in the hexal system it has 8 (33 233 343).

Other number systems that were popular were base-20. This unit is called a 'score' in English. Take for example this funny Limerick (author unknown):

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.

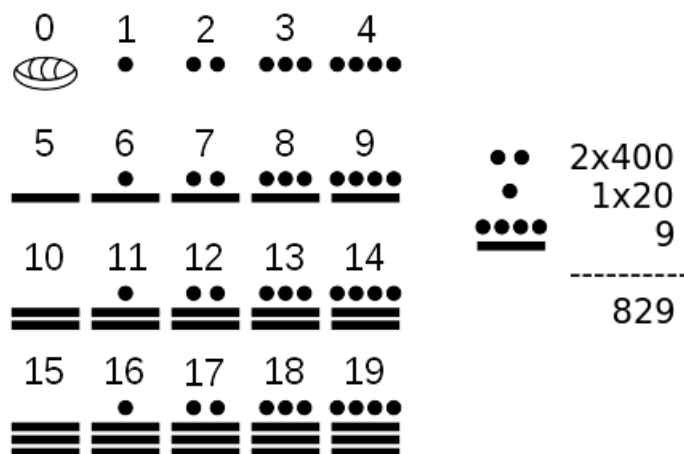


Figure 5: The number system of the Mayas based on scores (20). Their unit was represented by a dot, with five of them written as a horizontal bar. The numbers 0 to 19 are shown here. Subsequent digits were stacked vertically on top of each other. An example is the number 829 ($= 2 \times 20^2 + 1 \times 20^1 + 9 \times 20^0$) 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.

Interestingly, also the Mayas used this system of scores, see Figure 5. 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)?

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 answer is that the 20-base is a very natural outcome if you want to do calculations, because it is a number that is divisible by 2, 4, 5 and 10.

The best system was then 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, or with a remainder as we learned in primary school. Compare for example the difficulty of our

𐎶 1	𐎶𐎵 11	𐎶𐎶𐎵 21	𐎶𐎶𐎶𐎵 31	𐎶𐎶𐎶𐎵𐎶 41	𐎶𐎶𐎶𐎵𐎶𐎵 51
𐎶𐎶 2	𐎶𐎶𐎶 12	𐎶𐎶𐎶𐎶 22	𐎶𐎶𐎶𐎶𐎶 32	𐎶𐎶𐎶𐎶𐎶𐎶 42	𐎶𐎶𐎶𐎶𐎶𐎶𐎶 52
𐎶𐎶𐎶 3	𐎶𐎶𐎶𐎶 13	𐎶𐎶𐎶𐎶𐎶 23	𐎶𐎶𐎶𐎶𐎶𐎶 33	𐎶𐎶𐎶𐎶𐎶𐎶𐎶 43	𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶 53
𐎶𐎶𐎶𐎶 4	𐎶𐎶𐎶𐎶𐎶 14	𐎶𐎶𐎶𐎶𐎶𐎶 24	𐎶𐎶𐎶𐎶𐎶𐎶𐎶 34	𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶 44	𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶 54
𐎶𐎶𐎶𐎶𐎶 5	𐎶𐎶𐎶𐎶𐎶𐎶 15	𐎶𐎶𐎶𐎶𐎶𐎶𐎶 25	𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶 35	𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶 45	𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶 55
𐎶𐎶𐎶𐎶𐎶𐎶 6	𐎶𐎶𐎶𐎶𐎶𐎶𐎶 16	𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶 26	𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶 36	𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶 46	𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶 56
𐎶𐎶𐎶𐎶𐎶𐎶𐎶 7	𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶 17	𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶 27	𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶 37	𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶 47	𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶 57
𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶 8	𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶 18	𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶 28	𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶 38	𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶 48	𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶 58
𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶 9	𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶 19	𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶 29	𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶 39	𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶 49	𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶 59
𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶 10	𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶 20	𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶 30	𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶 40	𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶 50	

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

decimal system to write out $1/3$. It gives a result with an infinite number of digits, $0.3333 \dots (= 3/10 + 3/10^2 + 3/10^3 \dots)$. While for the Babylonians, $1/3$ is simply $20/60$, or a single digit '20' after the floating point.

Note that our geometry and timing are 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 and 101 by placing the I after 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 $-10 + 100 - 1 + 10$. Digits

in Roman are:

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

Examples:

$$2018 = \text{MMXVIII} = 1000 + 1000 + 10 + 5 + 1 + 1 + 1$$

$$2019 = \text{MMXIX} = 1000 + 1000 + 10 - 1 + 10$$

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 a factor equal to the base number.
- Negative numbers are indicated by a preceding '-' sign. Positive numbers can (but don't have to) be preceded by a '+' sign. Alternatively, numbers can be limited to positive, unsigned, values only.

This is all a matter of *conventions*, and at school these conventions are hammered into our heads in such a way that we think it is the only way to do things. Maybe you did not even think it was possible to have a binary system until you were in secondary school. Or any base-number system. In any case, the most common convention is one in which the position of the number determines its weight, as presented above. As an example, if the number base is x , then the number represented by

$$\pm abcde_x$$

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. In modern times the most-often used convention is base-10 for humans and base-2 (binary) for computers. Let's analyze the latter in detail now.

2.1 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, simply called 'binary'. 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 to a logic '0'. Now, if we have a set of transistor circuits, each in a certain state designated by '1' or '0', we can *imagine* that they represent a binary number. The number does not really 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 things that exist in reality are things that are in the realm of Physics, and all physical things can be expressed in terms of the seven basic SI units (kilogram, meter, second, ampere, mol, candela, kelvin) or their derivatives. If it does not have a unit in SI it does not exist. Do not confuse a number with a quantity which has unit 'mol'. If there are two people in the room, in fact there are $2.0/N_A$ mol people in the room, with N_A the number of Avogadro, $N_A = 6.022 \times 10^{23}/\text{mol}$.

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; the unit $V = \text{kg}\cdot\text{m}^2\cdot\text{s}^{-3}\cdot\text{A}^{-1}$) 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, depending on the *interpretation* of the bit pattern. To show what is meant, consider this code in C:

```
int i = 65;
printf("%c\n");
```

It will not print "65", but "A". The %c tells the computer to interpret the bit pattern as an ASCII code, rather than an integer.

The reason why it is useful to do this abstraction is that, *if* we assume these to represent numbers, the logic of the hardware (ALU, arithmetic and 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 the model in our heads. We are thus able to think as a computer, or a computer 'thinks' the same way we do. The computer *implements* our thinking, and does it, in contrast to us, flawlessly and rapidly.

Returning to the subject, binary numbers are very useful in informatics because the gates work with two possible states. Note that it is not obligatory for computers to use this technology. Russians are famous for having developed ternary-logic computers based on gates that have three possible states, and trits instead of the more common binary logic (two possible states, bits). Apparently, it had some advantages, such as lower power consumption and lower production cost (this according to 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. Rests to say that all logic and all computation could also theoretically be done in ternary-gate electronic computers.

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 might have the following representations:

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 imagination. 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, whatever the physical states are.

An important observation to make here: A combination of n binary gates — or n 'bits' — can take $N = 2^n$ different possible 'values' or output combinations. Reasoning the other way around: we need at least $n = \log_2(N)$ bits (gates) to represent a number that has N 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.

Table II: Look-up table for binary, hexadecimal, octal and BCD.

Decimal 0b	Binary 0d	Hexadecimal 0x	Octal 0o	BCD
0	0000	0	0	0
1	0001	1	1	1
2	0010	2	2	2
3	0011	3	3	3
4	0100	4	4	4
5	0101	5	5	5
6	0110	6	6	6
7	0111	7	7	7
8	1000	8	10	8
9	1001	9	11	9
10	1010	A	12	-
11	1011	B	13	-
12	1100	C	14	-
13	1101	D	15	-
14	1110	E	16	-
15	1111	F	17	-

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:

$$1101 = 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 8 + 4 + 0 + 1 = 13.$$

2.2 Octal, hexadecimal and binary-coded decimal

The hexadecimal number system is very often used in informatics. It is ubiquitous for a very simple reason already mentioned above: 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 II.

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). To indicate that we are dealing with hexadecimal, instead of writing the 16 subscript after the number (something that is difficult to do in ASCII texts), the number is often preceded by 'zero-x', 0x, as in 0x1AC3, which is $1 \times 16^3 + 10 \times 16^2 + 12 \times 16 + 3 = 6851_{10}$. This can also be written as 0d6851, which is the exact same thing. With the same convention, binary numbers are often preceded by 0b, although less frequently used in writing code. Even less used are 0o for octal and 0d for decimal. The 0d is redundant, since decimal is considered default.

To convert hexadecimal to and from binary is very easy. As the table shows, every hexadecimal digit is equivalent to four binary digits. Thus we can write out hexadecimal by replacing every digit by its corresponding 4-bit equivalent. Likewise, binary numbers are converted to hexadecimal by grouping the binary digits in sets of 4 bits and looking up in the table what hexadecimal digits they are. An example: 0xA3 is 1010 0011 in binary. And 0b1001 1101 0100 0001 is 0x9D41.

The same trick we can also use for octal numbers. A single octal digit is converted to three binary digits, ranging from 000 (octal 0o0) to 111 (octal 0o7). An example, 375_8 (0o375) becomes 011 111 101. Converting to octal is done by grouping in sets of three bits (with optional adding of zeros in front if needed), and looking in the table. As an example, 0b11101001 is converted into 0o351.

The table also shows binary-coded decimal (BCD), which is similar to hexadecimal with the last combinations 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 or hex it is 00000111 + 00000111 = 00001110, or $0x07 + 0x07 = 0x0E$. As can easily be shown, binary calculations are more efficient in computers but are further away from human thinking, and this is the reason why earlier architectures often use decimal or binary-coded decimal.

2.3 Arithmetic

This part may seem a little odd. To talk about how to do arithmetic in the decimal system, while this is exactly what we have already learned very well in primary school. But, in spite of us having learned it well, perhaps we did not realize *what* the underlying method was. What algorithm we used for arithmetic? More specifically for addition, subtraction, multiplication and division. We will see that these algorithms can be used in any number system, including binary. Axiomatically we state here that the arithmetic is independent of the number base they are performed in. The number base is merely the convention how to represent the number. So, doing the number

in decimal will have the same result as converting all operands to binary, or any other number system, doing the arithmetic in that system, and at the end converting back to decimal to present it to the human user who is used to decimal numbers.

The arithmetic itself is based on us memorizing (burning in our hardware, our minds) some simple tables and executing simple operations. We make here one simplification in that we use only two-operand arithmetic (although we have learned to do multi-operand additions $A+B+C+\dots$ in school).

Let's start with addition, $Z = A+B$. We perform additions of multi-digit numbers by doing addition one digit at a time. As an example, adding 714 to 888: We start with the least significant digits and add them. 8 plus 4 is 12. Houston, we got a problem, 12 is not a single digit and we have an overflow. We have learned to deal with this by just keeping the last digit and remembering the overflow digit '1', which we call a 'carry'. Now this carry is then carried over to the next digit summation, which now is $8+1+\text{carry}$ which gives 0 plus a new carry. We get this result from the very simple memorized addition table (see Table III) that includes such carry-ins and carry-outs. Our addition becomes (white spaces are considered 0)

$$\begin{array}{rcl}
 1\ 1\ 1\ 0 & \text{carries} & \\
 8\ 8\ 8 & \text{operand A} & \\
 7\ 1\ 4 & \text{operand B} & + \\
 \hline
 1\ 6\ 0\ 2 & &
 \end{array}$$

Note that we start with the carry-in of the least significant digit always equal to 0. Note also that the carry cannot be larger than 1 for bi-operand additions. At most we can have an addition of $9+9+\text{carry}=19$ which will give 9 with a carry of 1 as output. This statement is valid for additions in any number system.

When we design a computer, all we need is some hardware that somehow has this addition table burned in and the algorithm programmed in. We could of course also have burned in all possible sums of any integers A and B up to a certain amount of digits, but that is becoming rapidly extensive (and expensive) hardware. Moreover, it is not very flexible. Up to how many digits are we going to burn it in hardware? Two? Three? A hundred? We just prefer to do the simple table in hardware and the rest in the algorithm, with some useful tricks here and there (see Table IV).

Now let's look at how additions work in binary calculations. The idea is the same, but obviously the table is much simpler, see Table V. An example, adding 110 to 011 becomes

$$\begin{array}{rcl}
 1\ 0\ 0 & \text{carries} & \\
 0\ 1\ 1 & \text{operand A} & \\
 1\ 1\ 0 & \text{operand B} & + \\
 \hline
 1\ 0\ 0\ 1 & &
 \end{array}$$

Table III: Addition table for decimal numbers. As an example, adding 6 (row 6) to 6 with a carry ("c" in column 6) gives a sum of 3 with a carry out (carry) equal to 1, shown in bold font.

	0 c	1 c	2 c	3 c	4 c	5 c	6 c	7 c	8 c	9 c	
0	0 1 0 0	1 2 0 0	2 3 0 0	3 4 0 0	4 5 0 0	5 6 0 0	6 7 0 0	7 8 0 0	8 9 0 0	9 0 0 1	sum carry
1	1 2 0 0	2 3 0 0	3 4 0 0	4 5 0 0	5 6 0 0	6 7 0 0	7 8 0 0	8 9 0 0	9 0 0 1	0 1 1 1	sum carry
2	2 3 0 0	3 4 0 0	4 5 0 0	5 6 0 0	6 7 0 0	7 8 0 0	8 9 0 0	9 0 0 1	0 1 1 1	1 2 1 1	sum carry
3	3 4 0 0	4 5 0 0	5 6 0 0	6 7 0 0	7 8 0 0	8 9 0 0	9 0 0 1	0 1 1 1	1 2 1 1	2 3 1 1	sum carry
4	4 5 0 0	5 6 0 0	6 7 0 0	7 8 0 0	8 9 0 0	9 0 0 1	0 1 1 1	1 2 1 1	2 3 1 1	3 4 1 1	sum carry
5	5 6 0 0	6 7 0 0	7 8 0 0	8 9 0 0	9 0 0 1	0 1 1 1	1 2 1 1	2 3 1 1	3 4 1 1	4 5 1 1	sum carry
6	6 7 0 0	7 8 0 0	8 9 0 0	9 0 0 1	0 1 1 1	1 2 1 1	2 3 1 1	3 4 1 1	4 5 1 1	5 6 1 1	sum carry
7	7 8 0 0	8 9 0 0	9 0 0 1	0 1 1 1	1 2 1 1	2 3 1 1	3 4 1 1	4 5 1 1	5 6 1 1	6 7 1 1	sum carry
8	8 9 0 0	9 0 0 1	0 1 1 1	1 2 1 1	2 3 1 1	3 4 1 1	4 5 1 1	5 6 1 1	6 7 1 1	7 8 1 1	sum carry
9	9 0 0 1	0 1 1 1	1 2 1 1	2 3 1 1	3 4 1 1	4 5 1 1	5 6 1 1	6 7 1 1	7 8 1 1	8 9 1 1	sum carry

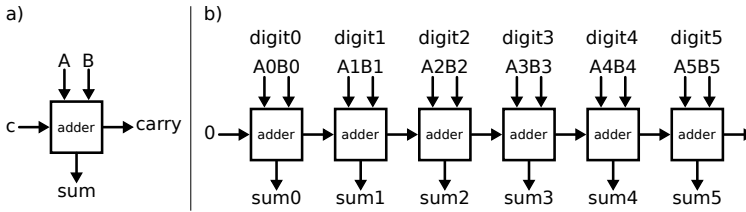


Figure 7: a) A single-digit adder implementing a table such as the one shown in Table III and Table V for decimal and binary. b) How this hardware can perform a multi-digit addition.

The hardware can be much simpler since it needs to remember only 16 values compared to 400 for the decimal adder.

✱

Subtractions can be done by a similar table, but with 'borrows' instead of carries. The reader is invited to design such tables. However, as we will

Table IV: Some simple calculation 'tricks'

-
- Multiplying by the base (always written as '10' in that base!) is simply adding a zero at the end of the number. Example: 10 times 53 is 530. We call this operation in computer jargon a 'left-shift'.
 - A number which has a sum of digits divisible by 3 is divisible by 3. Example: 7893 has a sum of digits $7+8+9+3 = 27$, which has a sum of digits $2+7 = 9$, which is divisible by 3 and thus 7893 is divisible by 3 (meaning leaving no remainder when dividing by 3). Same trick is for divisibility by 9. Moreover: Divisible by 2 if last digit is divisible by 2. Divisible by 5 if last digit is 0 or 5.
 - The square of a two-digit number ending with 0 is the square of the first digit followed by '00': $(D0)^2 = D \times D \text{ } 00$. Example: $40^2 = 16 \text{ } 00$. The square of a two-digit number ending with 5 is the product of the first digit times the first digit plus one and added '25': $(D5)^2 = D \times (D+1) \text{ } 25$. Example: 45^2 , knowing $4 \times 5 = 20$, is 20 25. Generally, a two-digit square (example $28^2 = 784$) can be done in the following way: 1) Add the last digit to the full number, e.g., $28 + 8 = 36$. 2) Multiply this number by the first digit and left-shift one case. E.g., $2 \times 36 = 72 \text{ } 0$. 3) Add the square of the last digit to this number, e.g. $720 + 64 = 784$.
 - $9 \times D = (D-1) (10-D)$. Example for D equal to 8: $9 \times 8 = 7 \text{ } 2$.
-

Table V: Addition table for binary numbers. As an example, adding 1 (row 1) to 0 with a carry ("c" in column 0) gives a sum of 0 with a carry out (carry) equal to 1, shown in bold font.

	0 c	1 c	
0	0 1	1 0	sum
	0 0	0 1	carry out
1	1 0	0 1	sum
	0 1	1 1	carry out

see, there is a much more elegant way of subtractions, and that is to convert the subtraction into an addition of the negative number. As we will see, this is easily done when we use two's-complement representation of numbers, as discussed later in this chapter. For this reason we postpone the discussion

Table VI: Multiplication table for decimal shown as digit pairs cp , with c the carry and p the product of the operation.

\times	0	1	2	3	4	5	6	7	8	9
00	00	00	00	00	00	00	00	00	00	00
01	00	01	02	03	04	05	06	07	08	09
02	00	02	04	06	08	10	12	14	16	18
03	00	03	06	09	12	15	18	21	24	27
04	00	04	08	12	16	20	24	28	32	36
05	00	05	10	15	20	25	30	35	40	45
06	00	06	12	18	24	30	36	42	48	54
07	00	07	14	21	28	35	42	49	56	63
08	00	08	16	24	32	40	48	56	64	72
09	00	09	18	27	36	45	54	63	72	81

of subtractions to a later stage, namely at the place where we design the hardware (Chapter 6).

✱

Also for multiplications we use an algorithm of reducing it to single-digit operations that are then looked up in a table. The basic bi-operand single-digit multiplication information is shown in Table VI. The product and carry entries are here written in double-digit representations cp , with c the carry and p the product of the operation. For example, 6×6 gives 36, meaning the result is 6, with a carry of 3.

We can now perform a multi-digit bi-operand multiplication by performing shift-multiply-add operations, a method that is called the Russian-peasant algorithm. Take for example 238×123 . We first multiply each digit of the first operand by the last digit (least significant digit, LSD) of the second operand (3), 238×3 which gives 714 as can easily be seen (note also the carries): From our look-up table we know that $3 \times 8 = 24$ (4 plus 2 as carry). Then the next step $3 \times 3 = 9$ plus the carry is 1 plus 1 as carry. Then, the final step is $3 \times 2 = 6$. Plus the carry from the previous step is 7. Final result: 714. Remember how we did this in school (the unused digits of operand B at every step put in brackets):

1 2 0

2 3 8

(12)3

7 1 4

carries

operand A

LSD operand B \times

intermediate result

Then we shift operand A to the left (multiplying effectively by the base 10; adding a 0, thus 2380) and operand B to the right (dividing effectively by

Table VII: Multiplication table for binary. Note: Never a carry occurs.

×	0	1
0	0	0
1	0	1

the base 10, forgetting the remainder). So, the last digit of operand B is now 2, so we multiply 2380 by 2, and then add the previous result to it:

0 1 0 0	carries
2 3 8 0	operand A
(1)2	LSD of operand B ×
<hr/> 4 7 6 0	new result
1 0 0 0	carries
4 7 6 0	new result
7 1 4	previous intermediate result +
<hr/> 5 4 7 4	new intermediate result

Once again shift operand A to the left and operand B to the right. So, the last digit of operand B is now 1, so we multiply 23800 by 1, and then add the previous result to it:

0 0 0 0 0	carries
2 3 8 0 0	operand A
1	LSD of operand B ×
<hr/> 2 3 8 0 0	new result
0 1 0 0 0	carries
2 3 8 0 0	new result
5 4 7 4	previous intermediate result +
<hr/> 2 9 2 7 4	final result

The next shift-right division-by-10 of operand B results in 0 and we have finished our calculation. Final result: 29274.

In binary this algorithm is much simpler. The look-up table for binary numbers is very simple, it can barely be called a table, the output is either 0, if operand B is 0, or equal to operand A if B is 1. There is never a carry! See Table VII. This makes the Russian-peasant algorithm very easy; shift-multiply-add becomes simply shift-add.

An example of a long multiplication by this algorithm in binary is shown here below for multiplying A = 13 (1101) with B = 11 (1011). Note that if the rightmost bit of operand B is 0, no addition has to be done, and if that

bit is 1, no multiplication has to be done, the operand A simply needs to be added:

1 1 0 1	A
1 0 1 1	B. (LSB of B is 1: add A)
<hr/> 1 1 0 1	temporary result
1 1 0 1 0	L-shifted A, (R-shifted B=10 1: add A)
<hr/> 1 0 0 1 1 1	temporary result
1 1 0 1 0	L-shifted A, (R-shifted B=1 0: no action)
<hr/> 1 0 0 1 1 1	temporary result
1 1 0 1 0 0 0	L-shifted A, (R-shifted B=1: add A)
<hr/> 1 0 0 0 1 1 1 1	final result (143 ₁₀)

No multiplication has to be done in binary. Just shift, mask (seeing if a digit is 1) and add.

2.4 Number conversion

Now the question is, how to convert between number systems? The obvious way is to do the counting in the *destination* number system. An example is the one given above from hexal to decimal, $12345_6 = 1 \times 6^4 + 2 \times 6^3 + 3 \times 6^2 + 4 \times 6^1 + 5 \times 6^0 = 1865_{10}$, with the right side of the equation a decimal calculation. We are very familiar with calculations in decimal, which are fundamentally based on the multiplication tables that were hammered into us at an early age, see Table VI. From the expression above it is clear that also the powers n of the source base numbers x , expressed in the destination number system, are useful, see Tables VIII, IX and X for decimal, pental and binary, respectively. In this case we will need the line with powers of 6 expressed in the decimal system. With this table we can see that 12345_6 is

$1 \times 6^4 =$	1,296
$2 \times 6^3 =$	432
$3 \times 6^2 =$	108
$4 \times 6^1 =$	24
$5 \times 6^0 =$	5
	<hr/>
	1,865

When converting to another base number system, we have to do the calculation in that destination system. For that we have to reinvent the wheel and learn the multiplication table of our system. An example is converting base-5 123 (38 in decimal) to base-6. We need the multiplication and addition tables of base-6, which are given in Table XI. 123_5 is then $1 \times 5^2 + 2 \times 5 + 3 = 41_6 + 14_6 + 3_6$, and that is by summing (the first line are the carries in the sum)

Table VIII: Some powers x^n in the decimal system (base 10).

$n \rightarrow$ $x \downarrow$	5	4	3	2	1	0
2	32	16	8	4	2	1
3	243	81	27	9	3	1
4	1,024	256	64	16	4	1
5	3,125	625	125	25	5	1
6	7,776	1,296	216	36	6	1
7	16,807	2401	343	49	7	1
8	32,768	4,096	512	64	8	1
9	59,049	6,561	729	81	9	1
16	1,048,576	65,536	4,096	256	16	1

Table IX: Multiplication table and some powers x^n in the pental system (base 5).

	0	1	2	3	4
0	0	0	0	0	0
1	0	1	2	3	4
2	0	2	4	11	13
3	0	3	11	14	22
4	0	4	13	22	31

$x \downarrow$ $n \rightarrow$	10 (= 5 ₁₀)	4	3	2	1	0
2	112	31	13	4	2	1
3	2244	311	102	14	3	1
4	13044	2011	224	31	4	1
11 (= 6 ₁₀)	222101	20141	1331	121	11	1
13 (= 8 ₁₀)	2022033	112341	4022	224	13	1
20 (= 10 ₁₀)	11200000	310000	13000	400	20	1
31 (= 16 ₁₀)			112341	2011	31	1

$$\begin{array}{r} 110 \text{ (carries)} \\ \hline 41 \\ 14 \\ 3 \text{ +} \\ \hline 102 \end{array}$$

The result is 102₆. And that is indeed $1 \times 6^2 + 2 = 38_{10}$.

Table X: Multiplication table and some powers x^n in the binary system (base 2).

		0		1	
0		0	0		
1		0	1		

$x \downarrow$	$n \rightarrow$	101	100	11	10	1	0
10 (= 2 ₁₀)		100000	10000	1000	100	10	1
11 (= 3 ₁₀)		11110011	1010001	11011	1001	11	1
100 (= 4 ₁₀)			100000000	1000000	10000	100	1

Table XI: Addition (left) and multiplication (right) tables for hexal (base-6) shown as digit pairs cp, with c the carry and p result of the operation.

		Multiply								Add					
×		0	1	2	3	4	5	+		0	1	2	3	4	5
00		00	00	00	00	00	00	00		00	01	02	03	04	05
01		00	01	02	03	04	05	01		01	02	03	04	05	10
02		00	02	04	10	12	14	02		02	03	04	05	10	11
03		00	03	10	13	20	23	03		03	04	05	10	11	12
04		00	04	12	20	24	32	04		04	05	10	11	12	13
05		00	05	14	23	32	41	05		05	10	11	12	13	14

exercise: Base-5 to base-4

Convert 123₅ to base-4.

We can also do the calculations in the *source* system. It involves divisions (compared to multiplications above). An example is the division of 12345₆ by 14₆ in the conversion of base-6 to base-10. In base 6:

12345/14 = 510 + 5/14 (remainder is 5₁₀)

510/14 = 30 + 10/14 (remainder is 6₁₀)

30/14 = 1 + 12/14 (remainder is 8₁₀)

1/14 = 0 + 1/14 (remainder is 1₁₀)

So the number in the decimal system – reading the remainders from the last to the first – is 1865, which is equal to the result found before.

Considering our acquaintance with base-10 calculations, conversions *from* base-10 to other bases are better done by the latter technique, while con-

versions *to* base-10 are easier done with the former method. Especially, conversions *to* base-2 are done by successive divisions by 2 (with the remainders in reverse order the resulting bit pattern), and conversions *from* base-2 are done by successive multiplications by 2 (or powers of 2) and addition. Conversions from base- x to base- y (with x and y both not 10) can also be done by first converting base- x to base-10 and then from base-10 to base y . In this way we make use of our familiarity with base-10 calculations.

Now let's do a more complicated example, from hexal to pental. What is 1234_6 in base-5? We can use the multiplication table and power table from the pental system, Table IX. The first four powers of six (11) in base-5 are: 1, 11, 121, 1331. So we need to calculate $1 \times 1331 + 2 \times 121 + 3 \times 11 + 4 \times 1$ in the pental system:

$$\begin{array}{rcl}
 1 \times 1331 & = & 1331 \\
 2 \times 121 & = & 242 \\
 3 \times 11 & = & 33 \\
 4 \times 1 & = & 4 \\
 \hline
 & = & 2220
 \end{array}$$

In other words, the answer is 2220_5 . Check (by converting both 1234_6 and 2220_5 to the decimal system, 310_{10}) that this is indeed correct.

While floating point numbers will be treated in a separate section further on, it should be noted here that conversions after the floating point are done by the opposite operation. So, if we want to convert for instance 0.8125_{10} to binary, we do successive *multiplications* by 2. If the number is larger or equal to 1, we write a 1 (0 otherwise) and continue with the remainder after the floating point. So

$$\begin{array}{rcl}
 0.8125 \times 2 & = & 1 + 0.625 \\
 0.625 \times 2 & = & 1 + 0.25 \\
 0.25 \times 2 & = & 0 + 0.5 \\
 0.5 \times 2 & = & 1 + 0.0
 \end{array}$$

and the number is 0.1101 in binary.

Note that in an arithmetic evaluation we can first do the conversion to the new number system, then do the calculation, or first do the calculation and then the conversion. The end result is the same. Even if we do not realize it, this is the basis of computer calculations; they can be done in binary and the result then presented in decimal.

Note also that numbers in the unary system (base 1) are possible. Just as binary digits have two possibilities (0 and 1) and base- n digits in general have n possibilities, base-1 digits have 1 possibility (0). The value of a number is simply the number of 0s in the number. So $7_{10} = 0000000_1$. We can no longer use the convention that adding 0s in front of a number does not change its value, though, as it obviously does in the case of base-1 numbers.

Nor does floating point make any sense. Unary counting does, however, make sense in payments based on things such as gold; note that every digit in the number has the same weight, similar to the equality of every gold coin in your pocket. It is a tallying system, not a counting system.

2.5 Negative numbers

So far so good. The complications 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. The problem is that the sign symbol (+ or -), introduced by us before is not directly implemented in hardware, as that has only 0s and 1s, conventionally. So we have to augment our convention.

As a first thought, we may think of using — ‘sacrificing’ — one bit for the sign, and continuing to use rest for the magnitude (which now has a smaller range, 0 unto $2^{n-1} - 1$). This scheme is called sign-magnitude. See Figure 8a 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:

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

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

<i>bit pattern</i>		<i>unsigned</i>	<i>sign- magnitude</i>
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 Figure 8b 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:

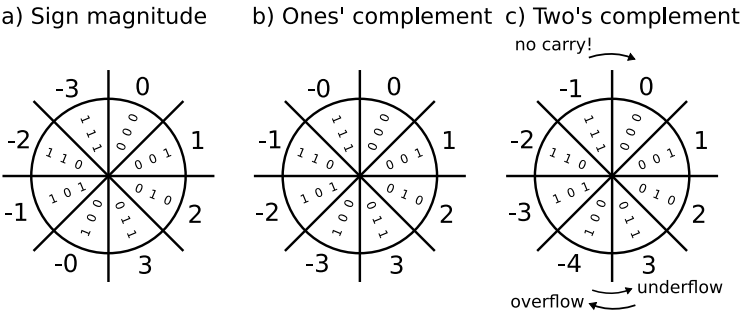


Figure 8: Three ways of representing signed numbers; 3-bit examples. a) In sign-magnitude, the MSB is the sign bit, 0 for '+' and 1 for '-', while the rest of the bits form a normal unsigned $n-1$ -bit number. b) In ones'-complement, to find a negative number, simply all bits are inverted. c) In two's-complement, the MSB has a negative weight -2^{n-1} while the other bits have positive weight $+2^{n-2}$ to $+2^0$.

<i>bit pattern</i>		<i>unsigned</i>	<i>ones'- complement</i>
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 Figure 8c 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,

111111111111111111111111111111011

is also -5).

Note that two's-complement has only one version of zero. It can use the same hardware logic as unsigned integers:

<i>bit pattern</i>		<i>unsigned</i>	<i>two's- complement</i>
0001	:	1	+1
1010	:	10	-6
1011	:	11	-5

Interestingly, the carry can be ignored (see Figure 8c). On the other hand, a phenomenon occurs halfway the bit pattern, when the MSB changes $0 \rightarrow 1$,

<i>bit pattern</i>		<i>unsigned</i>	<i>two's- complement</i>
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 $1 \rightarrow 0$.

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:

$$\begin{array}{rcl}
 3 & = & 00000011 \\
 \text{invert:} & & 11111100 \\
 \text{add 1:} & & 11111101 \quad = -3
 \end{array}$$

- 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 modern architectures are represented in two's-complement. We can consider the hardware itself as a black box. However, since this book is not only about programming, we will see how binary calculations are implemented in hardware.

For completeness sake, we can also see how sign-magnitude, $(n-1)$ s-complement, and n 's-complement works in other base- n systems. For example the decimal (base-10) system. In sign magnitude we just use the convention of adding a digit in front, a maximum digit ('9' in this case) is negative and a '0' is positive, so +123 in sign magnitude becomes 0123 and -123

becomes 9123. From this it is immediately clear that when communicating numbers it is of utmost importance that both sides of the communication use the same convention (protocol) or information will not be transmitted correctly, people may easily confuse 9123 as nine thousand one hundred and twenty three. When you communicate, you must speak the same language!

Continuing, in 9s-complement, to get the number with opposite sign, we simply invert all digits (that is subtract all individual digits from 9). So the positive number 0123 (that is +123 in all conventions) becomes -123 by taking the 9s-complement of all digits: 9876. The 10's-complement is found by either taking the 9s-complement and adding one (9877) or by subtracting the (floating point number) 0.123 from 10 (hence the name 10's-complement) and removing the floating point '.' part:

$$\begin{array}{r} 10.000 \\ 0.123 \\ \hline 9(.)877 \end{array}$$

so the answer is 9877. Just like in 2's-complement, also in 10's-complement are subtractions done by adding the 10's-complement sign-inverted number. An example, $(+154) - (+138) = (+154) + (-138)$:

$$\begin{array}{r} 0154 \\ 9862 \\ \hline (1)0016 \end{array}$$

which is +16, where we have ignored the carry (as explained in Figure 8). Or $(+138) - (+154) = (+138) + (-154)$:

$$\begin{array}{r} 0138 \\ 9847 \\ \hline 9985 \end{array}$$

and that is equal to -16 in 10's-complement.

2.6 ASCII

Another way of interpreting bit patterns 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 M):

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

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 or not! 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. "Anything else?"

2.7 Gray code

A seemingly strange way of representing binary numbers is by Gray code, as shown in Table XII. The advantage of this scheme is that adjacent numbers only differ by one bit. This is important for some electronic applications, when it should not be possible to jump to intermediate undetermined states when switching. Imagine changing from 7 (0111) to 8 (1000) in which all four bits commutate. In this case, depending on the order of changes of the bits, which unlikely occur all at the same time, intermediate fake states may be introduced, for instance from 7 to 8: 0111→1111→1011→1001→1000, which would represent 7→15→11→9→8 and is clearly not correct. In Gray code only one bit would change: 0100→1100 and no fake states are introduced.

To convert from Gray code to binary use the following algorithm: From left to right, copy the first bit of Gray code and then for every next bit sum the new bit of Gray code to the previous bit of binary (no carry). Example: 11011100 in Gray code becomes 10010111 in binary. To convert from binary to Gray code: From left to right, keep the MSB (most significant bit) and then for every bit in Gray code sum the last two bits of binary. Example: 10010111 in binary becomes 11011100 in Gray code.

Table XII: 4-bit Gray code.

decimal	Binary	Gray code
0	0000	0000
1	0001	0001
2	0010	0011
3	0011	0010
4	0100	0110
5	0101	0111
6	0110	0101
7	0111	0100
8	1000	1100
9	1001	1101
10	1010	1111
11	1011	1110
12	1100	1010
13	1101	1011
14	1110	1001
15	1111	1000

2.8 Floating point

Floating point numbers are a subject by itself. 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 degree, 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), any interval, no matter how small, contains an infinite number of numbers, and not all can be stored in a 32 bit-register. Our calculations are bound to be incorrect. Take for example the division $1.0/3.0$, which is $0.33333\dots$ and thus incorrect on any finite state machine. But how incorrect are they going to be? And is that acceptable, or not? We therefore have to state from the beginning that computers cannot do calculations with real domain numbers (\mathbb{R}). The best they can do is doing integer numbers that are real-ish. They mimic \mathbb{R} floating point numbers.

They are approximations, and we have to keep that in the back of our heads when we do floating point. With that in mind we can continue.

In our integer number system we used the convention that digits had weight and the weight started with 1 (right-most digit) and increased by a factor equal to the base every step to the left. As an example, if the number base is x , then the number represented by $\pm abcde_x$ 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.

In floating-point notation numbers follow this same scheme. After the floating point (floating comma in some countries), directly after the least-significant digit (with weight 1), the digits get *divided* by the base. So

$$\pm abc.de_x$$

is equal to

$$\pm(a \times x^2 + b \times x^1 + c \times x^0 + d \times x^{-1} + e \times x^{-2}).$$

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.

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 while adjusting the exponent, 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.

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... (most calculators do like this), 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! Most statisticians like to write their probabilities as ".493", etc., thus without this redundant 0. Also many programming languages allow this notation.

Now let's take a specific example of (decimal) floating point numbers with 3 digits for the fraction and 2 digits for the exponent, both also including a sign.

$$\pm abc \times 10^{\pm de}$$

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 9, 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' and the calculator normally refuses to continue. 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 — half 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 (0.1%). 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 decrease the number of digits for the exponent, the relative error of our calculations drops, but, as a price to pay, the range of our numbers narrows. For instance, for a system with 4 digits for the fraction and 1 for the exponent

$$\pm abcd \times 10^{\pm e}$$

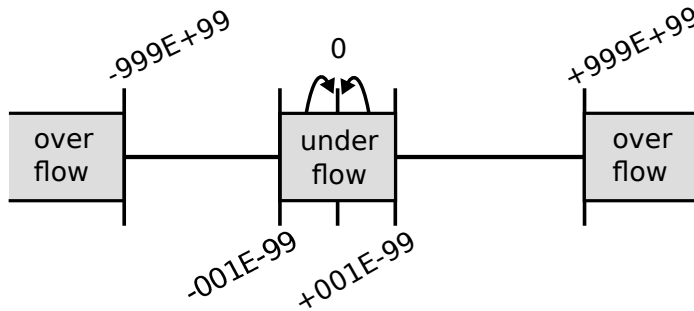


Figure 9: The seven ranges of base-10 floating-point numbers with three digits for the fraction and two for the exponent. When the number is too big (on either side), it is called 'overflow'. When it is too small it is called 'underflow', which is normally mapped to 0.

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

The relative error is now about $1/10,000$ (0.01%).

As we will see when we start designing the architecture of our computer, the IEEE 754 norm implements the floating point numbers in computing. It is based on a base-2 sign-magnitude number representation, with fields inside the bit pattern for the exponent and the fraction. As an example, a float is represented by a 24-bit sign-magnitude fraction with an implicit 1 and an 8-bit excess-127 exponent.

2.9 Exercises

exercise: Binary to decimal

- 1: Convert from binary to decimal:
 - a) 1010001
 - b) 0.11
 - c) 1011010.1010

exercise: Octal to decimal

2: Convert from octal to decimal:

- a) 273
- b) 1021
- c) 16.423

exercise: Hexadecimal to decimal

3: Convert from hexadecimal to decimal:

- a) 145
- b) A2C1
- c) 1A.B2

exercise: Binary to hexadecimal and octal

4: Convert from binary to hexadecimal and octal:

- a) 10101110101101111011
- b) 011011110011110000000001

exercise: Hexadecimal to binary

5: Convert from hexadecimal to binary:

- a) 1A.B2
- b) A2C1
- c) F0AC29E

exercise: Decimal to unsigned binary

6: Convert from decimal to unsigned binary:

- a) 122
- b) 98
- c) 48.45
- d) 195.98
- e) 500
- f) 1000 (be smart, use e)
- g) 2000 (be even smarter, use e,f)

exercise: Arithmetic

7: Do the following (unsigned) arithmetic:

- a) $101011 + 10111$

- b) $1101+1110+1001$
- c) $11101-10110$
- d) $1100.010-1000.111$

exercise: Sign-magnitude decimal to 10's-complement decimal

8: Convert from sign-magnitude decimal to 10's-complement decimal. Use both techniques, $10.000\dots - N$ and 9s'-complement+1:

- a) +123
- b) -48
- c) -323
- d) -2047

exercise: Sign-magnitude binary to 2's-complement binary

9: Convert from sign-magnitude binary ($0 \equiv +$, $1 \equiv -$; first number is +0010001) to 2's-complement binary. Use both techniques, $10.000\dots - N$ and 1s'-complement+1:

- a) 0001 0001
- b) 1011 0111
- c) 1000 0110
- d) 1000 1111

exercise: Subtraction decimal

10: Convert to 10's-complement decimal and do the subtractions (adding the 10's-complement of the negative number):

- a) $423 - 198$
- b) $327 - 432$

exercise: Subtraction binary

11: Convert to 2's-complement binary and do the subtractions (adding the 2's-complement of the negative number):

- a) $79 - 42$
- b) $87 - 99$

exercise: Binary-coded decimal

12: Do the following arithmetic in BCD:

- a) $79 + 101$
- b) $87 + 179$
- c) $198 - 43$
- d) $143 - 98$

exercise: Gray-code

13: Convert the following patterns from Gray-code to binary and from binary to Gray code:

- a) 1010
- b) 11011
- c) 11000010001

exercise: Sign-magnitude arithmetic

14: Do the following arithmetic in sign-magnitude:

- a) 1110×1101
- b) 00010101×00001010

exercise: decimal to 2's-complement

15: Convert to 2's-complement 32-bit binary:

- a) 512
- b) -1023 (note: $1023 = 2^{10} - 1$)

exercise: 2's-complement to decimal

16: Convert to decimal the 2's-complement 32-bit binary numbers:

- a) 1111 1111 1111 1111 1111 1111 0000 1100
- b) 1111 1111 1111 1111 1111 1111 1111 1111
- c) 0111 1111 1111 1111 1111 1111 1111 1111

3 | Boolean algebra/logic

In the previous chapter number systems were described. It was also pointed out that the binary number system is the one that is used by computers, because it is easily implemented in digital electronics with the inputs and outputs of gates having only two possible states.

The logic step is then to use Boolean algebra in the calculations because Boolean algebra, or Boolean logic, also works with two distinct possible values, or states. We call it Boolean algebra when we do actual calculations and call it Boolean logic when we are using only logical operations, but from a mathematical point of view, they are indistinguishable. The former normally talks about 0s and 1s (and is thus a basis for a true number system, where more than one digit is allowed), while the latter talks about 'true' and 'false' (and does not have a sequence of digits as numbers do). They can be fully interchanged. Convention is often that 'true' is '1' and 'false' is '0'. If we now see a sequence 011101 it can either mean a set of boolean logic values, false, true, true, true, false, true, or a set of boolean logic values '0', '1', '1', '1', '0', '1'. Or it can be a binary number (as explained in the previous chapter). The different functionality is implemented by the hardware by so-called (boolean) logic gates.

As we have seen in the introduction chapter, there are exactly 16 possible logic 2-input-1-output gates. Some of them are silly (like: the output is equal to one of the inputs, for instance $out = A$, which is a rather elaborate way of making a simple wire, or the output is constant, independent of both inputs), some of them use only one input (like: output is equal to the inverted input A). The six most used ones were given in Figure 2 and have the following behavior:

<i>Logic gate</i>	<i>Explanation</i>
OR	Or: 'true' if at least one input is 'true' 'false' if both are 'false'
AND	And: 'true' if both inputs are 'true' otherwise 'false'

XOR	Exclusive or: 'true' if exactly one input is 'true' otherwise 'false'
NOT	Inversion. 'true' if the single input is 'false' and 'false' if 'true'
NOR	Not or: 'false' if either one or both inputs are 'true' otherwise 'true'
NAND	Not and: 'false' if both inputs are 'true' otherwise 'true'

We can put this in so-called truth tables, where '0' represents 'false' and '1' represents 'true', see Table XIV.

Note here, once again, that all this is mere convention. We might as well have given the gates different names and not talk about 'true' and 'false' or '1' and '0', but, let's say, 'blue' and 'red' or 'car' and 'human'. What is important is that the hardware will implement – and is consistent with – our ideas of trues and falses and thus the behavior of the computer is useful for us. How this is done is not of our concern here at this moment; it will be discussed later, but we know, of course, that electronics are underlying all computers. We might thus think of a '0' being a low voltage and a '1' high voltage, though this is not necessarily the case. It does not matter to us; the only thing that matters is *that* it is implemented following the rules described here. From here on we will write 1 instead of '1' and 0 instead of '0', though we should well keep in mind that these are *not* numbers!

3.1 Set theory and Boolean algebra

In the above we tried to explain the working of a computer by a mathematical approach – logics. We understand that the computer is made of logic gates and with these logic gates we can build a computer. We will now take the opposite approach. We will start with the mathematics of logic and work from there. Once having established the most basic mathematics, we will constantly be building on the results obtained at the preceding step and construct ever more complex systems. At every step new functionalities will emerge. We will see how we can build any logic function and see how in fact any operation (including arithmetics, to name but one) can be expressed in logic functions. It means we now have to go back to the absolute starting point of computing, namely the mathematics of binary logic of Aristotle. Binary in that we deal with pieces of information of the type that can have two possible values. We can imagine answers to questions such as "Is it raining?" It has answers of the type true/false. While this is not necessary – the only thing that is needed is that there exist only two possible values – we can constantly keep this in mind.

Table XIV: Truth tables of the six most important logic operations. 1 represents true and 0 represents false.

X	NOT X	A	B	A NOR B	A	B	A NAND B
0	1	0	0	1	0	0	1
1	0	0	1	0	0	1	1
		1	0	0	1	0	1
		1	1	0	1	1	0

A	B	A OR B	A	B	A AND B	A	B	A XOR B
0	0	0	0	0	0	0	0	0
0	1	1	0	1	0	0	1	1
1	0	1	1	0	0	1	0	1
1	1	1	1	1	1	1	1	0

To derive the mathematics that are underlying the algebra and thus logic gates and thus our entire computer, we will use Set Theory. This is a formalization of Aristotelean Categories and on which Boolean algebra is based that is named after George Boole, the 19th century English mathematician. Set Theory deals with sets of elements that follow certain rules when operations are applied. Classic Set Theory takes operations of union (\cup) and intersection (\cap) of subsets of elements of the full set. This is directly related to logic, because an element is in a union of subsets if the element was in any of the subsets. So, if it was in subset 1 *or* subset 2 *or* ... Likewise, an element is in an intersection of subsets if it was in all of the subsets, so in subset 1 *and* subset 2 *and* ... So we recognize here the binary answers to the questions "Was the element in subset X?" We will also use the complement operation, (X'), which means all elements of the full set that are *not* in the subset X .

Boolean algebra then works with subsets – sets of exactly one element, if you will – with moreover only two possible values, where they can be any two values, even things as 'Ajax' and 'Benfica' (for your information: the two best football teams on this planet). (Aristotelean logic then further defines the values as *logical* 'true' and 'false'). The operations of Boolean algebra are then called disjunction (\vee) and conjunction (\wedge). Disjunction represents the OR-operation and conjunction the AND-operation, and in Aristotelean logic they even have this exact meaning, OR and AND. Tradition is to write these operator symbols as '+' and '.', which will add some confusion because they will remind us of arithmetic operations of summation and multiplication, respectively, which they are definitely not! In spite of the confusion, we will also use it here.

Apart from this, note also that no inverse operations exist. There is some kind of adding ($a + b$), but there is no inverse operation of 'subtrac-

Table XV: Set Theory and corresponding Boolean Logic operations.

Set theory		Boolean algebra		Aristotel-	Here
name	symbol	name	symbol	ean logic	
union	\cup	disjunction	\vee	OR	$+$
intersection	\cap	conjunction	\wedge	AND	\cdot
complement	X'	negation	\neg	NOT	\overline{X}

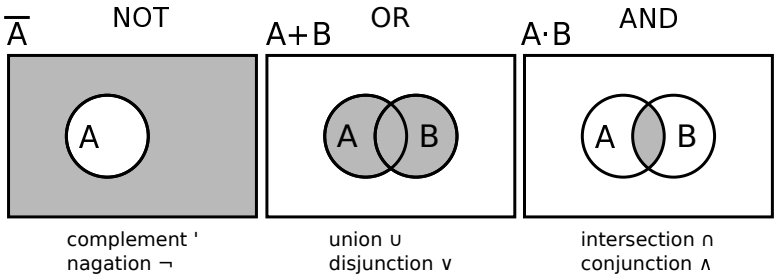


Figure 10: Venn diagrams showing graphically the NOT, OR and AND operations. In this book we will use these Aristotelean names.

tion'! There is no $a - b$. Likewise, there is some kind of multiplication, but there is no division! $a \cdot b$, but no a/b . There are AND-operations and OR-operations, but no reverse-AND-operations and reverse-OR-operations. There are inverse sets or inverse elements, but no inverse operations. This makes sense if we see it as Aristotelean logic, as logic inverses of 'and' and 'or' do not exist. This has to do with the principle that logic operations destroy information, as we will see, and information once destroyed cannot be recovered. A logic operation cannot be 'undone' by its inverse.

In summary, in Boolean algebra, conjunction (AND) and disjunction (OR), and an inverse operation (NOT) are used. The main operators on these variables are as given in Table XV.

3.2 Huntington postulates

As we have argued, mathematics is the art of creating virtual worlds on basis of axioms or postulates. Similar to the construction of the world of natural number that are based on the Peano axioms on which also arithmetics (addition and multiplication) can be based we are now going to define the basis for Boolean Algebra. The mathematics behind this logic and logic gates is described by the Huntington postulates which implements Set The-

ory with the *three* (sic) operations AND (\cdot), OR ($+$) and NOT (\bar{X}). We will see that, on basis of these postulates, we can design or describe the entire computer, from things as simple as logic gates to complex systems such as mobile-network applets. The six Huntington postulates* are the following:

1. As a first observation it is to be pointed out that this implements the concept of Set Theory, and these sets are **closed sets**. The operations {OR, AND, NOT} on any element of the group of possible (input) values results in an element of the group of values. In other words, for any combination of values of A and B in the set:

$A + B$ is an element of the set

$A \cdot B$ is an element of the set

\bar{A} is an element of the set.

As we know, in binary the set is limited to $\{0, 1\}$ or $\{\text{true}, \text{false}\}$. It means in practice that inputs of 0 volt and 5 volt produce outputs of 0 volt or 5 volt, etc. This is not completely true in practice; voltages have ranges and tolerances. For instance, for an input to be treated as 0, the voltage has to be below 0.7 volt. And an output of a circuit that is 0 is guaranteed to be below 0.5 volt (to make sure it is treated as a 0 in the next stage).

2. Boolean algebra has the **commutative law**:

$$A + B = B + A$$

$$A \cdot B = B \cdot A.$$

This means in reality that we can exchange input pins of our elementary gates (OR and AND) without altering the output.

3. The **associative law** tells us:

$$(A + B) + C = A + (B + C)$$

$$(A \cdot B) \cdot C = A \cdot (B \cdot C),$$

meaning that it does not matter in what order we process the operations, first processing A and B and then combining that result with C , or first processing B and C and then combining the result with A . This also means that brackets are not needed and we can write $A + B + C$ and $A \cdot B \cdot C$. Applying the commutative law, we can also prove that $A + B + C = C + B + A$ and $A \cdot B \cdot C = C \cdot B \cdot A$, etc. It also implies that indeed we can treat the logic (and the circuits) fully by dyadic (two-input) operations (and devices). Multi-input logic can be constructed from dyadic logic. We will see that this, in fact, is true for any combination of logic operations, for instance $A + B \cdot C$, though in this case of mixed operation, the associative law no longer applies: $A + (B \cdot C) \neq (A + B) \cdot C$. Moreover, as we will see, the XOR, NOR and

*Edward V. Huntington, "Sets of Independent Postulates for the Algebra of Logic". Transactions of the American Mathematical Society, Vol. 5, No. 3 (Jul., 1904), pp. 288-309. URL: <https://www.jstor.org/stable/1986459>.

NAND logic can be constructed from the above three logic operations (gates).

4. The **distributive law** from mathematics implies that

$$A + (B \cdot C) = (A + B) \cdot (A + C)$$

$$A \cdot (B + C) = (A \cdot B) + (A \cdot C).$$

The second one looks familiar from mathematics *if* we assume the symbols represent multiplication and addition. In that case, the first one does not seem to make sense. Yet, it is exactly the same thing. Both should read as

$$A \text{ op1 } (B \text{ op2 } C) = (A \text{ op1 } B) \text{ op2 } (A \text{ op1 } C),$$

with op1 and op2 two operations. Substituting op1=+ and op2= \cdot will result in the first, while substituting op1= \cdot and op2=+ will result in the second distributive law. Note that it is also valid for op1=op2,

$$A + (B + C) = (A + B) + (A + C)$$

$$A \cdot (B \cdot C) = (A \cdot B) \cdot (A \cdot C),$$

making use of the commutative law and the useful corollaries $A \cdot A = A$ and $A + A = A$ that will be discussed below. In standard arithmetic, the distributive law does not apply because normal mathematical operations have priority, $A + (B \times C) \neq (A + B) \times (A + C)$.

5. **Identity elements** exist. An element, which we label 0, exists such that for all A

$$A + 0 = A.$$

This element is unique. There exists only one element that satisfies the above equation.

Likewise, an element, which we label 1, exists such that for all A

$$A \cdot 1 = A.$$

Also this element is unique; there cannot be two elements that satisfy the equation. Notice that we have not yet identified what these elements are, nor have we identified what the operations $+$ and \cdot are. 0 could represent 'true' and 1 'false', with \cdot the or-operation and $+$ the and-operation. It would work. Yet, we know where this will lead to (hence the suggestive symbols), and actually already presented in the opening section of this chapter; we started this chapter actually with physical logic gates. Boolean algebra is *defined* by the seven rules stated here. These rules *define* the existence of such elements and then we *attribute* the symbols 0 and 1 to them. From there the logic follows and we see that we can define operators that follow the logic and electronic engineers can design electronic gates that implement the logic operations. From an engineering point of view: we need a formalism that describes the functionality of our electronic gates and see that Boolean algebra serves the purpose very well. From a mathematician's point of view we see that a computer implements Boolean logic very well.

Table XVI: The Huntington postulates.

H1a	$\forall A, B \in \{0, 1\} : A + B \in \{0, 1\}$
H1b	$\forall A, B \in \{0, 1\} : A \cdot B \in \{0, 1\}$
H2a	$A + B = B + A$
H2b	$A \cdot B = B \cdot A$
H3a	$A + (B + C) = (A + B) + C$
H3b	$A \cdot (B \cdot C) = (A \cdot B) \cdot C$
H4a	$A + (B \cdot C) = (A + B) \cdot (A + C)$
H4b	$A \cdot (B + C) = (A \cdot B) + (A \cdot C)$
H5a	$A \cdot 1 = A$
H5b	$A + 0 = A$
H6a	$A + \bar{A} = 1$
H6b	$A \cdot \bar{A} = 0$

6. Finally, for every element A there exists a unique element \bar{A} , called A 's **complement**, such that (simultaneously)

$$A \cdot \bar{A} = 0$$

$$A + \bar{A} = 1.$$

This element is the inverse of A , written as \bar{A} or $!A$, also called not- A .

The six Huntington postulates are summarized in Table XVI. We see that attributing $+$ to logic-OR, \cdot to logic-AND, and $!$ to logic-NOT does indeed implement Boolean logic with the seven rules described here if we consider '0' as false and '1' as true. (Note that we might as well have chosen '0' to be true and '1' to be false, with then $+$ the logic-AND operation and \cdot the logic-OR operation).

exercise: Proof of Postulate H5a

Prove that there is only one element z for which $A + z = A$ for all A .

Answer: Suppose there are two such different z , namely z_1 and z_2 . Then $A + z_1 = A$ and $A + z_2 = A$. Substituting $A = z_2$ in the former and $A = z_1$ in the latter we get $z_2 + z_1 = z_2$ and $z_1 + z_2 = z_1$, using the commutative law we get $z_1 = z_1 + z_2 = z_2$ and thus $z_1 = z_2$, meaning z_1 and z_2 cannot be different.

exercise: Proof of Postulate H5b

Prove that there is only one element u for which $A \cdot u = A$ for all A .

Answer: Suppose there are two such different u , namely u_1 and u_2 . Then $A \cdot u_1 = A$ and $A \cdot u_2 = A$. Substituting $A = u_2$ in the former and $A = u_1$ in the latter we get $u_2 \cdot u_1 = u_2$ and $u_1 \cdot u_2 = u_1$, using the commutative law we get $u_1 = u_1 \cdot u_2 = u_2$ and thus $u_1 = u_2$, meaning u_1 and u_2 cannot be different.

Some useful corollaries of these six rules are given in Table XVII*. Two of them stating that operands are idempotent, A used in an OR or AND operation with itself results in itself: $A + A = A$ and $A \cdot A = A$.

The last two lines of this table are examples of De Morgan's laws and can be expressed in English as:

The negation of a disjunction is the conjunction of negations;
and the negation of a conjunction is the disjunction of negations.

or:

The complement of the union of two sets is the same as the intersection of their complements; and the complement of the intersection of two sets is the same as the union of their complements.

The Venn diagrams in Figure 11 make this clearer for the special case of just one disjunction and conjunction, two variable A and B .

Generally, they are written in mnemonic form to easier remember them, making use of the mislabeling of '+' as 'sum' and '.' as 'product', and they become the negation-of-a-sum is the product-of-negations (NoS=PoN):

$$\overline{\sum_i A_i} = \prod_i \overline{A_i}$$

and the negation-of-a-product is the sum-of-negations (NoP=SoN):

$$\overline{\prod_i A_i} = \sum_i \overline{A_i}$$

In this case the Σ symbol is used to represent the disjunction operation ('+', OR) on the elements and Π the conjunction operation ('.', AND) and do not represent the classic arithmetic summing or multiplying operations.

*Huntington himself called some of these corollaries also postulates.

Table XVII: Useful corollaries.

C1	$A \cdot A = A$	(*)
C2	$A + A = A$	(*)
C3	$A + 1 = 1$	
C4	$A \cdot 0 = 0$	
C5	$A + (A \cdot B) = A$	
C6	$A \cdot (A + B) = A$	
C7	$A + (\bar{A} \cdot B) = A + B$	
C8	$A \cdot (\bar{A} + B) = A \cdot B$	
C9	$\bar{\bar{A}} = A$	
C10	$\overline{(A + B)} = \bar{A} \cdot \bar{B}$	NoS=PoN (°)
C11	$\overline{(A \cdot B)} = \bar{A} + \bar{B}$	NoP=SoN (°)

*: Idempotent

°: De Morgan. Figure 11

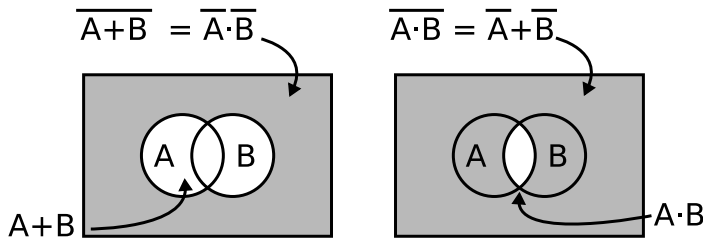


Figure 11: Venn diagrams graphically showing the De Morgan Laws: The negation of the OR-operation is the AND-operation of the negations ("negation of a sum is the product of negations"; NoS=PoN) and the negation of the AND-operation is the OR-operation of the negations ("negation of a product is the sum of negations"; NoP=SoN).

exercise: Proof of Corollaries C1-C9

Prove the corollaries C1 - C9 of Table XVII.

Answer:

$$\begin{aligned} \text{C1: } A \cdot A &\stackrel{H5b}{=} A \cdot A + 0 \stackrel{H6b}{=} (A \cdot A) + (A \cdot \bar{A}) \stackrel{H4b}{=} A \cdot (A + \bar{A}) \stackrel{H6a}{=} \\ &A \cdot 1 \stackrel{H5a}{=} A \end{aligned}$$

$$\text{C2: } A + A \stackrel{H5a}{=} (A + A) \cdot 1 \stackrel{H6a}{=} (A + A) \cdot (A + \bar{A}) \stackrel{H4a}{=} A + (A \cdot \bar{A})$$

$$\stackrel{H6b}{=} A + 0 \stackrel{H5b}{=} A$$

$$C3: A + 1 \stackrel{H6a}{=} A + (A + \bar{A}) \stackrel{H3a}{=} (A + A) + \bar{A} \stackrel{C2}{=} A + \bar{A} \stackrel{H6a}{=} 1$$

$$C4: A \cdot 0 \stackrel{H6b}{=} A \cdot (A \cdot \bar{A}) \stackrel{H3b}{=} (A \cdot A) \cdot \bar{A} \stackrel{C1}{=} A \cdot \bar{A} \stackrel{H6b}{=} 0$$

$$C5: A + (A \cdot B) \stackrel{H5a}{=} (A \cdot 1) + (A \cdot B) \stackrel{H4b}{=} A \cdot (1 + B) \stackrel{C3}{=} A \cdot 1 \stackrel{H5a}{=} A$$

$$C6: A \cdot (A + B) \stackrel{H4b}{=} (A \cdot A) + (A \cdot B) \stackrel{C1}{=} A + (A \cdot B) \stackrel{C5}{=} A$$

$$C7: A + (\bar{A} \cdot B) \stackrel{H4a}{=} (A + \bar{A}) \cdot (A + B) \stackrel{H6a}{=} 1 \cdot (A + B) \stackrel{H5a}{=} A + B$$

$$C8: A \cdot (\bar{A} + B) \stackrel{H4a}{=} (A \cdot \bar{A}) + (A \cdot B) \stackrel{H6B}{=} 0 + (A \cdot B) \stackrel{H5b}{=} A \cdot B$$

C9: Substitute $B = \bar{(\bar{A})}$ in C7 and find $\bar{(\bar{A})} = A$ or $\bar{(\bar{A})} = 0$. Substitute $B = \bar{(\bar{A})}$ in C8 and find $\bar{(\bar{A})} = A$ or $\bar{(\bar{A})} = 1$. Hence $\bar{(\bar{A})} = A$.

exercise: Proof of the De Morgan Laws

Proof the De Morgan Laws (corollaries C10 - C11) of Table XVII.

Answer:

C10: $(A + B) \cdot (\bar{A} \cdot \bar{B}) = A \cdot \bar{A} \cdot \bar{B} + \bar{A} \cdot B \cdot \bar{B} = 0$ and
 $(A + B) + (\bar{A} \cdot \bar{B}) = (A + \bar{A} + B) \cdot (A + B + \bar{B}) = 1$
 Apparently $(\bar{A} \cdot \bar{B})$ is the inverse of $(A + B)$ and we have already proven before that only one such inverse exist. Therefore,
 $(A + B) = (\bar{A} \cdot \bar{B})$
 NoS=PoN (negation of sum is product of negations).

C11: $(A \cdot B) \cdot (\bar{A} + \bar{B}) = A \cdot \bar{A} \cdot \bar{B} + A \cdot B \cdot \bar{B} = 0$ and
 $(A \cdot B) + (\bar{A} + \bar{B}) = (A + \bar{A} + \bar{B}) \cdot (\bar{A} + B + \bar{B}) = 1$
 Apparently $(\bar{A} + \bar{B})$ is the inverse of $(A \cdot B)$ and we have already proven before that only one such inverse exist. Therefore,
 $(A \cdot B) = (\bar{A} + \bar{B})$
 NoP=SoN (negation of product is sum of negations).

3.3 Formal derivation of the truth tables

In the above, we already knew where this was leading too; we knew that the operations $+$ and \cdot represented the well-known Aristotelean/Boolean logic

operations OR and AND, respectively. However, the more formal derivation of what the operations actually are can be done on basis of the Huntington postulates themselves, without any further knowledge. This works as follows: Imagine we do not know what the operations are, so we name them simply OPX and OPY. Without loss of generality, we can call the two elements '0' and '1', without telling what they actually mean; they are just symbols (a circle and a stick). The Huntington postulates then become:

H1a	$\forall A, B \in \{0, 1\} : A \text{ OPX } B \in \{0, 1\}$
H1b	$\forall A, B \in \{0, 1\} : A \text{ OPY } B \in \{0, 1\}$
H1c	$\forall A \in \{0, 1\} : A \in \{0, 1\}$
H2a	$A \text{ OPX } B = B \text{ OPX } A$
H2b	$A \text{ OPY } B = B \text{ OPY } A$
H3a	$A \text{ OPX } (B \text{ OPX } C) = (A \text{ OPX } B) \text{ OPX } C$
H3b	$A \text{ OPY } (B \text{ OPY } C) = (A \text{ OPY } B) \text{ OPY } C$
H4a	$A \text{ OPX } (B \text{ OPY } C) = (A \text{ OPX } B) \text{ OPY } (A \text{ OPX } C)$
H4b	$A \text{ OPY } (B \text{ OPX } C) = (A \text{ OPY } B) \text{ OPX } (A \text{ OPY } C)$
H5a	$A \text{ OPY } 1 = A$
H5b	$A \text{ OPX } 0 = A$
H6a	$A \text{ OPX } \bar{A} = 1$
H6b	$A \text{ OPY } \bar{A} = 0$

In the first step we prove that $\bar{0} = 1$ by the following steps:

- 1) $\bar{0} \in \{0, 1\}$ (H1c), in other words
 $\bar{0} = 0$ or $\bar{0} = 1$
- 2) $0 \text{ OPX } \bar{0} = 1$ (H6a, with $A = 0$), but also
- 3) $0 \text{ OPX } 0 = 0$ (H5b, with $A = 0$), therefore
- 4) $\bar{0} \neq 0$ therefore
 $\bar{0} = 1$

The same way we can prove that $\bar{1} = 0$. Substituting this in H6 we get

H6a(0):	$0 \text{ OPX } 1 = 1$
H6a(1):	$1 \text{ OPX } 0 = 1$
H6b(0):	$0 \text{ OPY } 1 = 0$
H6b(1):	$1 \text{ OPY } 0 = 0$

We can also write out the postulates H5 explicitly:

H5a(0):	0 OPY 1 = 0*
H5a(1):	1 OPY 1 = 1
H5b(0):	0 OPX 0 = 0
H5b(1):	1 OPX 0 = 1*
*: equal to H6 above	

We have here already six of the eight entries of our two truth tables. The final entries can be found by applying the postulates, and find once again some useful corollaries. The same way we did before. Namely:

C1: $A \text{ OPY } A = A$, because:

$$A \text{ OPY } A \stackrel{H5b}{=} A \text{ OPY } A \text{ OPX } 0 \stackrel{H6b}{=} (A \text{ OPY } A) \text{ OPX } (A \text{ OPY } \bar{A}) \stackrel{H4b}{=} A \text{ OPY } (A \text{ OPX } \bar{A}) \stackrel{H6a}{=} A \text{ OPY } 1 \stackrel{H5a}{=} A$$

C2: $A \text{ OPX } A = A$, because:

$$A \text{ OPX } A \stackrel{H5a}{=} (A \text{ OPX } A) \text{ OPY } 1 \stackrel{H6a}{=} (A \text{ OPX } A) \text{ OPY } (A \text{ OPX } \bar{A}) \stackrel{H4a}{=} A \text{ OPX } (A \text{ OPY } \bar{A}) \stackrel{H6b}{=} A \text{ OPX } 0 \stackrel{H5b}{=} A$$

This means that

C1(0)	0 OPY 0 = 0
C1(1)	1 OPY 1 = 1*
C2(0)	0 OPX 0 = 0*
C2(1)	1 OPX 1 = 1
*: equal to H5 above	

Arriving at the final truth tables of our two operations OPX and OPY as shown below:

A	B	OPX	A	B	OPY
0	0	0	0	0	0
0	1	1	0	1	0
1	0	1	1	0	0
1	1	1	1	1	1

And we see that this is Boolean logic *if* $0 \equiv \text{false}$, $1 \equiv \text{true}$, $\text{OPX} \equiv \text{OR}$, and $\text{OPY} \equiv \text{AND}$. Actually, we might just as well have said that $0 \equiv \text{true}$, $1 \equiv \text{false}$, $\text{OPX} \equiv \text{AND}$, and $\text{OPY} \equiv \text{OR}$; it would work just as well. Once again, the mathematician nor the computer engineer cares about whether this follows (Aristotelean) 'logic'. It is simply the creation of a (virtual) world (in this case truth tables) on basis of axioms (postulates) for the mathematician, and the implementation of these truth tables in electronics for the engineer, where in the latter a mapping is done as in $0 \equiv 0$ volt and 1

$\equiv 5$ volt (in transistor-transistor logic [TTL]) or 3.3 volt (in complementary metal-oxide-semiconductor logic [CMOS]).



For curiosity’s sake we end this section by mentioning that it is apparently not possible to make a similar derivation for ternary logic. (The postulates only work for 2^n -logic). Or as Robert Jay Thomas writes*,

Since any Boolean algebra must have 2^n elements, where n is a natural number, such an algebra cannot be a Boolean algebra. Therefore, if we wish to have an algebra, or logic, in which the variables are three valued, we must develop another system.

To give you an idea, as mentioned before, there are 16 (2^4) possible two-inputs binary (note: $2^2 = 4$, the number of lines in a truth table) logic gates. We could still do the analysis by exhaustive search. In the same way we can see that with there are $3^2 = 9$ lines of a truth table of a two-input ternary gate. With each value having three possibilities, there are $3^9 = 19,683$ possible logic gates and an exhaustive brute-force analysis is prohibitive. Nor is a mathematical derivation possible, as stated by Thomas.

So, the solution will be given by declaration. The system mostly presented is the Kleene logic of indeterminacy, also known as Priest’s logic of paradox. It can use the values -1 , 0 and $+1$ for representing ‘false’, ‘undetermined’, and ‘true’, respectively, so called balanced ternary values. It can also use unbalanced ternary values 0 , 1 and 2 , already prepared for arithmetic, in line with our numbers systems presented in Chapter 2 and probably more adequate for computing. The truth tables (in 3×3 form) for unbalanced ternary are given by,

					<table><tr><td>A</td><td>\bar{A}</td></tr><tr><td>0</td><td>2</td></tr><tr><td>1</td><td>1</td></tr><tr><td>2</td><td>0</td></tr></table>					A	\bar{A}	0	2	1	1	2	0
A	\bar{A}																
0	2																
1	1																
2	0																
MIN		B			MAX		B										
(TAND)		0	1	2	(TOR)		0	1	2								
A	0	0	0	0	A	0	0	1	2								
	1	0	1	1		1	1	1	2								
	2	0	1	2		2	2	2	2								

*Robert Jay Thomas, "Classification and properties of three values logics". Thesis University of Illinois (1964).

XOR		B			IMP		B		
(TXOR)		0	1	2			0	1	2
A	0	0	1	2	A	0	2	2	2
	1	1	1	1		1	1	1	2
	2	2	1	0		2	0	1	2

These implement the negation, minimum, maximum, exclusive or, and material implication, respectively. (Note: material implication does not follow postulate 2, the commutative law, that would require that $(A \text{ IMP } B) = (B \text{ IMP } A)$; the truth table in 3×3 format is not symmetric, and IMP cannot serve as an elementary ternary gate.

It is for the reader to take it from there if interested in ternary logic. To my knowledge, however, it is not possible to implement all possible logic functions on basis of these elementary truth tables, in contrast to binary logic. An example is the equivalent of the binary half-adder (to be presented in the chapter on integration, Ch. 5) with outputs of sum Σ and carry c would be

Ternary half adder:

A	B	Σ	c
0	0	0	0
0	1	1	0
0	2	2	0
1	0	1	0
1	1	2	0
1	2	0	1
2	0	2	0
2	1	0	1
2	2	1	1

Σ		B		
		0	1	2
A	0	0	1	2
	1	1	2	0
	2	2	0	1

c		B		
		0	1	2
A	0	0	0	0
	1	0	0	1
	2	0	1	1

An empiric search with a maximum of 8,000 ternary logic gates done by the author did not result in a realization of this table. In practice, ternary logic is often implemented by decoding the ternary digits (trits) to binary, doing the computation in binary and then encoding the result in ternary. The solution for the half adder will be given along the binary half-adder in Chapter 5 on page 113.

The rest of the book is about binary logic only. Note that this ternary logic has nothing to do with tri-state to be discussed later. Tri-state of a logic gate means it is not generating *any* logical value at its output. It is electrically disconnect from the rest of the circuit. Ternary gate means the output can be any of three values.

We end this section by a funny observation, why boolean yes/no logic is not always adequate:

The judge was asking the defendant, "Yes or no. Did you, or did you not stop beating your wife?"

A third option seems in place here . . .

3.4 From postulates to truth tables

The next step is to see how *any* logic can be implemented by the three basic Huntington operations NOT, AND and OR. For that we go back to the truth tables introduced in the beginning of the chapter. In these, the logic output value for all possible combinations of logic input values is summarized. Moreover, it is convention to do this in such a way that the input combinations, when read as a binary number, are ordered from low to high. For example, if the logic operation (or logic gate) has three inputs, a , b and c , and we assume that a is the MSB (most significant bit) and c the LSB (least significant bit), then the first line of the truth table presents the output value for $a = 0, b = 0, c = 0$, the second line gives the output value for $a = 0, b = 0, c = 1$, the third line for $a = 0, b = 1, c = 0$, etc.

With the Huntington postulates we will first construct the truth table for AND, OR and NOT operations. Note that this seems the opposite way of doing things, namely we expect that the truth tables are describing the operations of logic gates from which we see that they follow the Huntington postulates. But don't forget that we followed here the reasoning of building *from* mathematics and not *to* mathematics. Here we will construct the truth tables of the three fundamental operations OR, AND and NOT, which should, if everything goes well, reproduce Table XIV.

Let's first fill the truth table for the OR operation (+). From the table of useful corollaries (Table XVII; chose a suitable value for A) we see that $0 + 0 = 0$, $0 + 1 = 1$, $1 + 0 = 1$, and $1 + 1 = 1$, which is indeed what we found in the beginning of the chapter. Let's do the same for the AND operation (\cdot). We find from the corollaries table that $0 \cdot 0 = 0$, $0 \cdot 1 = 0$, $1 \cdot 0 = 0$, and $1 \cdot 1 = 1$, which is once again the same result as found in the beginning of the chapter. Finally, the NOT operation, knowing that $A \cdot \bar{A} = 0$ and $A + \bar{A} = 1$, and looking at the useful correlations table, we derive that $\bar{0} = 1$ and $\bar{1} = 0$, which was already presented in the truth tables. We conclude that the Huntington postulates are apparently indeed a good basis for implementing Aristotelean logic. And the truth tables summarize the operations based on the Huntington postulates.

3.5 Implementing any logic (function); sum-of-products (SoP) and product-of-sums (PoS)

On basis of the three fundamental operations of the Huntington postulates (AND, OR, NOT) we can build *any* logic. Any truth table of any number of input variables (so any logic gate with any number of inputs) can be

Table XIX: Minterms and Maxterms for a triple-input logical function.

a	b	c	minterm	Maxterm
0	0	0	$m_0 = \bar{a}\bar{b}\bar{c}$	$M_0 = (a + b + c)$
0	0	1	$m_1 = \bar{a}\bar{b}c$	$M_1 = (a + b + \bar{c})$
0	1	0	$m_2 = \bar{a}b\bar{c}$	$M_2 = (a + \bar{b} + c)$
0	1	1	$m_3 = \bar{a}bc$	$M_3 = (a + \bar{b} + \bar{c})$
1	0	0	$m_4 = a\bar{b}\bar{c}$	$M_4 = (\bar{a} + b + c)$
1	0	1	$m_5 = a\bar{b}c$	$M_5 = (\bar{a} + b + \bar{c})$
1	1	0	$m_6 = ab\bar{c}$	$M_6 = (\bar{a} + \bar{b} + c)$
1	1	1	$m_7 = abc$	$M_7 = (\bar{a} + \bar{b} + \bar{c})$

made. While it does not always lead to the simplest form, and we will actually immediately afterwards learn how to simplify the result, we will now present two similar methods that will lead to a successful Boolean-algebraic expression of any truth table. They are called sum-of-products (SoP) and product-of-sums (PoS), respectively. This nomenclature is based on the fact that the AND operator (\cdot) resembles a product symbol and the OR operator ($+$) a sum symbol. A sum-of-products is thus, in fact, an OR of ANDs and a product-of-sums is an AND of ORs.

In the sum-of-products (SoP) method, we write in the truth table at every line the (unique) 'product' of all variables that results in 1 for that input combination. So, for instance for a triple input (a, b, c) logic function only the product $\bar{a} \cdot \bar{b} \cdot \bar{c}$ is 1 for $a = 0$, $b = 0$, and $c = 0$, the first line of the truth table. The unique product is called a 'minterm' m and all the minterms are summarized in Table XIX, labeled by the numeric value of the binary inputs: m_0 belongs to $abc = 000 = 0_{10}$, etc. Note that from now on we will no longer write the AND-operator (\cdot) and assume it to be implicit any time two variables are joined. So, for example, ab implies $a \cdot b$, which implied a AND b .

To find the logic implementation of any function by the SoP method, simply 'sum' (OR) all minterms that have a 1 in the corresponding line of the truth table. An example is given in Table XX. To further abbreviate the function, often the convention is used to name all minterms by Σ , as shown in the table. We can think of every minterm introducing one 1 in a truth table that started out 'empty' (with only 0s). In other words, the Σ notation lists all lines of the truth table with a 1.

Similarly, a solution by the PoS method can be found by starting the place in the truth table on every line a 'Maxterm' that represents a sum that is uniquely 0 for those input values. For instance, for $a = 0$, $b = 0$, $c =$, the Maxterm M_0 is $(a + b + c)$. In our final Boolean algebra expression we

Table XX: Example of SoP and PoS solutions of a logic function F .

a	b	c	F	minterm	Maxterm
0	0	0	0		$M_0 = (a + b + c)$
0	0	1	1	$m_1 = \bar{a}\bar{b}c$	
0	1	0	1	$m_2 = \bar{a}b\bar{c}$	
0	1	1	1	$m_3 = \bar{a}bc$	
1	0	0	0		$M_4 = (\bar{a} + b + c)$
1	0	1	0		$M_5 = (\bar{a} + b + \bar{c})$
1	1	0	1	$m_6 = ab\bar{c}$	
1	1	1	0		$M_7 = (\bar{a} + \bar{b} + \bar{c})$

$$\begin{aligned}
 F_{\text{SoP}} &= \Sigma(1, 2, 3, 6) = m_1 + m_2 + m_3 + m_6 \\
 &= \bar{a}\bar{b}c + \bar{a}b\bar{c} + \bar{a}bc + ab\bar{c} \\
 F_{\text{PoS}} &= \Pi(0, 4, 5, 7) = M_0 M_4 M_5 M_7 \\
 &= (a + b + c)(\bar{a} + b + c)(\bar{a} + b + \bar{c})(\bar{a} + \bar{b} + \bar{c})
 \end{aligned}$$

simply 'multiply' (AND) all Maxterms that have a 0 in the corresponding line of the truth table. Table XX also shows a PoS solution to the same function F used for the SoP solution. Also here a further abbreviation is used by naming all Maxterms by Π , as shown in the table. We can think of every Maxterm introducing one 0 in the truth table that started out 'full' (with only 1s). And the Π shorthand notation lists all lines of the truth table with a 0. It is therefore obvious that Σ and Π are complementary notations, since a line has either a 1 or a 0, making it contribute to either Σ or Π , but not both.

Although these procedures assuredly result in algebraic solutions (and, as we will see later, electronic solutions as well), it does not mean that this is the unique solution or even the best solution. Without much work they can readily be simplified by using Karnaugh maps, as we will see in the next section.

Note that the triple-input AND-operations are, in fact, two double-input AND-operations. For example, $abc = a \cdot (b \cdot c)$ or $(a \cdot b) \cdot c$, the two being similar, as demonstrated by the Huntington postulates. Likewise, the triple-input OR-operations are two double-input OR-operations. For example, $(a + b + c) = (a + b) + c$ or $a + (b + c)$. That is because Boolean algebra, as built on the Huntington postulates, use double-operand operations only. As we will see later, real electronic gates can, however, have multiple-operand versions of the operations.



Having said all this, and having shown that any logic can be made from

AND, OR and NOT operations, we will see that in reality in CMOS technology (complementary metal-oxide-semiconductor) a much better implementation is by only NAND and NOR gates. That is because AND and OR gates take 6 transistors each, while NAND and NOR gates can be made of only 4 transistors. In fact, AND and OR gates, as we will see, are simply NAND and NOR gates, respectively, with a two-transistor inverter added at the output.

Thus we can save transistors using De Morgan's laws. As an example, the SoP operation of four inputs, a, b, c, d :

$$(a \text{ AND } b) \text{ OR } (c \text{ AND } d)$$

can easily be converted into a NAND-only expression:

$$\begin{aligned} (a \cdot b) + (c \cdot d) &= \overline{\overline{(a \cdot b) + (c \cdot d)}} \\ &= \overline{\overline{(a \cdot b)} \cdot \overline{(c \cdot d)}} \end{aligned}$$

and that is

$$(a \text{ NAND } b) \text{ NAND } (c \text{ NAND } d).$$

In general, a SoP can be converted into

$$\Sigma_j (\Pi_i x_i)_{\text{j}} = \overline{\Pi_j (\overline{\Pi_i x_i})_{\text{j}}}.$$

Likewise, a PoS can be converted into only NOR operations. For example,

$$(a \text{ OR } b) \text{ AND } (c \text{ OR } d)$$

can easily be converted into only NAND operations using the De Morgan's laws:

$$\begin{aligned} (a + b) \cdot (c + d) &= \overline{\overline{(a + b) \cdot (c + d)}} \\ &= \overline{\overline{(a + b)} + \overline{(c + d)}} \end{aligned}$$

and that is

$$(a \text{ NOR } b) \text{ NOR } (c \text{ NOR } d).$$

In general, a PoS can be converted into

$$\Pi_j (\Sigma_i x_i)_{\text{j}} = \overline{\Sigma_j (\overline{\Sigma_i x_i})_{\text{j}}}.$$

Igoring the NOT operations for a moment, this method reduces the number of transistors in CMOS technology from $6 \times (I + J)$ to $4 \times (I + J)$, with I and J the number of input variables and minterms/Maxterms respectively.

3.6 Karnaugh maps

The task of an engineer is to minimize the number of transistors and the number of steps in a logical circuit. In general we must reduce complexity of the hardware to lower the cost and increase the speed. We do this by reducing the mathematical complexity. Find simpler solutions to the same logical functions. The first step of simplification was found in the previous section. By using the De Morgan's laws we could convert logic to NAND-operations-only and NOR-operations-only (as we will see in the next chapter, NAND and NOR gates are simpler than their AND and OR siblings). In this section we will see how using Karnaugh maps can help us further reduce complexity.

The method of using Karnaugh maps are based on properties of the Boolean algebra. As an example, take the logic function

$$F = A \cdot B \cdot C + \overline{A} \cdot B \cdot C.$$

The distributive law tells us that this is equal to

$$F = (A + \overline{A}) \cdot B \cdot C,$$

and the expression inside the parentheses is equal to the element 1. Thus

$$F = 1 \cdot B \cdot C = B \cdot C.$$

We have thus completely eliminated signal A from the logic. This very much simplifies the final logic. The beginning expression had four AND-operations and one OR-operation. The final expression can be made with only one AND-operation.

The question is now, how to find such simplifications in more complex Boolean algebraic functions? Here is where the Karnaugh maps come in handy. What we will do is place the output values of the truth table in a two-dimensional table, where *adjacent cells only differ by exactly one of the input terms*. An example is the following truth table of a 3-input logic function $F(A, B, C)$ given by

A	B	C	F	min	Max
0	0	0	0	$\overline{A}\overline{B}\overline{C}$	$(A + B + C)$
0	0	1	0	$\overline{A}\overline{B}C$	$(A + B + \overline{C})$
0	1	0	0	$\overline{A}B\overline{C}$	$(A + \overline{B} + C)$
0	1	1	1	$\overline{A}BC$	$(A + \overline{B} + \overline{C})$
1	0	0	0	$A\overline{B}\overline{C}$	$(\overline{A} + B + C)$
1	0	1	1	$A\overline{B}C$	$(\overline{A} + B + \overline{C})$
1	1	0	1	$AB\overline{C}$	$(\overline{A} + \overline{B} + C)$
1	1	1	1	ABC	$(\overline{A} + \overline{B} + \overline{C})$

We obtain the following Karnaugh map:

	A		\bar{A}	
B	1	1	1	0
\bar{B}	0	1	0	0
	\bar{C}	C	\bar{C}	

For instance, the value for the minterm ABC is 1, so we place a 1 at the corresponding cell of the Karnaugh map. This way we fill up the entire map. The trick is now this:

- For every neighboring pair of 1s we can eliminate a variable from part of the expression.

Take for example the two 1s highlighted below:

	A		\bar{A}	
B	1	1	1	0
\bar{B}	0	1	0	0
	\bar{C}	C	\bar{C}	

these represent the expression

$$ABC + A\bar{B}C$$

and B can be eliminated from this:

$$ABC + A\bar{B}C = AC(B + \bar{B}) = AC \cdot 1 = AC.$$

We can actually directly visually identify this term of the final expression in the Karnaugh map. Look again at it and identify that the loop falls fully within the A-half as well as the C-half, so the term must be AC .

We can repeat this for every pair of 1s we find. (NB: it does not matter if a 1 is already used in this approach in an expression; we can use it as often as we want). We can then also combine two pairs of pairs of 1s and remove even two variables from the expression. The final SoP Karnaugh map looks like this:

	A		\bar{A}	
B	1	1	1	0
\bar{B}	0	1	0	0
	\bar{C}	C	\bar{C}	

So that the original SoP expression

$$F = \overline{A}BC + A\overline{B}C + AB\overline{C} + ABC$$

is reduced to

$$F = AB + AC + BC,$$

5 gates instead of 11. That can further be reduced using de Morgan's laws as described above.

Likewise, we can construct the Karnaugh map for the PoS implementation. Note that 0s and 1s appear in different places of the map. That is because a line for, for example $A=0$, $B=0$, $C=0$ has a minterm $\overline{A}\overline{B}\overline{C}$ but a Maxterm $(A + B + C)$, note the inversions. Instead of redrawing the entire Karnaugh map, we can simply convert a SoP Karnaugh map to a PoS Karnaugh map by negating the variables along the edges of the table:

	\overline{A}	A	
\overline{B}	1	1	0
B	0	1	0
	C	\overline{C}	C

Now 'hunting' for 0s, the PoS

$$F = (A + B + C)(A + B + \overline{C})(A + \overline{B} + C)(\overline{A} + B + C)$$

can be reduced to

$$F = (A + B)(A + C)(B + C).$$

Note the cyclic character of a Karnaugh map. A zero on the bottom left was joined with a 0 on the bottom right. This is allowed because they also differ in just one variable, in this case the variable A . Also here we reduce the number of operations from 11 to 5.

*

To show the usefulness of Karnaugh maps we can also demonstrate some of the corollaries of Table XVII. This in case we forget our rules, but only remember Karnaugh maps. This is done in Figure 12.

*

In some cases the truth table can contain so-called 'don't-cares', often indicated with an x. We can imagine a circuit that drives a numerical display using BCD digits. The combinations 0xA (1010) to 0xF (1111) are irrelevant. It does not matter how the circuit would respond in these cases, since they can never occur and thus do not have to be taken into account. In these

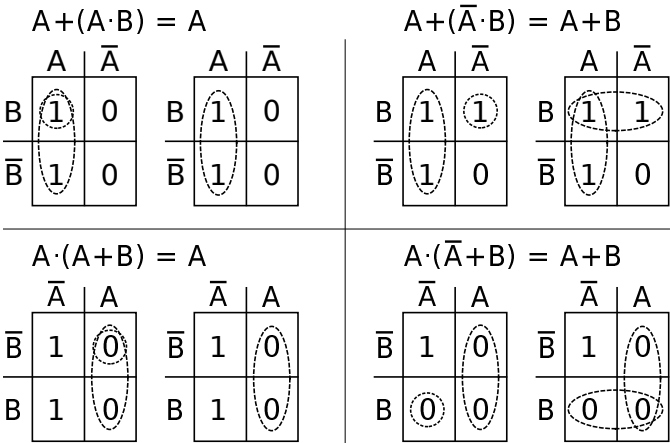


Figure 12: Prove of corollaries C5...C8 by use of Karnaugh maps.

cases we can substitute for the don't-care x a 0 or a 1, whichever comes in more handy.

Take for example a 7-segment display shown in Figure 13. The truth table for the seven display segments is

digit	binary <i>a b c d</i>	segment						
		A	B	C	D	E	F	G
0:	0 0 0 0	1	1	1	0	1	1	1
1:	0 0 0 1	0	0	1	0	0	1	0
2:	0 0 1 0	1	0	1	1	1	0	1
3:	0 0 1 1	1	0	1	1	0	1	1
4:	0 1 0 0	0	1	1	1	0	1	0
5:	0 1 0 1	1	1	0	1	0	1	1
6:	0 1 1 0	1	1	0	1	1	1	1
7:	0 1 1 1	1	0	1	0	0	1	0
8:	1 0 0 0	1	1	1	1	1	1	1
9:	1 0 0 1	1	1	1	1	0	1	1
A:	1 0 1 0	x	x	x	x	x	x	x
B:	1 0 1 1	x	x	x	x	x	x	x
C:	1 1 0 0	x	x	x	x	x	x	x
D:	1 1 0 1	x	x	x	x	x	x	x
E:	1 1 1 0	x	x	x	x	x	x	x
F:	1 1 1 1	x	x	x	x	x	x	x

The Karnaugh map of segment C of this 7-segment display is given in Fig. . 13. For x we can substitute 0 or 1. In this case we chose 0 for the two indicated and the rest 1, and we chose a product-of-sums approximation so

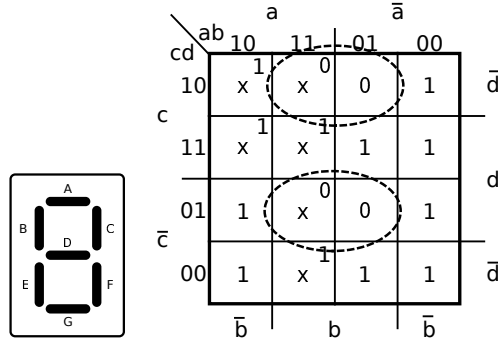


Figure 13: Seven-segment display and the Karnaugh map of segment C. For x we can substitute 0 or 1, whatever comes in handy. In this case we chose 0 for the two indicated and the rest 1, and we chose a product-of-sums approximation so that the final expression becomes $C = (b+c+\bar{d})(b+\bar{c}+d)$.

that the final expression becomes $C = (b + c + \bar{d})(b + \bar{c} + d)$.

✱

Finally, we can take a look here at the XOR (exclusive OR) operation, often written as \oplus . It is 1 for either A or B equal to one, but not both, that latter combination being excluded and making it different from a normal OR operation. Its truth table is thus given by

XOR		
A	B	$A \oplus B$
0	0	0
0	1	1
1	0	1
1	1	0

Formally applying the SoP approach, the XOR operation can be expressed as

$$A \oplus B = \bar{A}B + A\bar{B},$$

and this cannot be simplified using a Karnaugh map, since nowhere does it have neighboring 1s. Yet, looking at a Karnaugh map, we see the checkered chessboard structure. Thus, when we see checkered structures in the Karnaugh map, beware, possibly XOR operations can be used.

For completeness sake, all the SoP and PoS representations of the XOR

	A	\bar{A}
B	0	1
\bar{B}	1	0

Figure 14: Karnaugh map of an XOR operation (\oplus). Note the checkered pattern (highlighted by shading the fields containing 1).

and NOT-XOR operations are

$$\begin{aligned}
 A \oplus B &= \bar{A}B + A\bar{B} \\
 \overline{A \oplus B} &= AB + \bar{A}\bar{B} \\
 A \oplus B &= (A + B)(\bar{A} + \bar{B}) \\
 \overline{A \oplus B} &= (\bar{A} + B)(A + \bar{B}),
 \end{aligned}$$

which all have checkered Karnaugh maps. (Note we had to go back to the dot notation in the second one, because the negation of a product is not the same as the product of negations, $\overline{A \cdot B} \neq \bar{A} \cdot \bar{B}$, and without the dot, the text processor makes it into $\overline{AB} \neq \overline{A}B$, indistinguishable). Note also that this NOT-XOR is also called EQUAL (see Fig. 2), written as $A=B$, where the '=' symbol is an operator, not an attribution. It is 1 if either both A and B are 0 or both 1.

exercise: Comparator

Give a Boolean expression for a comparator that has two 2-bit inputs A (A_1, A_0) and B (B_1, B_0) and outputs 1 if $A < B$ and 0 otherwise.

Answer: The truth table of this comparator is given in Table XXI, which converts into the Karnaugh map of Figure 15. Using a SoP approach we get the Karnaugh map given in that figure and the expression $(A < B) = \bar{A}1B1 + \bar{A}0B1B0 + \bar{A}1A0B0$.

3.7 An alternative to Karnaugh maps

An alternative to the graphical method of Karnaugh maps is the Quine-McCluskey method (explained well by Donald Krambeck). It works as follows. Let's take the example of the sum-of-products described by

$$f(a, b, c, d) = \Sigma(0, 1, 2, 5, 6, 7, 8, 9, 10, 14)$$

First we group the terms in binary by the number of 1s in them. A group with zero 1s, a group with one 1, a group with two 1s, etc.:

Table XXI: Truth table of a 2-bit comparator.

A1	A0	B1	B0	A < B
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	1
0	1	1	1	1
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0

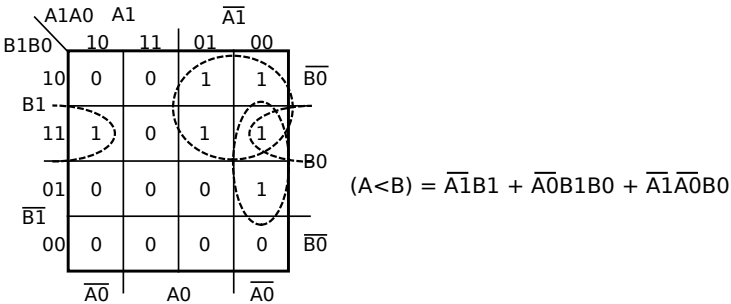


Figure 15: Karnaugh map and final expression of a 2-bit comparator.

1s	min	binary
	term	abcd
gr. 0	(0)	0000
gr. 1	(1)	0001
	(2)	0010
	(8)	1000
gr. 2	(5)	0101
	(6)	0110
	(9)	1001
	(10)	1010
gr. 3	(7)	0111
	(14)	1110

We will again try to find min-terms that differ by exactly one variable. It is obvious that members of the same group will differ by at least two variables. Compare for instance term (1) with term (2) of group 1. They differ in the last two bits, c and d . Likewise, groups that are not adjacent also differ in more than one bit. We thus only have to look at adjacent groups and find terms with one bit differing. These we then write in a new table, with the differing bit written as '-', and sort them again by number of 1s. For instance, (0)=0000 of group 0 combines with (1)=0001 of group 1 and is written as (0,1)=000- representing $\overline{a}\overline{b}\overline{c}$ and now belongs to group 0. The new table becomes:

1s	min terms	binary abcd
gr. 0	(0,1)	000-
	(0,2)	00-0
	(0,8)	-000
gr. 1	(1,5)	0-01
	(1,9)	-001
	(2,6)	0-10
	(2,10)	-010
	(8,9)	100-
	(8,10)	10-0
gr. 2	(5,7)	01-1
	(6,7)	011-
	(6,14)	-110
	(10,14)	1-10

Make sure that at all terms of the original table are at least used ones in the new table somehow, in the worst case it'd have to stay all by itself. We can repeat the procedure to get

1s	min terms	binary abcd
gr. 0	(0,1,8,9)	-00-
	(0,2,8,10)	-0-0
gr. 1	(2,6,10,14)	--10

No further combinations can be made and we wind up with the final expression

$$f(a, b, c, d) = \overline{b}\overline{c} + \overline{b}d + c\overline{d}$$

In case of existence of don't-cares the situation gets slightly more complex. We can use Petrick's method on top of the above method. Imagine we have the function

$$f(a, b, c, d) = \Sigma(\overline{1}, 2, 3, 7, 9, \overline{10}, 11, 13, \overline{15}),$$

with the terms with overline representing don't-cares. We again use the Quine-McCluskey method, with all terms, including the don't cares. In a compacted simplified table (to save space) we get:

(1) 0001	(1,3) 00-1	(1,3,9,11) -0-1
(2) 0010	(1,9) -001	(2,3,10,11) -01-
(3) 0011	(2,3) 001-	(3,7,11,15) --11
(9) 1001	(2,10) -010	(9,11,13,15) 1--1
(10) 1010	(3,7) 0-11	
(7) 0111	(3,11) -011	
(11) 1011	(9,11) 10-1	
(13) 1101	(9,13) 1-01	
(15) 1111	(10,11) 101-	
	(7,15) -111	
	(11,15) 1-11	
	(13,15) 11-1	

Or

$$f(a, b, c, d) = \bar{b}d + \bar{b}c + cd + ad$$

If we now draw a Petrick table, where we remove all columns of don't cares (1, 10, and 15) we get

	2	3	7	9	11	13
(1,3,9,11)		x		x	x	
(2,3,10,11)	x	x			x	
(3,7,11,15)		x	x		x	
(9,11,13,15)				x	x	x

We see that the second term, (2,3,10,11), must be included because it is the only one covering (2). We can then cross out the x in columns (3) and (11) in the other rows, because these are already covered by (2,3,10,11). We get the reduced Petrick table:

	2	3	7	9	11	13
(1,3,9,11)				x		
(2,3,10,11)	x	x			x	
(3,7,11,15)			x			
(9,11,13,15)				x		x

Likewise, (3,7,11,15) must be used for the (7) term, and (9,11,13,15) must be used for the (13) term. Crossing the x in the other rows that are already covered by these terms, we see that (1,3,9,11) is unnecessary since it has all x's crossed off, and we wind up with (2,3,10,11), (3,7,11,15) and (9,11,13,15), or

$$f(a, b, c, d) = \bar{b}c + cd + ad$$

exercise: Quine-McCluskey/Petrick/Karnaugh

Use Karnaugh maps to show the above two examples are indeed correct.

(Solution at the end of the chapter on page 69).

3.8 Exercises*exercise:* Boolean proof

1: Prove the following equations:

- a) $(\bar{a} + \bar{b})(a + b) = a\bar{b} + \bar{a}b \quad (\equiv a \oplus b)$
- b) $(ab + c)b = ab\bar{c} + \bar{a}bc + abc$
- c) $bc + ad = (b + a)(b + d)(a + c)(c + d)$

exercise: Logic complement

2: Find the complement of the following expressions:

- a) $F = a + \bar{b}c$
- b) $F = a(b + c) + b\bar{d}(\bar{a} + c)$

exercise: SoP and PoS

3: Find the truth table and SoP and PoS implementations (simplified by Karnaugh maps) of the following logical functions:

- a) $F = a + b\bar{c}$
- b) $F = ac + bc + ab$
- c) $F = (a + \bar{b})(\bar{a} + \bar{b} + c)$

exercise: Karnaugh maps: SoP

4: Using Karnaugh maps, find the simplified SoP function of:

- a) $F(a, b, c, d) = \Sigma(1, 3, 5, 7, 8, 10, 12)$
- b) $F(a, b, c) = \Sigma(0, 1, 3, 4, 6, 7)$

exercise: Karnaugh map: PoS

5: Using Karnaugh maps, find the simplified PoS (!) function of:

- a) $F(a, b, c) = \Sigma(0, 1, 2, 5, 7)$
- b) $F(a, b, c, d) = \Sigma(0, 1, 9, 10, 11)$

a) $F(a, b, c) = \Pi(0, 1, 2, 5, 7)$

exercise: Karnaugh map: don't cares

6: Using Karnaugh maps, find the simplified SoP functions of (underline is 'don't care'):

a) $F(a, b, c) = \Sigma(\underline{0}, 3, 5, \underline{7})$

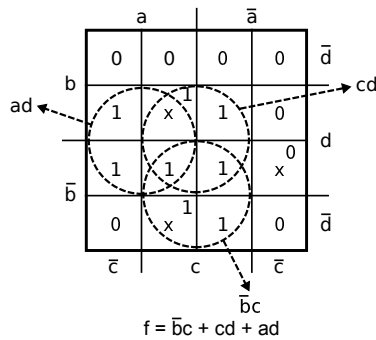
b) $F(a, b, c, d) = \Sigma(1, 2, 3, 5, 6, 7, \underline{9}, 10, 11, \underline{12}, \underline{15})$

exercise: Majority-vote

7: What is the logical expression for a majority vote-function ('true' if majority of inputs is 'true', otherwise 'false')? Use a truth table and a Karnaugh map to come up with your answer.

exercise: Priority circuit

8: A circuit has 4 inputs, $I_3 \dots I_0$. The 2-bit output (O_1, O_0) of the circuit is the binary representation of the highest input with a 1. For example, $O_1, O_0 = 10$ indicates that I_2 is the highest input with a 1, so $I_3 = 0, I_2 = 1, I_1 = x, I_0 = x$ (x is 'don't care'). Find the logic function that describes the circuit.



Solution of exercise on page 68.

4 | Hardware components

It is now time to take a look how the boolean logic (and thus numbers) is implemented in hardware. In principle we can implement the logic in many ways, and they do not necessarily be electronic. Take for example the Difference Engine of Charles Babbage (see Figure 16). It consisted of mechanical calculations performed on cylinders and will be discussed in the chapter with examples of architectures at the end of this book.

However, modern non-human calculations are mostly done by electronic computers of all sorts. We therefore have to start by explaining how the basic Boolean operations are achieved by electronics, and then we show how these basic boolean 'gates' can be combined to result in more sophisticated logical functions. We will then add feedback to components, to give them a 'memory' effect, at which stage we will talk about 'finite-state machines'. These components will then be further integrated to result in rudimentary calculation devices (arithmetic and logic units). Further integration will then lead to central processing units. This will then be placed on an electronic circuit board and make it communicate with other components and this is called a 'computer'. In summary, we will integrate components:

- Transistors

- Gates

- Logic circuits and memory elements

- Arithmetic and logic units (ALU)

- Central-processing units (CPU)

- Computers.

Starting with electronics, transistors.

4.1 Electronics: from transistors to gates

We have to keep in mind here what electronics is all about. While we can treat informatics fully formally, as a subject disconnected from its implementation, understanding the underlying layer will make us understand the layer

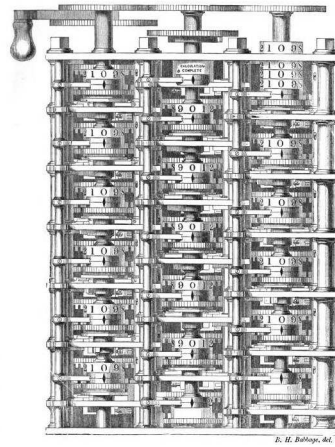


Figure 16: A portion of the Difference Engine of Charles Babbage. (Public domain image).

above much better. The underlying layer in most informatics equipment is electronics. Electronics is about charge (unit: coulomb, C) – that comes in two flavors, negative and positive – and its controlled movement. Electric charge being one of the fundamental properties of matter (apart from mass), the movement of this charge we call electric current (unit: ampere, A, which equals coulomb per second). To induce such a current, like movement of any matter, we have to apply a force to it and this force is called the Coulomb force, existing of attraction and repulsion by nearby charge. The associated energy, per unit charge, of this force is the electric potential, known as volt (V). Like with everything in nature, matter tries to minimize energy and thus charge attempts to move in the suitable direction. The question is, how well (fast) will it manage to do that. The easier it is, the faster the charge will move and the bigger the current. The inhibition of charge moving freely we call resistance, and if we can somehow control this resistance of electronic components, we can control the current. In comes the transistor.

A transistor, short for 'trans-resistor', is a three-pin component that let's the resistivity between two pins be regulated by a voltage on a third pin. We will skip here the entire description of the analysis of the analog behavior of such transistors, which is a subject of its own, and I happily refer the reader to my own book *Electronic Instrumentation*. We will assume here that a transistor has only two states (hence 'binary' or 'digital', with a digit being binary with only two possibilities, 0 or 1). A transistor either fully conducts (zero resistance) or doesn't conduct at all (infinite resistance), depending on the voltage at the controlling pin. Moreover, the resistance between the

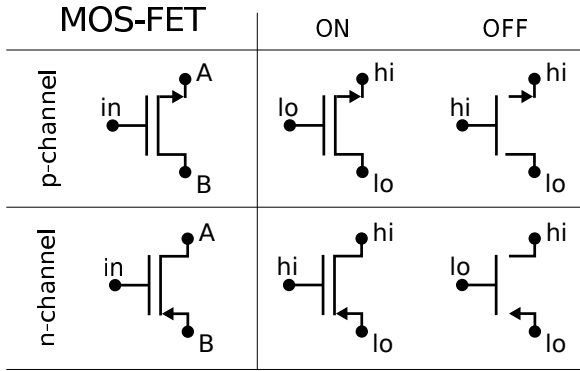


Figure 17: MOS-FET transistors of p-channel and n-channel type. The arrow points to higher voltage (and in direction of flow of negatively-charged electrons). To make a transistor stop conducting from A to B ('OFF'), we have to make the voltage at the in pin equal to the voltage of the pin with the arrow. To make it conduct ('ON') it should be equal to the other pin. The right image shows how to do that, 'hi' means high voltage (power supply), and 'lo' means low voltage (other power supply or ground).

controlling pin and the other two pins is infinite; never is there any 'leakage' of current or signal. We will actually confine our description to field-effect transistors (FETs), which come very close to this behavior. Especially we use MOS type. Metal oxide semiconductor.

Such transistors come in two types, which we can call n-channel and p-channel, see Figure 17. The n-channel transistors conduct when they have a layer of electrons attracted to the interface by the in-pin. Positive voltages attract electrons, so we have to make the in-pin positive, for instance by setting it equal to pin A, and pin B is at ground. In that case a current can flow between A and B. A p-channel conducts with positive charge (holes) and we have to attract them to the interface by a negative voltage, for instance equal to pin B, while pin a is at a high voltage. In summary: To switch the transistor off, we have to make the input pin equal to the pin with the arrow. To switch it on, we have to make the in-pin equal to the pin without the arrow. The arrow indicates this function in the transistor symbol. It also indicates the flow of electrons that is then opposite the direction of current.

With these two components we can build our elementary logic gates. As a first example, Figure 18 shows an inverter. It is composed of two transistors of complementary type, hence we call it CMOS technology. Complementary MOS. They are connected in cascade to the power supplies Vdd and Vss, for instance +5 V and 0 V. If the input voltage at 'in' is high (hi = +5 V, equal to Vdd), the top transistor Qp does not conduct, but the bottom transistor, Qn, does. That means that the output, 'out', is effectively shorted to Vss,

and that means low voltage (lo = 0 V). On the other hand, if the input voltage is low, the top transistor Qp conducts and the top one, Qn, not, making the output shorted to Vdd (hi = +5 V). In summary, the behavior is like this:

in	out
lo	hi
hi	lo

Or if we use the *convention* that low voltage is written down as '0' and high voltage as '1', we have the truth table

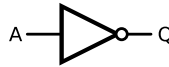
in	out
0	1
1	0

or as boolean values

in	out
false	true
true	false

This is the classical behavior of an inverter. In digital logic we call it a NOT-gate, and the output is labeled Q instead of out. For the symbol we will use a triangle (which represents a buffer, something that can supply the necessary current to the next stage), and a circle (which indicates the NOT-operation):

NOT	
A	Q
0	1
1	0



Note that in both cases there is no direct path from the top power supply (Vdd) to the bottom power supply (Vss). Either the n-channel is switched off, or the p-channel is. Nowhere is a current possible between the two supply sources. This reduces the overall power consumption drastically. Only power is used to drive the output. This is the great advantage of CMOS and the reason why it is the most-used technology in computer applications.

We can now build more complicated gates that have two input terminals. Figure 19 shows a NAND-gate made of four MOS-FET transistors (2 p-channel and 2 n-channel). To have the output connected to the lo(w) ('0') power supply, both n-channel transistors QnA and QnB, that are connected in cascade, have to be switched on with a hi(gh) ('1') at their inputs A and B. And in this case we use two p-channel transistors QpA and QpB that are both switched off by the same inputs A and B, to disconnect the hi(gh) power supply. If one of the inputs A or B is lo(w), or both, the connection

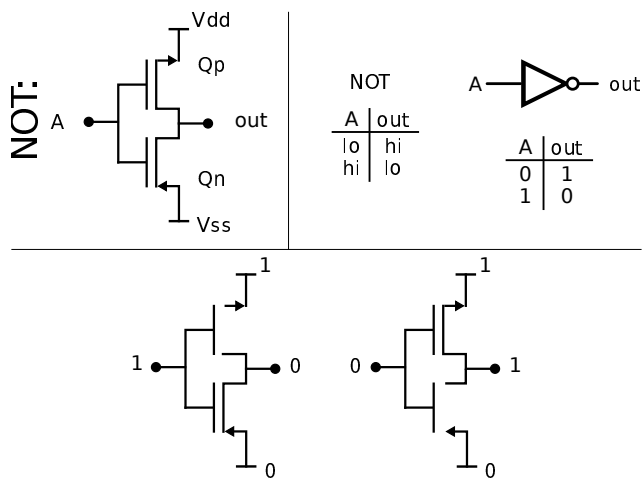



Figure 18: Left: An inverter NOT-gate based on two MOS-FET transistors. When the input is hi ('1'), the top transistor (p-channel) is switched off and the bottom transistor (n-channel) is switched on. (A situation shown in the bottom left). The result is a short circuit of the output to 0. With a lo ('0') at the input, the transistors switch roles and the output is connected to 1 (hi). (A situation shown in the bottom right). In CMOS, never a direct path exists between the two power supplies, Vdd and Vss, thus drastically reducing power consumption.

of out with the lo(w) power supply is lost and an connection of out with the hi(gh) power supply established through either QpA or QpB, or both. In other words, the truth table of this NAND circuit is

NAND		
A	B	Q
0	0	1
0	1	1
1	0	1
1	1	0



And the symbol is shown here as well. (From now on we will abbreviate hi(gh) voltage with '1' and lo(w) voltage with '0'). Once again, note that never is a direct path from Vdd to Vss possible. This is the quintessential aspect of CMOS technology.

Following this reasoning, 3-input NAND-gates can easily be made by adding another p-channel transistor QpC in parallel with QpA and QpB and another n-channel transistor QnC in cascade with QnA and QnB. The output of this circuit is lo (0) if all three n-channel transistors conduct and

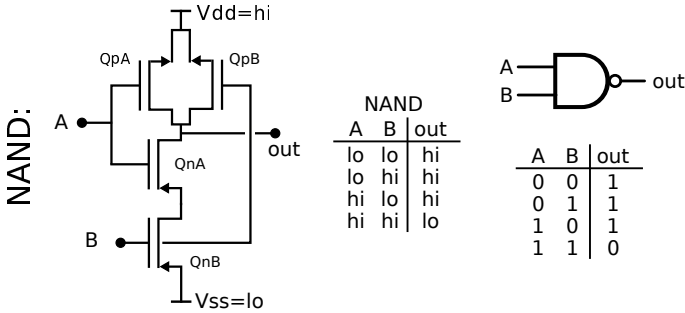
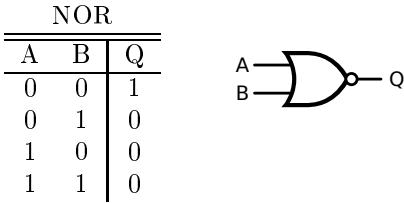


Figure 19: A NAND-gate composed of four MOS-FET transistors. To make the output lo, both inputs A and B have to be hi to switch on both n-channel transistors QnA and QnB. Otherwise the output is hi.

all three p-channel transistors are switched off, which only occurs for $A = B = C = 1$. We may even add more inputs and transistor pairs to get a general n -input NAND-gate, but there is a natural limit to this, for electronic reasons. Yet, we can say that an n -input NAND-gate can be made of $2n$ transistors.

Similar to a NAND-gate, Figure 20 shows a NOR-gate made of four MOS-FET transistors. When either input A or input B is hi (1), the output is connected to lo (0). Only if both inputs are low, both n-channel transistors (QnA and QnB) are switched off, and then both p-channel transistors that are connected in cascade are switched on and out is connected to hi (1). In other words, the truth table of this NOR circuit is



Just like n -input NAND-gates, equally easy is making an n -input NOR-gate from $2n$ transistors.

✱

AND and OR gates can simply be made from NAND and NOR gates, respectively, by placing an inverter at the output, see Figure 21. They thus consist of 6 transistors each.

Finally, to complete our list of basic binary gates, an XOR gate can be made of 14 transistors. This is not easy to show, but imagine we start with the SoP solution,

$$A \oplus B = \overline{A} \cdot B + A \cdot \overline{B}$$

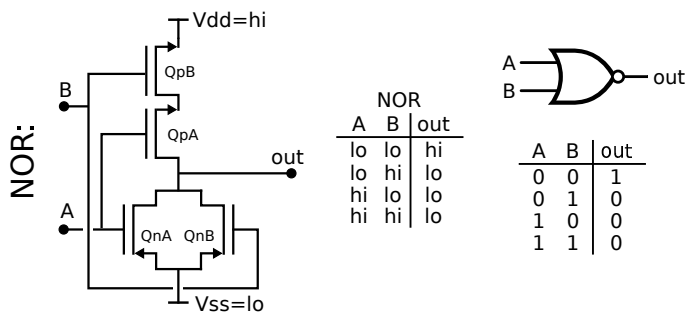


Figure 20: A NOR-gate composed of four MOS-FET transistors. To make the output hi, both inputs A and B have to be lo to switch on both p-channel transistors, QpA and QpB. Otherwise the output is lo.

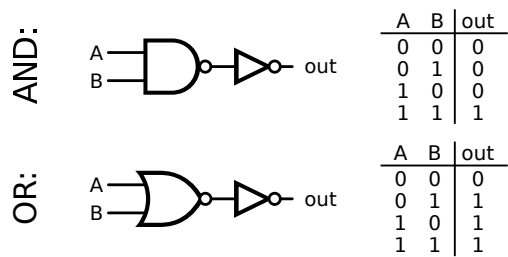


Figure 21: An AND-gate is a NAND-gate with an inverter at the output. An OR-gate is a NOR-gate with an inverter at the output. Both thus consist of 6 transistors.

This would imply two inverters, two AND-gates, and an OR-gate, with a total of 22 transistors. However, as we have seen, this can be converted by De Morgan’s laws into

$$A \oplus B = \overline{\overline{A \cdot B}} \cdot \overline{\overline{A \cdot B}},$$

two inverters and three NAND-gates, with a total of 16 transistors, see Figure 22. An XOR-gate is pronounced as ‘eks-or’ and not ‘ksor’.

Table XXII lists the 6 basic gates and the number of transistors each one is composed of. It is clear that when we can base circuits on NAND-gates and NOR-gates instead of AND-gates and OR-gates, this will save transistors and stages in the circuit. This will make them cheaper and faster.

The table also lists the number of layers of transistors each gate consists of. By this is meant the path with maximum number of transistors from inputs to output. For example, a NAND gate has 2 layers, because a path exists from input B to output through 2 transistors (QnB and QnA), and no longer paths exists. The number of layers determines the switching speed of

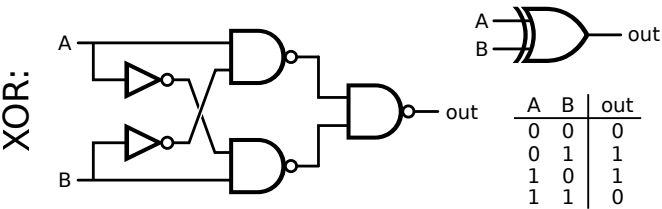


Figure 22: An XOR-gate made from three NAND-gates and two inverters with a total of 16 transistors.

Table XXII: Fundamental gates and the number of transistors it takes to make them and the number of transistor layers it has (CMOS technology).

Gate	Number of transistors	Number of layers
NOT	2	1
NAND	4	2
NOR	4	2
AND	6	3
OR	6	3
XOR	16	5
tri-state*	+4	+1

*: See Section 4.1.1

the gate. A transistor takes time to switch on or off. This has to do with drift speeds and diffusion speeds of charge carriers, and while they are different for electrons and holes, we assume them here the same, for simplicity’s sake. It is used to make an estimate about the relative speed of our circuits.

It has to be noted that an inverter (NOT-gate) can be made from NAND or NOR gates by making use of the idempotency rules found as corollaries of the Huntington postulates. Remember that these tell us that $A+A = A$ and $A\cdot A=A$. Thus $\overline{A} = \overline{(A+A)} = A \text{ NOR } A$, or $\overline{A} = \overline{(A\cdot A)} = A \text{ NAND } A$, see Figure 23. We can also see this in the circuits of NAND and NOR gates (Figure 19 and 20, resp.); connecting input B to A will make the two n-channel transistors Q_{nA} and Q_{nB} equivalent and effectively into a single transistor, likewise for the two p-channel transistors Q_{pA} and Q_{pB} . The gates become double-barreled NOT-gates. This trick does not work with AND and OR gates. Yet, it does mean that we could have based our Boolean algebra and Huntington postulates on only NAND and NOR operations. However, we prefer the Aristotelean approach of NOT, AND and OR operations, since that is how our mind works. Nature, however, prefers

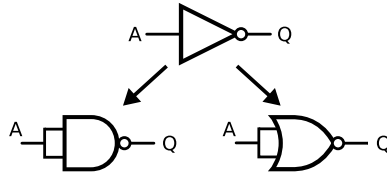


Figure 23: An inverter can be made from a NAND-gate or NOR-gate by simply linking the two inputs together.

NAND and NOR gates, since they are much simpler (See Table XXII).

4.1.1 Tri-state

In some cases we do not want the output to be hi, nor lo, but rather for it to be nothing at all. That is, it should not generate an output at all. This is especially the case when we have different output devices that all use the same line of communication, also called a 'bus'. Imagine the data bus between the CPU (central processing unit) and memory. Sometimes the CPU wants to output to the bus – talk – values that will be read by memory and sometimes the CPU wants to receive data – listen. Putting things on the bus is achieved by output gates as described above. But it cannot be that two or more devices output things on the bus at the same time, because it can cause a short-cut; imagine the CPU writing 0 and memory writing a 1. Current will directly flow from the Vdd power-supply of memory to the Vss power-supply of the CPU.

To avoid this problem, devices (output gates) can be put in so-called 'tri-state', which is neither 0 nor 1. This means they are disconnected from the power supply altogether and that is achieved by adding two complementary transistors between the logic gate and the power supplies. Figure 24 shows an example of a tri-state NAND-gate. When the enable line is lo (0), both power supply transistors are switched off, and regardless of the input states A and B of the logic gate, the output is in a non-determined, high-ohmic, tri-state. Only when the enable input line is high does the output from the logic gate appear at the output.

Figure 25 then shows how a tri-state bus can be made with this technique. Many components connected to the same physical line of communication.

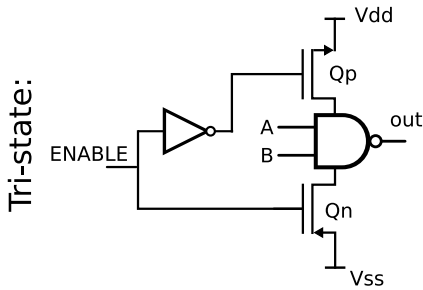


Figure 24: A NAND-gate with a control circuit allowing for the output to be in (high-ohmic) tri-state, out = Ω apart from lo (0) and hi (1).

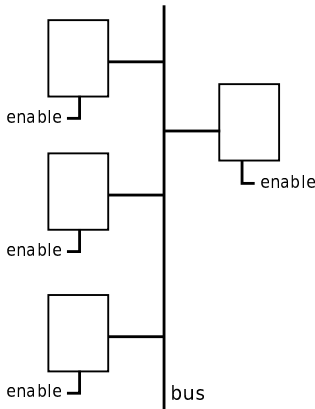


Figure 25: An example of a tri-state configuration. Several gates connected to the same communication line, also called a 'bus'. Only one component can be with an enabled output state. All the others must be in high-ohmic state, or there might occur a conflict, with one component placing a 0 on the bus and another a 1. There can be only one 'talker', but many 'listeners'.

tri-state NAND			
A	B	ENABLE	Q
0	0	0	Ω
0	0	1	1
0	1	0	Ω
0	1	1	1
1	0	0	Ω
1	0	1	1
1	1	0	Ω
1	1	1	0

Ω : undetermined

4.2 From gates to logic circuits

We can now build more advanced logic circuits based on these simple fundamental gates. For that we make use of the algebraic tricks learned in the previous chapter for finding Boolean expressions of any complexity. SoP, PoS and Karnaugh maps. We start by finding our Boolean expression and then take an AND-gate for every logic-AND operation and an OR-gate for every logic-OR operation. Thus, when we look at a truth table and the final SoP circuit we can see that there is a multi-input AND-gate for the lines in the table that have an output equal to 1 and then the outputs of all these first-layer AND-gates are fed into the multi-input-OR-gate to result in the final output Q. We thus have a layer of AND-gates and a layer of OR-gates:

- In a sum-of-products approach we use an n-input AND-gate for every line of the truth table that has a corresponding 1 at the output Q, and then feed the outputs of these gates to a n-input OR port. On every AND-gate, an input is fed directly into the gate if the value of the input is 1 and inverted if it is 0. In the truth table we can add a column with the corresponding boolean expression that implements that individual line, the so-called min-terms.

An example is the following truth table of a 3-input logic function $Q(A, B, C)$ given by

A	B	C	Q	min
0	0	0	0	$\overline{A} \cdot \overline{B} \cdot \overline{C}$
0	0	1	0	$\overline{A} \cdot \overline{B} \cdot C$
0	1	0	0	$\overline{A} \cdot B \cdot \overline{C}$
0	1	1	1	$\overline{A} \cdot B \cdot C$
1	0	0	0	$A \cdot \overline{B} \cdot \overline{C}$
1	0	1	1	$A \cdot \overline{B} \cdot C$
1	1	0	1	$A \cdot B \cdot \overline{C}$
1	1	1	1	$A \cdot B \cdot C$

As an example, the term $A \cdot \overline{B} \cdot \overline{C}$ is equal to 1 for – and only for – the input combination $A = 1$, $B = 0$, and $C = 0$. All other combinations result in 0. The total Boolean function Q is then given by ‘summing’ (OR’ing) all ‘products’ (and-gate-outputs, min-terms) that are preceded by a Q=1 in the table. In this case:

$$Q = \overline{A} \cdot B \cdot C + A \cdot \overline{B} \cdot C + A \cdot B \cdot \overline{C} + A \cdot B \cdot C,$$

which we can conventionally abbreviated as

$$\Sigma(011, 101, 110, 111),$$

or in decimal form

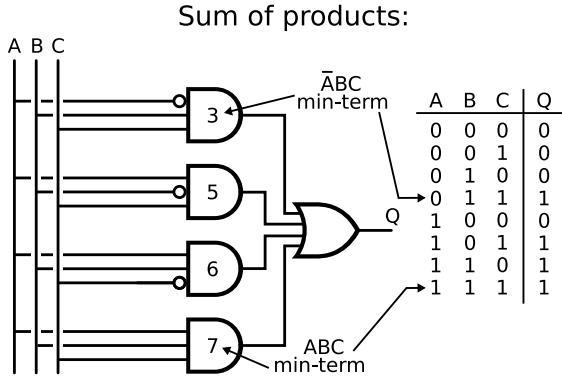


Figure 26: An example of a sum-of-products approach to find a circuit to implement any logical function, in this case $Q = \Sigma(3, 5, 6, 7) = \bar{A} \cdot B \cdot C + A \cdot \bar{B} \cdot C + A \cdot B \cdot \bar{C} + A \cdot B \cdot C$. Every line in the truth table with $Q=1$ results in a 'product' AND-gate (with those inputs negated that have a corresponding 0 in the input table) that implements a single min-term. The outputs of these AND-gates are then 'summed' by an OR-gate.

$$\Sigma(3, 5, 6, 7).$$

The resulting SoP circuit is then shown in Figure 26. Note that these n-input AND-gates and OR-gates can be constructed from binary 2-input AND-gates and OR-gates respectively, by placing them in series: $A \cdot B \cdot C = (A \cdot B) \cdot C$, etc. Actually, looking again at the basic circuit of a NAND-gate (Fig. 19), we can see that by adding another p-channel transistor Q_{pC} in parallel to the two existing p-channel transistors Q_{pA} and Q_{pB} , and an n-channel transistor Q_{nC} in series with the two already there (Q_{nA} and Q_{nB}), easily a 3-input NAND-gate can be made from 6 transistors. In this way, theoretically, n-input NAND-gates can be made, but there is a certain limit to it.

Finally, remember the De Morgan's law that stated that the negation of a product is the sum of negations. This can also be expressed in electronic form, as is shown in Figure 27. By placing an inverter at the output of every AND-gate (turning them into NAND-gates) and simultaneously placing a canceling inverter at every input of the OR-gate, thus having a net zero total effect, we can use this law to convert the resulting all-negated OR-gate into a NAND-gate as well. For our circuit it is mathematically

$$\begin{aligned}
 m_3 + m_5 + m_6 + m_7 &= \overline{\overline{m_3}} + \overline{\overline{m_5}} + \overline{\overline{m_6}} + \overline{\overline{m_7}} \\
 &= \overline{\overline{m_3} \cdot \overline{m_5} \cdot \overline{m_6} \cdot \overline{m_7}},
 \end{aligned}$$

where each minterm is an AND-operation of a combination of inputs at the original AND-gate. This will save transistors; as we have seen from Table

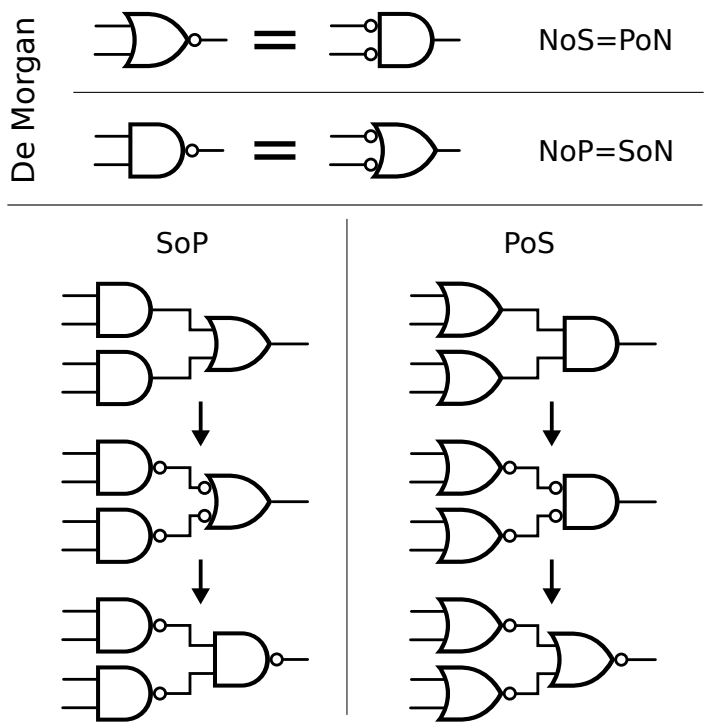


Figure 27: The De Morgan's laws in circuit form shown here for binary gates: the negation of a sum is the product of negations (NOS=PoN) and the negation of a product is the sum of negations (NoP=SoN). With this we can convert our sum-of-products (SoP) solution found with the Karnaugh maps into a NAND-gate only circuit, thus reducing transistors. We simply insert double negators (NOT NOT X = X) between the layers of the circuits and use De Morgan's law to convert the OR-NOT at the second layer into a NAND. Likewise, a product-of-sums (PoS) solution can be converted to a NOR-gate only circuit.

XXII, NAND-gates contain fewer of them. Our final NAND-only circuit is shown in Figure 28.

The same function can also be implemented using a product-of-sums approach. In this case, we use an OR-gate for every line in the truth table that has an output Q=0, with the signal of an inputs that is 1 inverted. We use an OR-gate for every Max-term

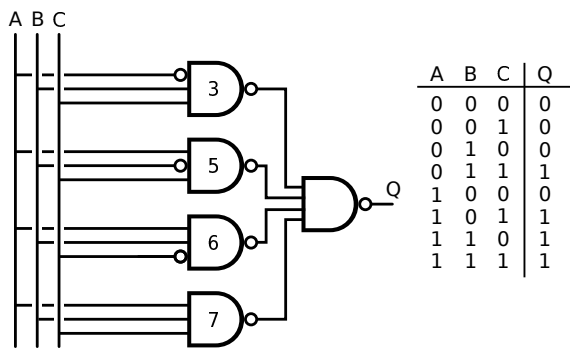


Figure 28: The circuit of Fig. 26 with the gates all converted into NAND-gates using the De Morgan’s law.

A	B	C	Q	Max
0	0	0	0	$(A+\overline{B}+\overline{C})$
0	0	1	0	$(A+\overline{B}+\overline{C})$
0	1	0	0	$(A+\overline{B}+\overline{C})$
0	1	1	1	$(A+\overline{B}+\overline{C})$
1	0	0	0	$(\overline{A}+B+\overline{C})$
1	0	1	1	$(\overline{A}+B+\overline{C})$
1	1	0	1	$(\overline{A}+\overline{B}+C)$
1	1	1	1	$(\overline{A}+\overline{B}+C)$

As an example, the term $A+\overline{B}+\overline{C}$ is equal to 0 for – and only for – the input combination $A=0, B=1$, and $C=1$. All other combinations result in 1. The total Boolean function Q is then given by ‘multiplying’ (AND’ing) all ‘sums’ (OR-gate-outputs, Max-terms) that are preceded by a $Q=0$ in the table:

$$Q = (A+B+C) \cdot (A+B+\overline{C}) \cdot (A+\overline{B}+C) \cdot (\overline{A}+B+C)$$

which we can conventionally abbreviated as

$$\Pi(000, 001, 010, 100),$$

or in decimal form

$$\Pi(0, 1, 2, 4).$$

This abbreviation can be found by indicating the binary values represented by A,B,C that produce an output equal to 0 (C is the least significant bit). The resulting circuit is now given by Figure 29. Note that these n -input OR-gates and AND-gates can be constructed from binary 2-input OR-gates and NAND-gates respectively, by placing them in series: $A+B+C = (A+B)+C$, etc.

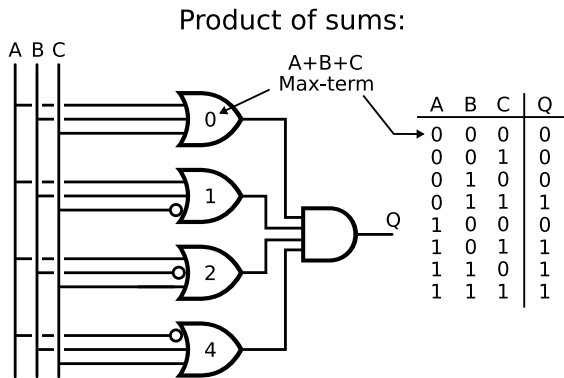


Figure 29: An example of a product-of-sums approach to find a circuit to implement any logical function, in this case $Q = \Pi(0, 1, 2, 4) = (A+B+C) \cdot (A+B+\bar{C}) \cdot (A+\bar{B}+C) \cdot (\bar{A}+\bar{B}+\bar{C})$. Every line in the truth table with $Q=0$ results in a 'sum' OR-gate (with those inputs negated that have a corresponding 1 in the input table) that implements a single Max-term. The outputs of these OR-gates are then 'multiplied' by an AND-gate.

Also this circuit can be simplified using the De Morgan's law that stated that the negation of a sum is the product of negations (see Figure 27). By placing an inverter at the output of every OR-gate (turning them into NOR-gates) and simultaneously placing a canceling inverter at every input of the second-layer AND-gate, thus having a net zero total effect, we can use this law to convert the resulting all-negated AND-gate into a NOR-gate as well, see Figure 30.

$$\begin{aligned}
 M_0 \cdot M_1 \cdot M_2 \cdot M_4 &= \overline{\overline{M_0}} \cdot \overline{\overline{M_1}} \cdot \overline{\overline{M_2}} \cdot \overline{\overline{M_4}} \\
 &= \overline{\overline{M_0} + \overline{M_1} + \overline{M_2} + \overline{M_4}},
 \end{aligned}$$

where each Maxterm is an OR-operation of a combination of inputs at the original OR-gate. Again, this will save transistors, as we have seen from Table XXII that NOR-gates contain fewer of them.

4.3 Karnaugh maps in electronics

The task of an engineer is to both minimize the number of transistors and the number of steps in an electronic circuit. The latter because every step causes a delay in the output; for physical reasons of charge-carrier diffusion times, etc., a transistor takes time to 'process' the input information and generate a state at the output. The transistor at the next stage can only start processing when its input signal is ready and stable at the output of

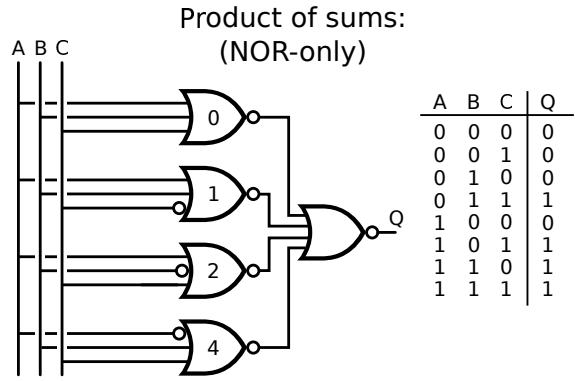


Figure 30: Circuit of Fig. 29 with the gates all converted into NOR-gates using the De Morgan’s law.

the previous stage. We must thus reduce the number of stages. The reason for reducing the number of transistors is obvious, since it will directly lower the size of the final circuit and thus its cost.

The first step of simplification was found in the previous section. By using the De Morgan’s law we could convert circuits to NAND-gate-only and NOR-gate-only circuits that have fewer transistors. In the chapter on Boolean algebra we have seen how the original SoP and PoS expressions can be simplified by the use of Karnaugh map. To give an example how this works in electronics,

$$Q = A \cdot B \cdot C + \overline{A} \cdot B \cdot C$$

The distributive law tells us that this is equal to

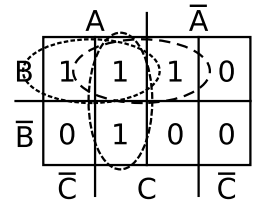
$$Q = (A + \overline{A}) \cdot B \cdot C$$

and the expression inside the parentheses is equal to the element 1. Thus

$$Q = 1 \cdot B \cdot C = B \cdot C.$$

We have thus completely eliminated signal A from the circuit! This very much simplified the final circuit. The beginning expression had 4 AND-gates and an OR-gate. The final expression can be made with only one AND-gate.

The final Karnaugh map looks like this:



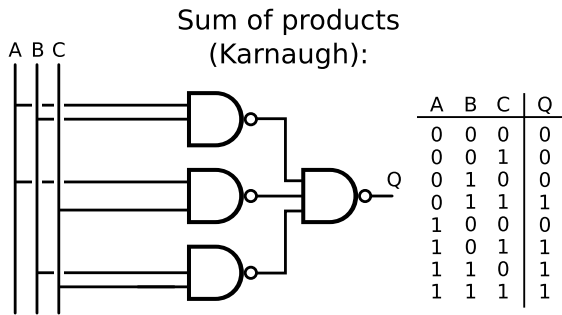


Figure 31: Circuit of Fig. 29 optimized using a Karnaugh map and the de Morgan's laws.

So that the original expression

$$Q = \overline{A} \cdot B \cdot C + A \cdot \overline{B} \cdot C + A \cdot B \cdot \overline{C} + A \cdot B \cdot C$$

is reduced to

$$Q = A \cdot B + A \cdot C + B \cdot C,$$

5 gates instead of 11. That can further be reduced using De Morgan's laws. The final circuit is shown in Figure 31 and has only three 2-input NAND-gates (4 transistors each) and a 3-input NAND-gate (6 transistors), giving a total of 18 transistors. A tremendous reduction.

The recipe for designing *any* logic function is the following:

1. Draw the truth table on basis of the description of it
2. Convert the truth table to the form of a Karnaugh map
3. Join as many 1s together (in a SoP approach) or 0s together (in a PoS approach). Find the simplified Boolean expression by at least using every 1 (SoP) or 0 (PoS) once
4. Implement the Boolean expression found above by a layer of AND-gates followed by a layer of OR-gates (SoP) or by a layer of OR-gates followed by a layer of AND-gates (PoS)
5. Convert them all to NAND-gates (SoP) or NOR-gates (PoS).

Table XXIII: Transient circuit behavior

Steady state	The steady-state of a circuit is the value of the output after the inputs have been stable for a long time
Transient behavior	The transient behavior of a circuit is the sequence of output values after the inputs change until the final steady state
Glitch	A glitch is a short-time value of the output that is not the (desired) steady state
Hazard	A hazard is the possibility of a glitch

4.4 Timing; transient behavior. Glitches and hazards

So far we have not talked yet about the timing of circuits. We only looked at the logic. "For inputs like this, the output will be like that", etc. This is what we call the steady-state behavior. (See Table XXIII for a description of some timing concepts). We have seen how we can find any logic function with sum-of-products or product-of-sums method. Moreover, we have seen how we can use the Huntington postulates to simplify the circuit and made use of Karnaugh maps to do this rapidly.

But how does a circuit get to this steady state? Can things go wrong? Ideally, we would like the output to change once to the final value after a certain amount of time (and ideally this should be as short as possible). However, in the transient behavior there can be so-called glitches, temporary values that are not the initial nor the final value. Imagine we change the inputs of a circuit that is in a steady-state 0 in such a way that the new steady-state logic value will also be 0, but temporarily it will be 1. This is a 'glitch'. It might be not bothering us, but it might also mess up the signal processing somewhere down the line, and we need to address this problem. When a glitch has the *possibility* to occur it is called a 'hazard'.

A static-1 hazard is defined by two input combinations X and Y such that

- X and Y differ by only one input variable
- X and Y both produce a logic 1 at the output in steady-state
- Momentarily a 0 may appear on the output.

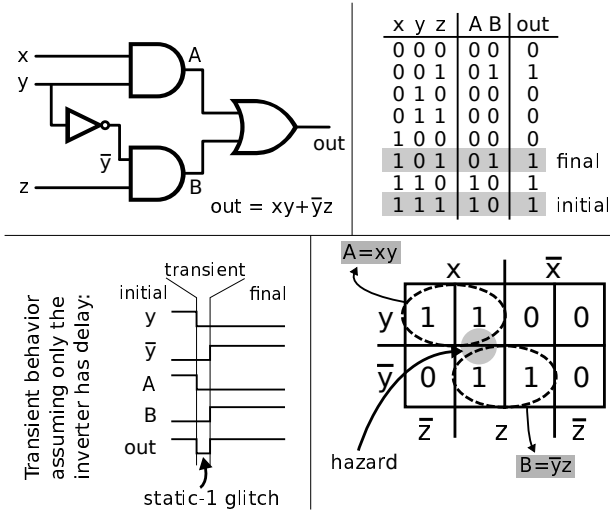


Figure 32: A circuit with a static-1 hazard. A hazard exists when two AND gates in the first layer are touching, but not overlapping in the Karnaugh map.

In other words, a static-1 hazard is a possibility of a 0 glitch. Likewise, a static-0 hazard is defined by two input combinations X and Y such that

- X and Y differ by only one input variable
- X and Y both produce a logic 0 at the output in steady-state
- Momentarily a 1 may appear on the output.

In other words, a static-0 hazard is a possibility of a 1 glitch.

These hazards exist because the AND-gates in a sum-of-products or the OR-gates in a product-of-sums have their result ready at their outputs at different times. Maybe because they are not completely identical, or because these components at the first layer of the circuitry have additional inverters at their inputs. A classic example of a hazard is the circuit shown in Figure 32, it implements the logic function $F = xy + \bar{y}z$ through a sum of products. Note the inverter at the bottom AND-gate. In this circuit, AND-gate A receives x and y, and AND gate B receives z and \bar{y} . This last \bar{y} is produced by an inverter, and this inverter is made of two transistor that, apart from inverting the signal, also causes a small delay. That means that \bar{y} is not at all times the inverse of y. There can be transitory moments in which they are the same, and that causes the glitch. Imagine we are in a steady-state with $x = y = z = 1$, and thus $out = 1$, see the truth table in Figure 32. When the input y changes from 1 to 0, both A and B switch; A goes from

Table XXIV: Transient truth table with all inputs at the AND gate. In this table $y=\overline{y}=0$ can occur causing a static-1 0-glitch.

x	y	z	\overline{y}	A	B	out	state
				xy	$\overline{y}z$	A+B	
0	0	0	0	0	0	0	
0	0	0	1	0	0	0	
0	0	1	0	0	0	0	
0	0	1	1	0	1	1	
0	1	0	0	0	0	0	
0	1	0	1	0	0	0	
0	1	1	0	0	0	0	
0	1	1	1	0	1	1	
1	0	0	0	0	0	0	
1	0	0	1	0	0	0	
1	0	1	0	0	0	0	0-glitch
1	0	1	1	0	1	1	final
1	1	0	0	1	0	1	
1	1	0	1	1	0	1	
1	1	1	0	1	0	1	initial
1	1	1	1	1	1	1	

1 to 0 and B switches from 0 to 1 (see the initial and final states indicated in the truth table). However, AND-gate A switches slightly earlier than AND-gate B, and temporarily there is a situation in which both AND-gates output 0, and the final output will temporarily be 0, before it settles again at the final steady-state 1. In effect, what happens is this sequence if we assume all other gates (A, B and out) are instantaneous:

state	x	y	\overline{y}	z	A	B	out	
					xy	$\overline{y}z$	A+B	
initial	1	1	0	1	1	0	1	
intermediate	1	0	0	1	0	0	0	glitch
final	1	0	1	1	0	1	1	

In effect, *physically* we have the transient truth table of Table XXIV, where illogical but physically possible transitory states ($y=\overline{y}$) are also included.

This transient behavior with a static-1 0-glitch is shown in the picture and is caused by the situation where both $y=0$ and $\overline{y}=0$. When we look at the Karnaugh map, we see that it occurs when we get a transition from two distinct loops in the Karnaugh map (two different AND-gates producing a 1). One loop feeding an AND-gate is implementing $A=xy$ and another loop is implementing $B=\overline{y}z$. The glitch occurs when we go from one loop

Table XXV: By adding another loop in the Karnaugh map, adding another AND-gate at the first layer of the SoP circuit, the 0-glitch can be eliminated. Even if $y=\bar{y}=0$ occurs, this causes no static-1 0-glitch.

x	y	z	\bar{y}	A xy	B $\bar{y}z$	C xz	out A+B+C	state
0	0	0	0	0	0	0	0	
0	0	0	1	0	0	0	0	
0	0	1	0	0	0	0	0	
0	0	1	1	0	1	0	1	
0	1	0	0	0	0	0	0	
0	1	0	1	0	0	0	0	
0	1	1	0	0	0	0	0	
0	1	1	1	0	1	0	1	
1	0	0	0	0	0	0	0	
1	0	0	1	0	0	0	0	
1	0	1	0	0	0	1	1	no glitch!
1	0	1	1	0	1	1	1	final
1	1	0	0	1	0	0	1	
1	1	0	1	1	0	0	1	
1	1	1	0	1	0	1	1	initial
1	1	1	1	1	1	1	1	

to another by switching from $y=1$ to $y=0$, implying going from $AB=10$ to $AB=01$. A glitch occurs when two or more AND-gates switch *and* a single of those AND-gates switching would temporarily cause an output 0.

To avoid this from happening, we add redundancy with a third product loop (AND-gate $C=xz$) that connects the two loops, $F = xy + \bar{y}z + xz$. As can be seen from the physical, transient, truth table of Table XXV, the glitch has been eliminated and the resulting circuit (Figure 33) is hazard free.

Note that there are no static-0 1-glitches in sum-of-products circuits. That it started at zero, winds up in 0, but temporarily has a 1 as output. That is because each AND-gate can only change once. So, when a specific AND-gate changes from 0 to 1 output (and thus possibly changing the final output in the sum to 1), the AND-gate can never switch back and then the output can never go back to 0, so it was not a static-0 state after all. The reader can easily verify this in the physical truth table above.

In a similar way we can address problems of static-0 1-glitches in product-of-sums implementations.

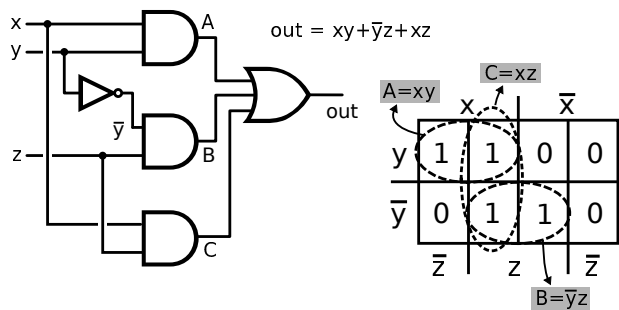


Figure 33: The circuit of Figure 32 made hazard-free by adding another product loop ($C=xz$) in the sum. In the Karnaugh map is visible as overlapping the loops that caused a hazard by a new loop.

4.5 Latches, flip-flops and memory

By adding feedback to the circuits we can turn them into memory elements. That is, the output state no longer only depends on the input signals, but also on the history of the state. An example is the simple circuit called a latch that consists of two NOR-gates with the output of each serving as one of the inputs of the other, see Figure 34. When the input combination is $SR=10$, the output is always $Q=1$, independent of what Q was before, and when the input is $SR=01$, the output is always $Q=0$.

S/R-latch			
S	R	Q	
0	0	Q	keep
0	1	0	reset
1	0	1	set
1	1	N/A	not allowed

This thus 'writes' the state of the latch and for this reason we have conveniently labeled the input terminals (instead of A and B) S and R, because a 1 on these terminals sets to 1 and resets to 0 the output Q, respectively. However, when the input combination is $SR=00$, the output depends on what the state was before; if it was $Q=0$, it stays $Q=0$, and if it was $Q=1$, it stays $Q=1$, as the reader can easily verify: The state $SRQ\overline{Q} = 0010$ is stable, as well as the state $SRQ\overline{Q} = 0001$. In all cases is the output of the sibling NOR-gate the opposite. Note also that the input combination $SR = 11$ is not allowed (N/A), since it results in an invalid state ($Q=\overline{Q}=0$).

We call this circuit a set/reset (S/R) latch. It is a memory element that can store one bit of information. By the combination $SR=01$ we can write a 0 in this memory and by the combination $SR=10$ we can write a 1. In the case of $SR=00$ we keep the memory (and only 'read' the output Q).

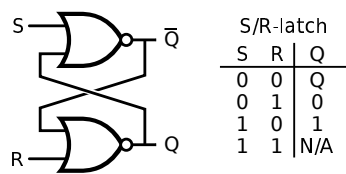


Figure 34: A set/reset (S/R) latch composed of two NOR-gates with feedback.

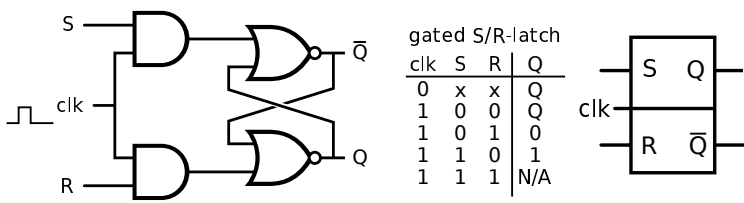


Figure 35: A gated S/R-latch made of a S/R-latch by adding AND ports at the entrance. This way, the state of the latch can only change when the clock is high. "N/A" means not allowed.

In most cases, we want synchronized components. That means that states can only change synchronously with a clock pulse. Following the same reasoning we can build a gated S/R-latch AND'ing the set/reset signals with the clock before they are fed into the NOR-gates, see Figure 35. This way, the state of the latch can only change when the clock pulse (*clk*) is high, otherwise the state will keep its actual state *Q*, since both signals fed to the NOR-pair are 0.

Flip-flops are very similar to latches, but they rather respond to *transitions* of states (especially the clock) rather than the states themselves. For engineers this is much better, because it is much more clear *when* the actions take place. The way this is done is by making use of the delay every electronic component causes. When we AND a clock signal with its inverted (and thus delayed) self, a very narrow pulse results, see Figure 36. If the width is very small, we can represent this by saying that the circuit responds to the *edge* of the clock pulse, also called 'edge-triggered'. This is indicated by an arrow in the bottom figure. Similarly to this rising-edge-triggered circuit we can make a circuit for falling-edge-triggered responses.

In this book no distinction will be made from now on; all latches described in this work are in fact implemented as edge-triggered flip-flops. A clock signal enters a circuit through such a pulse-generating gate as shown in Figure 36. We use synchronous components, and only come back to asynchronous circuits in the last chapter.

The problem with the S/R-latches and flip-flops is that they have a non-

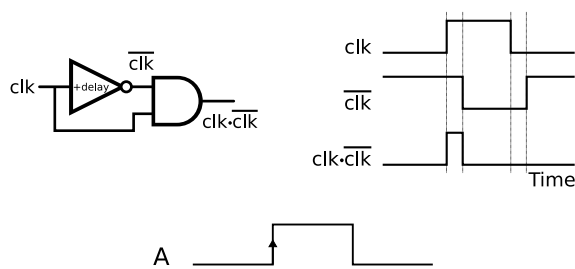


Figure 36: A short pulse version of the clock signal can be made by AND’ing it with its own inverted signal. Because the inverter also introduces a short delay, the resulting output is a pulse with a width equal to the delay. If the width is very small, we can represent this by saying that the circuit responds to the edge of the clock pulse, we call this edge-triggered. This is indicated by an arrow in the bottom drawing.

allowed input combination $SR=11$ that can put the latch or flip-flop into a non-determined state. An engineer’s nightmare! To avoid this problem so-called J/K flip-flops were invented (by Jack Kilby). It contains another feed-back loop to the clock-gate. This circuit works the same as a S/R flip-flop, but it has no problem with a $JK=11$ combination. This input combination actually inverts the output: $Q \rightarrow \overline{Q}$. Figure 37 shows the electronics of a J/K edge-triggered flip-flop and its simplified representation that we will use in this book. The truth table of the J/K latch (without the clk) is given by:

J/K latch		
J	K	Q
0	0	Q keep
0	1	0 reset
1	0	1 set
1	1	\overline{Q} invert

From the J/K flip-flop we can make two interesting derivatives. The first one is made by linking the J and K inputs together and calling it T. This stands for ‘toggle’, as it toggles the output $Q \rightarrow \overline{Q}$ upon receiving a clock pulse with a 1 at the T input. The other device is with the K input of the J/K flip-flop the inverse of the J input. In this case, the truth table shows us that the output is equal to the input, now called D for ‘data’ (or delay), when a clock pulse is received. See Figure 38. We will use the symbols for these flip-flops from now on.

When a bit of information is placed on the D-line, this will be copied into the memory when a clock pulse (clk) is received (and be ready on the output Q). We can already recognize here a classical computer memory element. In fact, so-called registers of computers work in this way by being a set of D

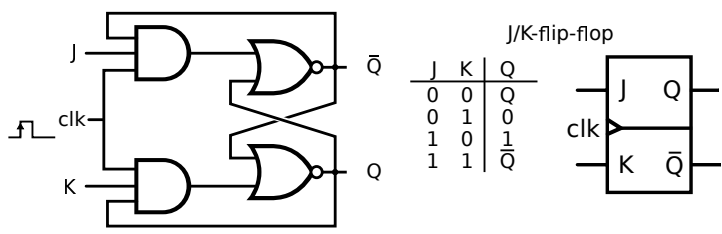


Figure 37: A J/K edge-triggered flip-flop, a basic ingredient of digital systems. The (positive-flank) edge-effect is indicated by an arrow in the clock pulse and by a triangle at the clk input in the final symbol of the component.

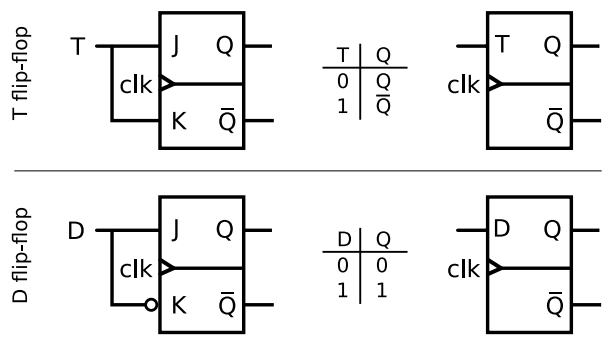
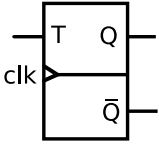
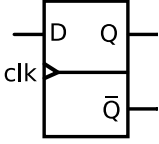


Figure 38: T(oggle) and D(ata) flip-flops.

type flip-flops. The simple truth tables of toggle (T) data (D) flip-flops are:

T flip-flop			
clk	T	Q	
0	0	Q	
0	1	Q	
1	0	Q	
1	1	\bar{Q}	

D flip-flop			
clk	D	Q	
0	0	Q	
0	1	Q	
1	0	0	
1	1	1	

Finally, Figure 39 shows a so-called master-slave flip-flop. It consists of

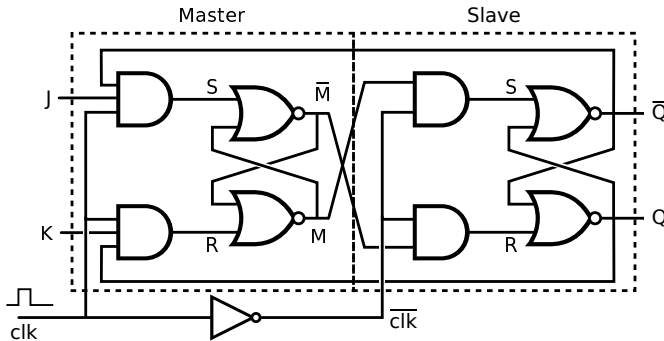


Figure 39: A master-slave J/K flip-flop. If the clock is high, the J/K signals reach the gated master S/R latch and the master (M) signal is set accordingly. These are fed to the gated slave S/R latch, but do not arrive there because the slave S/R latch is disabled by the inverted clock. When the clock is changed from 1 to 0 this slave latch now processes the signals coming from the (now) stable master latch and the output (Q) is set accordingly. The feedback from final output (Q, \bar{Q}) to input give it a J/K behavior (with $J=K=1$ allowed). The M/S flip-flop eliminates race conditions.

two gated S/R latches, fed back in a J/K configuration. When the clock is rising, the master S/R latch is enabled and its signal set according to the J/K and Q signals. It is, as it were, transparent. The second S/R latch, however is disabled by the inverted clock. at the falling edge of the clock the first S/R latch is stable at its output value and no longer responds to J/K input changes. The slave S/R latch is now transparent and lets through the output of the master S/R latch. So, the state as it were ripples through the M/S S/R latch. At the rising edge it passes the first stage and at the falling edge it passes through the second stage. The advantage of this over a normal J/K flip-flop is that so-called race conditions cannot occur. This can occurs when $J=K=1$ and the output will toggle, $Q \rightarrow \bar{Q}$. But if the active time is long enough it can toggle back to the original state. It can actually oscillate at high frequency, something that is normally unwanted. To prevent this, a master/slave set/reset edge-triggered flip-flop can be used.

The important question for what is coming next is, on basis of the truth tables of the flip-flops or latches, not how the output states change when signals are applied to it at the input, but rather the reverse: if we want to change the output from one state to another, how do we do that? What excitation signals do we need to supply at the inputs to achieve the desired output states or changes thereof? As an example, if we have an S/R flip-flop and its (output) state is 0, how do we change it to the state 1. Looking at the S/R truth table, we can easily see that we have to supply a $SR = 10$

combination. Keeping the state at 0 can be achieved in two ways: either keep the 0 state by $SR = 00$, or write a 'new' 0 into the S/R memory latch by $SR = 01$. In other words, we have to supply $SR = 0x$ to go from state 0 to state 0 (written as $0 \rightarrow 0$). The truth table for S/R latches and flip-flops is thus

old		new Q	Action $Q \rightarrow Q$
Q	S R		
0	0 0 (keep)	0	$0 \rightarrow 0$
0	0 1 (reset)	0	$0 \rightarrow 0$
0	1 0 (set)	1	$0 \rightarrow 1$
0	1 1 (N/A)		
1	0 0 (keep)	1	$1 \rightarrow 1$
1	0 1 (reset)	0	$1 \rightarrow 0$
1	1 0 (set)	1	$1 \rightarrow 1$
1	1 1 (N/A)		

which can be summarized by the following 'excitation table':

Action	Excitation
$Q \rightarrow Q$	S R
$0 \rightarrow 0$	0 x
$0 \rightarrow 1$	1 0
$1 \rightarrow 0$	0 1
$1 \rightarrow 1$	x 0

(where x means 'don't care'). The excitation tables for all types of flip-flops and latches used in this book are given in Table XXVI.

4.6 Finite-state machines (FSM)

The next step in slowly assembling a computer architecture is joining these memory elements just described to logic circuits described earlier. This results in a so-called finite-state machine, see Figure 40. It consists of two parts, a logic array and memory elements. The former have outputs that only depend on its inputs. The latter stores the state of the machine. These states are then also used as input in the logic array (without this closing of the circle, the saving of the state would be rather useless, as useless as write-only memory). Whereas all states have to be used as input, not all inputs of the logic array are necessarily coming from the memory elements. Some can come from other sources (maybe a keyboard, or an Internet connection). Moreover, not all states are necessarily also output states, and not all output states are necessarily also saved in memory. Although, it is obvious that at

Table XXVI: Excitation tables for various types flip-flops. 'x' means don't care.

Action	Excitation			
Q	S R	J K	T	D
$0 \rightarrow 0$	0 x	0 x	0	0
$0 \rightarrow 1$	1 0	1 x	1	1
$1 \rightarrow 0$	0 1	x 1	1	0
$1 \rightarrow 1$	x 0	x 0	0	1

least one output has to come out of the logic array; a computer that is not communicating its computed result is also rather useless.

In summary, we can imagine the following finite-state machine (FSM):

- A FSM with a inputs, b outputs and 0 memory elements is simply a logic array, composed of a logic circuit that can be designed with fundamental gates using the techniques described before, including Karnaugh maps, etc. We can call it a Boole machine since it follows simple Boolean logic.
- A FSM with a inputs, 0 outputs and m memory elements does not make sense. We can call it here a Berkeley machine. (Berkeley was the philosopher that said that the chair he left in the next room does no longer exist because it was not observed).
- A FSM with 0 inputs, b outputs and m memory elements is a sequencer; every time a clock pulse is received, new output and memory states are calculated on basis of the previous state of the machine. This is called a Moore machine.
- A general FSM with a inputs, b outputs and m memory elements is called a Mealy machine.

4.6.1 Moore machine sequencers

Let's take a look at some examples. The Boole machine (logic only) has already been dealt with in the previous sections, so we start here with a Moore machine sequencer. Imagine we want to make the sequence 010101010... , inverting the output at every rising clock pulse. How do we do that? With some thinking, the reader is probably capable of coming up with this very simple solution:

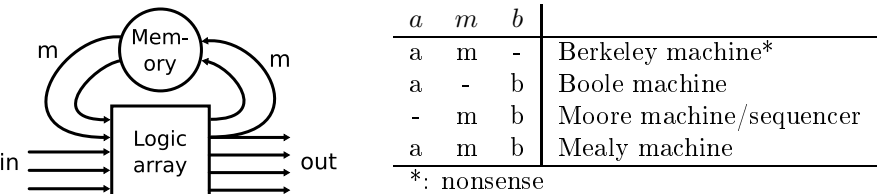
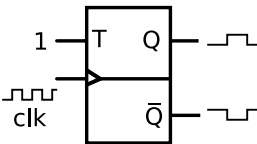


Figure 40: A finite-state machine (FSM) consists of a logic array (with b outputs only depending on its a and m inputs) and m memory elements that save the state of the machine.



That was too easy! Let’s try something more difficult. What if we want to make a three-bit counter, $000 \rightarrow 001 \rightarrow 010 \rightarrow 011 \rightarrow 100 \rightarrow 101 \rightarrow 110 \rightarrow 111 \rightarrow 000 \dots$. We first recognize that the output and the state are equal, so $m = b$ in the finite-state machine picture. We need three memory elements. Let’s call them FF2, FF1 and FF0, with outputs Q2, Q1 and Q0, each saving a bit of the output (with FF0/Q0 the LSB). The first step is now to make a list of possible states and at each state the next desired state (a transition table), and thus a ‘what to do’ (an action table). In this case it is very simple:

Transition table						Action table		
Current state			Next state			What to do?/Action		
Q2	Q1	Q0	Q2	Q1	Q0	Q2	Q1	Q0
0	0	0	0	0	1	$0 \rightarrow 0$	$0 \rightarrow 0$	$0 \rightarrow 1$
0	0	1	0	1	0	$0 \rightarrow 0$	$0 \rightarrow 1$	$1 \rightarrow 0$
0	1	0	0	1	1	$0 \rightarrow 0$	$1 \rightarrow 1$	$0 \rightarrow 1$
0	1	1	1	0	0	$0 \rightarrow 1$	$1 \rightarrow 0$	$1 \rightarrow 0$
1	0	0	1	0	1	$1 \rightarrow 1$	$0 \rightarrow 0$	$0 \rightarrow 1$
1	0	1	1	1	0	$1 \rightarrow 1$	$0 \rightarrow 1$	$1 \rightarrow 0$
1	1	0	1	1	1	$1 \rightarrow 1$	$1 \rightarrow 1$	$0 \rightarrow 1$
1	1	1	0	0	0	$1 \rightarrow 0$	$1 \rightarrow 0$	$1 \rightarrow 0$

We now need to select our hardware. From experience, for example the simple 1-bit sequencer above, we decide for T-type flip-flops. The truth table of this flip-flop we have to translate into an excitation table – an ‘how-to-do’ – because we have to find out how to change the output bits. So, we get the

following excitation table based on the T-type flip-flop truth table of page 95, which was also part of Table XXVI:

Action Q	Excitation T
$0 \rightarrow 0$	0
$0 \rightarrow 1$	1
$1 \rightarrow 0$	1
$1 \rightarrow 1$	0

(The excitation tables for all flip-flops are shown in Table XXVI).

On basis of this we design for each output bit T-type flip-flop the logic circuit to supply the correct input signal based on the current state (Q2,Q1,Q0). In other words, for each flip-flop we design a logic circuit with three inputs – Q2, Q1 and Q0 – and one output. As an example, for Q0 we have

Current state			Action (What to do?)	Excitation (How to do it?)
Q2	Q1	Q0	Q0	T0
0	0	0	$0 \rightarrow 1$	1
0	0	1	$1 \rightarrow 0$	1
0	1	0	$0 \rightarrow 1$	1
0	1	1	$1 \rightarrow 0$	1
1	0	0	$0 \rightarrow 1$	1
1	0	1	$1 \rightarrow 0$	1
1	1	0	$0 \rightarrow 1$	1
1	1	1	$1 \rightarrow 0$	1

Let’s do this for all flip-flops:

Current state			Action & Excitation					
Q2	Q1	Q0	Q2	T2	Q1	T1	Q0	T0
0	0	0	$0 \rightarrow 0$	0	$0 \rightarrow 0$	0	$0 \rightarrow 1$	1
0	0	1	$0 \rightarrow 0$	0	$0 \rightarrow 1$	1	$1 \rightarrow 0$	1
0	1	0	$0 \rightarrow 0$	0	$1 \rightarrow 1$	0	$0 \rightarrow 1$	1
0	1	1	$0 \rightarrow 1$	1	$1 \rightarrow 0$	1	$1 \rightarrow 0$	1
1	0	0	$1 \rightarrow 1$	0	$0 \rightarrow 0$	0	$0 \rightarrow 1$	1
1	0	1	$1 \rightarrow 1$	0	$0 \rightarrow 1$	1	$1 \rightarrow 0$	1
1	1	0	$1 \rightarrow 1$	0	$1 \rightarrow 1$	0	$0 \rightarrow 1$	1
1	1	1	$1 \rightarrow 0$	1	$1 \rightarrow 0$	1	$1 \rightarrow 0$	1

We can find the circuits for these three excitation signals T0, T1 and T2, using Karnaugh maps:

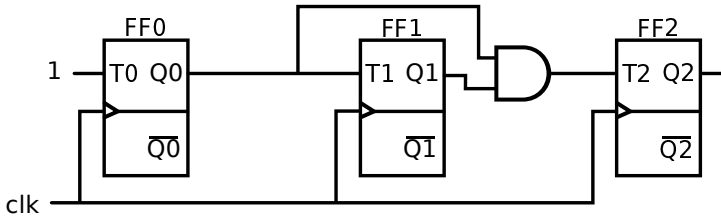


Figure 41: A three-bit synchronous counter as an example of a Moore machine sequencer. The 3-bit output sequence is $Q_2, Q_1, Q_0 = 000 \rightarrow 001 \rightarrow 010 \rightarrow 011 \rightarrow 100 \rightarrow 101 \rightarrow 110 \rightarrow 111 \rightarrow 000 \dots$

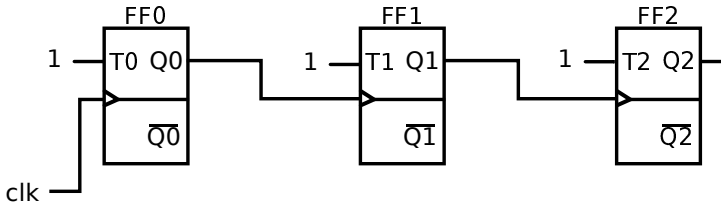
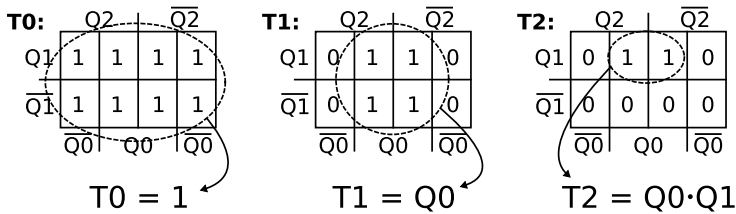


Figure 42: The asynchronous equivalent of the counter of Figure 41. Here the flip-flops have different clock signals.



So we can design the circuits:

- T0 is connected to logic 1
- T1 is connected to the output Q_0 of flip-flop FF0
- T2 receives a signal from the outputs of Q_0 and Q_1 AND'ed.

This results in the final circuit shown in Figure 41. Note in this circuit that all flip-flops receive the same clock signal. We call this a synchronous circuit. The asynchronous equivalent, with the flip-flops have different clock signals, is given in Figure 42.

exercise: 3-bit counters

Design 3-bit counters (counting cyclic 0 to 7) that use D-type flip-flops and J/K-type flip-flops.

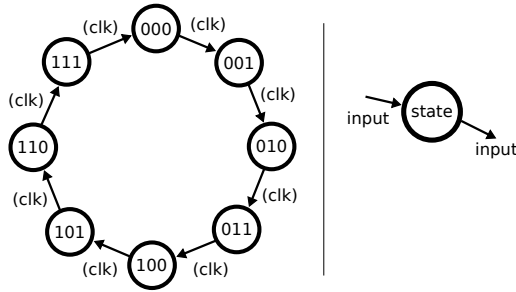
The recipe for designing a finite state Moore machine is the following:

1. Draw the transition table with columns for 'current state' and 'next state'
2. Add columns for the actions to be taken at each state to go to the next state
3. Chose the hardware, the type of flip-flop to use. One flip-flop for every bit of status. Draw the flip-flops as the basis of the circuit
4. Separately write down the truth table of this type of flip-flop and translate this into an excitation table for that hardware, what pulses to give at the input terminals to cause the desired states at the output terminals
5. With the above hardware excitation table, translate the actions to be taken into excitations to be supplied at each 'current state'
6. With the help of Karnaugh maps find the Boolean expressions of the above excitations as functions of variables of the 'current state'
7. Add the SoP or PoS logic gates found from the expressions above to the circuit.

4.6.2 Mealy machines

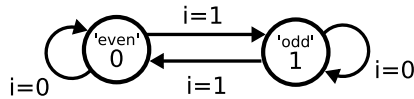
As written before, Mealy machines differ from Moore machines in that they have, apart from the state and output, also input lines. In this case we start by designing a state diagram. It consists of first writing down all possible states, identifying how many there are and then attribute memory elements and values to them, and their associated outputs (if different from the memory state). We then write these in circles and connect them by arrows – called 'edges' – that show the input a that is needed to bring the machine from one state to another. Since Moore machines is a subset of Mealy machines, we can also do this for the former, with the 'input' merely the clock signal. Take as an example our 3-bit counter. A Mealy machine

state diagram would be:



The clock signal is assumed implied and will not be shown anymore. From now on, all circuits are synchronous and take clock signals.

More interesting are Mealy machines that do take regular input. Take for example a circuit that checks the parity of an input string of bits, i . Parity is the number of 1s in a string, specifying whether it is odd or even. Analyzing this, we see that it has two possible states: 'odd' and 'even'; the number of 1s until now. Two possibilities, so a single bit is sufficient to store the state. We decide that 'even' = 0 and 'odd' = 1. We also decide to take this as our output bit Q and will use a single flip-flop FF. So, we have $a = 1$ input bits, $b = 1$ output bits and $m = 1$ states (equal to the output, $m = b$). Moreover, if the number of bits until now is odd ($Q=1$), and we receive an $i=1$, we have to change the parity state to $Q=0$, etc. Let's first make the state diagram:



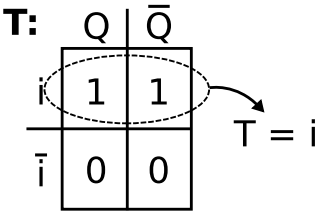
Note the state in the circles and the input values i at the arrows ('edges'). Now we can make an action table (with output equal to state):

Current state	Input	Next state	Action
Q	i	Q	$Q \rightarrow Q$
0	0	0	$0 \rightarrow 0$
0	1	1	$0 \rightarrow 1$
1	0	1	$1 \rightarrow 1$
1	1	0	$1 \rightarrow 0$

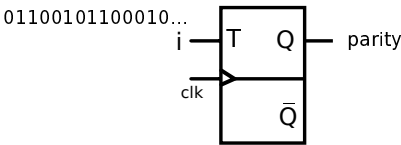
This looks very toggly, so we decide for a T-type flip-flop, and we translate the action table into an excitation table:

State & Input		Action	Excitation
Q	i	$Q \rightarrow Q$	T
0	0	$0 \rightarrow 0$	0
0	1	$0 \rightarrow 1$	1
1	0	$1 \rightarrow 1$	0
1	1	$1 \rightarrow 0$	1

The Karnaugh map:



And we see that simply the input of the T-type flip-flop is directly connected to the input signal:



exercise: Parity checker

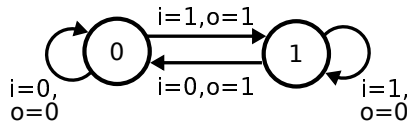
Exercise: Design a parity checker that uses D-type flip-flops or J/K-type flip-flops.

Let’s take a look at another example. Slightly more complicated, as we will see, because the output is not the same as the state and we thus need separate hardware for it. We are talking here about an edge detector. It will output a 0 if the incoming bit is the same as the previous bit and a 1 if it is different.

in:0011000111010000111...
out:0010100100111000100...

At the first transition from 0 to 1, it will output a 1, but then with the next 1, no change took place and the output is 0. Etc.

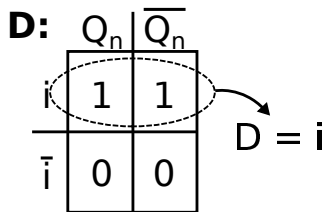
First we have to make a state diagram. Looking at the description, we recognize we have to store one bit of information, namely the value of the previous bit in the string. This bit of information can have two states (0 and 1), so we need one flip-flop to store it. We then proceed in determining how each state is linked to other states by what input i. The resulting state diagram is given by,



Note that the output is not equal to the final state (nor of initial state) and we thus need separate hardware circuit for it. Looking at the state itself, it is clear that the state is actually equal to the input. That looks very data'ish, so we select a D-type flip-flop for our state. And we can then make an action table (pro forma, because we already now that it will be equal to i , but let's just do all steps of this tango):

Current state	Input	Next state	Action	Excitation
Q_n	i	Q_{n+1}	$Q_n \rightarrow Q_{n+1}$	D
0	0	0	$0 \rightarrow 0$	0
0	1	1	$0 \rightarrow 1$	1
1	0	0	$1 \rightarrow 0$	0
1	1	1	$1 \rightarrow 1$	1

Note that we have distinguished here explicitly the previous or current state (Q_n) from the next state (Q_{n+1}) for reasons that will be clear in a moment. The Karnaugh map for the logic array of D depending on Q_n and i is given by,



And we see that indeed simply the input of the D-type flip-flop is directly connected to the input signal i . For the output we need a different circuit. By looking again at the state diagram, we find that the simple truth table (not by an excitation analysis, since no state changes are involved) is given by,

Current state	Input	Output
Q_n	i	o
0	0	0
0	1	1
1	0	1
1	1	0


```
    q = i;                // save state
}
```

We already get here our first hint of the link between programming and hardware.

4.7 Exercises

exercise: Shakespeare

0: Answer Shakespeare's famous question: "To be or not to be, that is the question". Show that the answer is "true". Answer at end.

exercise: Priority line

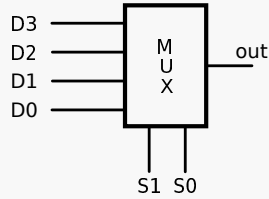
1: A circuit has 4 inputs, $I_3 \dots I_0$. The 2-bit output (O_1, O_0) of the circuit is the binary representation of the highest input with a 1. For example, $O_1, O_0 = 10$ indicates that I_2 is the highest input with a 1, so $I_3 = 0$, $I_2 = 1$, $I_1 = x$, $I_0 = x$ (x is 'don't care'). Implement the circuit with only NAND-gates.

exercise: Gray-code converter

2: Design a circuit which converts a number of 3 bits
a) from binary to Gray code
b) from Gray code to binary.
Use only XOR-gates.

exercise: Multiplexer logic

3: A multiplexer ('MUX', see also p. 120) copies the data at an input line D_i , selected by the 'address' lines S , to the output. A MUX with n select lines can be used to realize any logical function with $n + 1$ input variables. For example, a MUX with two selector lines, S_1 and S_0 (see below), can implement any three-input logic. Design a circuit based on a multiplexer that implements the logic function $F(a, b, c) = \Sigma(2, 3, 5, 6)$.



Hint: connect a to $S1$, and b to $S0$. Write the truth table.

exercise: 2-bit binary counter

- 4: Design a 2-bit binary counter, $00 \rightarrow 01 \rightarrow 10 \rightarrow 11 \rightarrow 00 \dots$,
 a) with SR flip-flops
 b) with JK flip-flops.

exercise: Synchronous 3-bit Gray-code counter

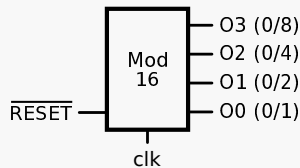
- 5: Design a synchronous 3-bit counter in Gray code. Use flip-flops of your choice.

exercise: Modulo-10 binary counter

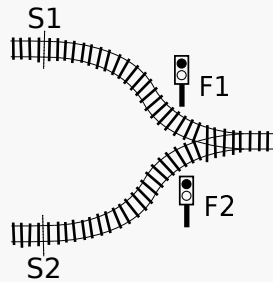
- 6: Design a modulo-10 binary counter ($0 \dots 9$), $0000 \rightarrow 0001 \rightarrow \dots 1001 \rightarrow 0000 \dots$

exercise: Clock

- 7: Imagine a chip with a modulo-16 counter ($0 \dots 15$; 4 bits output) and an active-low $\overline{\text{RESET}}$.



On basis of such chips, design an (asynchronous) clock with (24) hours, (60) minutes, and (60) seconds.

exercise: Railroad junction

8: At a junction, two railroad tracks join into one. Hundreds of meters before the junction there are two sensors, S1 and S2 which detect the passing of a train. Just before the junction are two signal F1 and F2 (0=red, 1=green). Only one train can pass; after one has passed the other has to wait (forever).

Design the state diagram, make a table and simplify it. Design the control circuit based on D-type flip-flops.

exercise: Mask

9: Design a finite-state-machine circuit that masks every even bit in an input stream to 0.

Example:

in: 10111010111100100...

out: 10101010101000100...

exercise: Odd bits

10: Design a finite-state-machine circuit that determines if in a bit stream every odd bit is a 1. The output must be 1 as long as every odd bit is 1 and 0 (forever) once an odd bit equal to 0 is encountered.

Example:

in: 10111010111100100...

out: 11111111111100000...

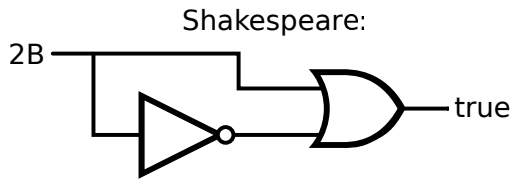
exercise: Elevator

11: A building has three floors and an elevator. Each elevator door has a button to call the elevator (C0, C1, C2). Assume that only one button may be pressed at any time. The outputs are the motor state M (on/off) and the motor direction D (up/down). Analyze

the problem and design the state diagram, and on basis of that the circuit.

exercise: Lock

12: A door has an electronic lock with a decimal keyboard. The code 3285 opens the door. The control circuit returns to its initial state after opening the door or when the introduced code starts being wrong. Analyze the problem and draw the state diagram.



5 | Integration

It is now time to integrate the digital systems components into a computer. We will do this here step by step so that one can see what is going on. In the previous chapter operations consisted of single-bit output functions. In this chapter we will see how we can join them to form multi-bit hardware. Then build an architecture around it.

As we are going to do arithmetic calculations it is beforehand important to make the observation that the calculations are *independent of the number system*. A mathematical property of a number is true in any number system. If a number is odd or even in decimal, it is odd or even in any number system. A prime number is one that cannot be divided by any integer number without leaving a remainder. This is true in any number system. Two plus three equals five in any number system. In decimal it is simply $2+3=5$. In binary it is $10+11=101$. Thus, to do calculations, we can choose our number system hardware in which we perform the calculations and then at the end represent the result in decimal (if it is intended for human reading). Since digital electronics is binary electronics, the obvious number system is binary. So, we will now start building a computer based on binary components. We do that by integration of already known components, like AND-gates, OR-gates, NOR-gates, NAND-gates, XOR-gates, flip-flops and latches, etc. Then we will integrate these integrated structures. Every step of integration will create emergent properties. We start with building a binary arithmetic circuit from logic gates, the half-adder.

5.1 Half-adder/full-adder

The digital gates in the previous chapter were all *logical* gates. That means they follow Boolean algebra, 'logic'. We are often more interested in numerical calculations, as in additions, subtractions, division and multiplications. In this chapter we will learn how that is done, starting with additions. In fact, calculations are also expressible in boolean logic. That is because they

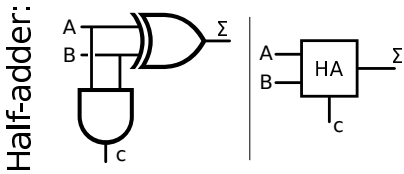


Figure 44: A half-adder calculating the sum (Σ) and carry (c) of two binary 1-bit numbers (A and B). Right: equivalent symbol.

are functions with output (bits) depending on the input (bits), and that is something we know how to deal with.

The simplest addition is one of two binary 1-bit numbers, $A + B = C$. There are four possibilities:

$$\begin{aligned} 0+0 &= 0 \\ 0+1 &= 1 \\ 1+0 &= 1 \\ 1+1 &= 0 \text{ (plus an 'overflow'; carry)} \end{aligned}$$

This overflow we call a carry. Note that these zeros and ones here now represent 1-bit binary *numbers* and not voltages or true/false Boolean values (bits). They are numbers implemented by Boolean logic, implemented in electronic hardware. So, now we have two output functions of two input bits. If we represent this in hardware and assign – what seems utterly reasonable – the number 0 to a logic-0 bit and the number 1 to a logic-1 bit we get two logical circuits implementing two logic functions. We call these Σ for the sum bit, and c for the carry bit. The truth table for these two logical functions looks like this:

Half-adder			
A	B	Σ	c
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

It is directly clear that the sum bit can be produced by an XOR-gate and the carry bit c can be generated by an AND-gate, so we wind up with the circuit for this so-called 'half-adder' (HA) as shown in Figure 44.

For curiosity's sake: We had left dangling the ternary-logic half-adder problem in Chapter 3, stating that it cannot be done with ternary-logic gates (only). The solution is normally to decode the ternary digits (trits) to binary and do the logic in binary and then at the end encode the result back to ternary. This approach is presented in Figure 45. That just for curiosity's

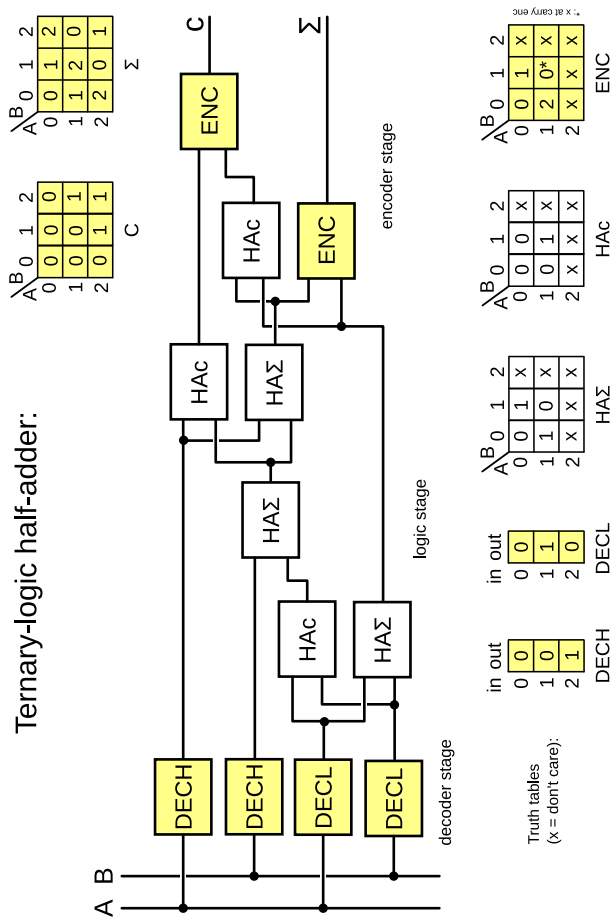


Figure 45: For curiosity’s sake the common solution to a ternary half-adder. The logic is implemented with (pseudo)binary logic gates (white boxes; note the don’t-cares in the truth tables) and conversions to and from binary (shaded boxes).

sake. The world is using binary logic, so let’s get back to it. The rest of the book is only about binary hardware.

When we join two half-adders together to form a full-adder, we can add two binary 1-bit numbers and a carry-in (cin), calculating the sum of the three and a carry-out (cout). $\Sigma = (A + B + c_{in}) \bmod 2$, and $c_{out} = (A + B + c) > 1 ? 1 : 0$. The former expression only looks at the least-significant bit and the latter is an expression “if $(A + B + c) > 1$ then $c_{out} = 1$ else $c_{out} = 0$ ”, which can also be represented by the integer division $(A + B + c)/2$. In a

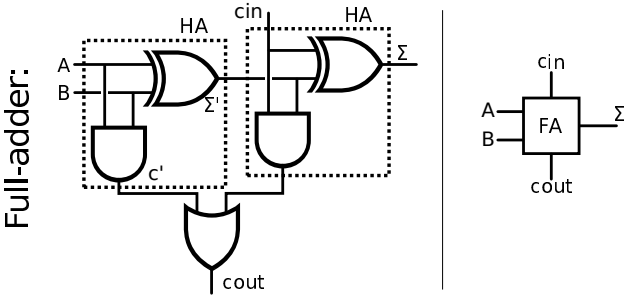


Figure 46: A full-adder calculating the sum (Σ) of two binary 1-bit numbers and a carry-in (cin) based on two half-adders (HA) of Figure 44. Right: equivalent symbol.

Boolean-logic truth table:

Full-adder (FA)				
A	B	cin	Σ	cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Figure 46 shows the circuit of such a full-adder (FA), composed of two half-adders and an OR-gate, that is the corner stone of every modern computer.

5.2 Summing and subtracting

Now that we have a full-adder that can add two binary 1-bit numbers (with carry), we can easily make an n -bit adder by daisy-chaining 1-bit adders, as shown in Figure 47. The carry-in of bit n comes from the carry-out of bit $n - 1$. The first carry-in is set to 0 (for additions). The calculation 'ripples' through the daisy chain and after a while also the most significant bit ($\Sigma 3$) and its carry (cout) are ready. This is why this type of hardware adding two numbers is called a ripple-carry adder.

We see here the first processor component; the Intel 4004 used 4-bit architecture and did all arithmetic with such adders. As we will see later on (with the Russian-peasant algorithm), it can also do multiplications and divisions.

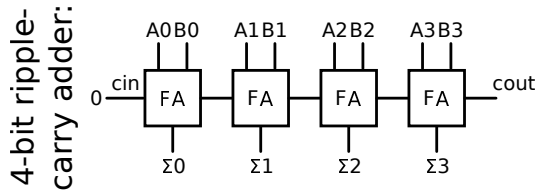


Figure 47: A ripple-carry adder circuit able to add two 4-bit numbers made from a daisy chain of 1-bit full-adders.

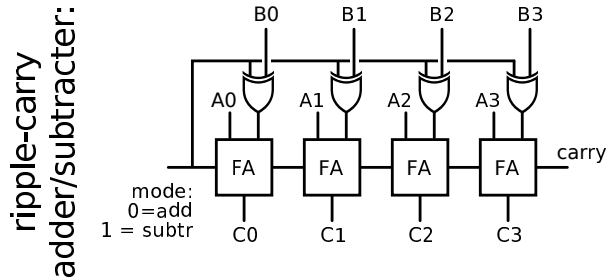


Figure 48: A 4-bit ripple-carry adder/subtractor based on a 4-bit ripple-carry adder, making use of the two's-complement sign-change of numbers: inverting all bits and adding 1 (here carry-in).

When we slightly adapt the 4-bit adder we can make it perform subtraction too. Remember that, in two's-complement, negative numbers are found by inverting all the bits and adding 1. We also observed then that subtracting is equal to adding the two's-complement of the number. Figure 48 shows how this is implemented in hardware. Every bit of the 4-bit operand B is XOR-ed with 1, which is equivalent to inverting it (1s'-complement). This operand \overline{B} is then added to operand A using the 4-bit adder from Fig. 47 that now also gets a carry-in equal to 1. So if the mode bit is set (1), the output is

$$C = A + \overline{B} + 1, \text{ thus } C = A - B.$$

If the mode bit at the carry-in input on the other hand is 0, all XOR-gates receive this 0 and simply copy the bits of operand B to the output unaltered. So if the mode bit is cleared (0), the output is

$$C = A + B + 0, \text{ thus } C = A + B.$$

An adder can be turned into a subtracter by setting the carry-in to 1 and inverting all bits of the B-operand by XOR'ing them with 1.

5.3 Advanced adding and subtracting

On basis of a 1-bit full-adder we managed to construct a ripple-carry adder that can add and subtract any size binary numbers by placing the adders in series. The carry-out of bit n becomes the carry-in of stage $n + 1$. The problem with this is that every step in a ripple-carry adder delays the output. Calculation in the next stage of a ripple-carry adder can only start when the previous level has finished. Or better to say, the output of stage n only becomes reliable, some time after stage $n - 1$ had become reliable,, because a reliable carry-out of the previous stage is needed to produce a stable, reliable output. We can get an estimate of the speed on basis of the number of transistor layers of the fundamental gates (see Table XXII). AND-gates and OR-gates have 3 layers, an XOR gate 5. Therefore, analyzing the circuit in Figure 46, the sum output (Σ) takes 10 layers (longest path: {A,B}-XOR-XOR- Σ), while the carry-out (cout) takes 11 layers (longest path: {A,B}-XOR-AND-OR-cout). If a single transistor takes 1 ps to switch, at worst a full-adder takes 10 ps to calculate the sum and 11 ps to complete the calculation of the carry. A 32-bit ripple-carry addition will take 32 times 11 ps (0.352 ns) to fully complete and that may be too long, even if the underlying transistors are fast. Instead of taking 1-bit full-adders as building blocks, we may take 2-bit full-adders, or generally n -bit full-adders. The complexity of the logic circuit increases rapidly (imagine a 17-input 9-output Karnaugh map of an 8-bit full-adder), but a 4-bit instant adder still seems feasible. An example of such a 4-bit carry-look-ahead adder circuit is shown in Figure 49. As you can see, the hardware is more complex than a 4-bit ripple-carry adder, but speed is gained; only 4 levels of gates are needed, meaning the output is stable much faster. The numbers of layers to carry-out (cout) is only 7 ({A,B}-{NOR,NAND}-AND-NOR-cout) compared to 44 in a ripple-carry architecture. (Assuming n -input gates switch as fast as 2-input gates). These 4-bit carry-look-ahead adders can then also be placed in a ripple-configuration, so that a 32-bit adder can be made of eight 4-bit-carry-look-ahead adders, with a total of $8 \times 7 = 56$ layers, a significant reduction from the original 352 layers of the ripple-carry-adder.

Once again, subtracting numbers is done by adding the two's-complement of the number, similar to the technique used in the ripple-carry adder; inverting all bits by XOR gates and adding 1 by asserting the first carry-in line, see Fig. 48.

5.4 Advanced logic circuits

Figure 50 shows a 4-bit 1-bit-right/left shifter. The input D is shifted right or left one bit depending on the mode bit. Imagine this mode bit is $R=1$. The leftmost AND-gate receives this 1 and bit D3, and its output is thus

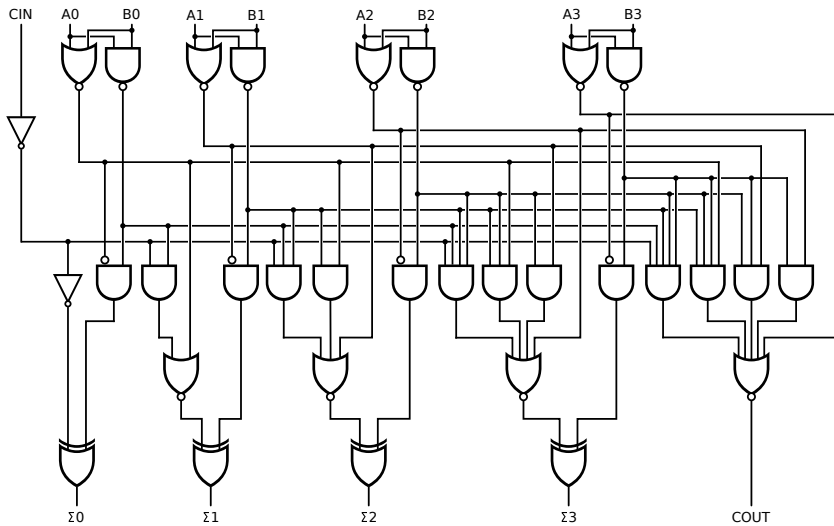


Figure 49: A 4-bit carry-look-ahead adder, for instance the 74LS83A integrated circuit.

equal to D3. This is fed into OR-gate of Z2. The other input from this Z2 OR-gate is 0 because it comes from the fourth AND-gate that receives a 0 from L and this AND-gate thus outputs 0, independent of what D1 might be. OR-gate Z2 is thus equal to $(D3 \text{ OR } 0)$ and that is D3. Doing the same analysis for the other bits, one can see that

$$\begin{aligned} \text{if } R=1 \text{ (L=0): } & Z3, Z2, Z1, Z0 = 0, D3, D2, D1 \\ \text{if } R=0 \text{ (L=1): } & Z3, Z2, Z1, Z0 = D2, D1, D0, 0 \end{aligned}$$

which is functionally a 1-bit R/L-shifter. For future reference, we observe here that a left-shift is equivalent to adding a 0 to the end of our number, and as we have seen in the chapter on number systems, this is equivalent to a multiplication by the base number. In our case here a left-shift is a multiplication of the number by a factor of 2. Likewise, a right-shift is a division of the number by 2. Where the adder is the basic component for addition and subtraction, the shifter is the basic ingredient for multiplication and division. Together they can do all integer arithmetic.

Since shifting one bit is equal to multiplication by 2 (left shift) or division by 2 (right shift), we can imagine how this shifter can be used in multiplications. Something that will be done later on. However, if we really want to use it for that purpose, for *arithmetic* shifts, it would be nice if the *sign* of the number is maintained in the operation. For that purpose we have the arithmetic shifter of Figure 51. We see that if the mode A (for arithmetic) is asserted, the MSB (D3) is maintained in the output Z3. That way, for a

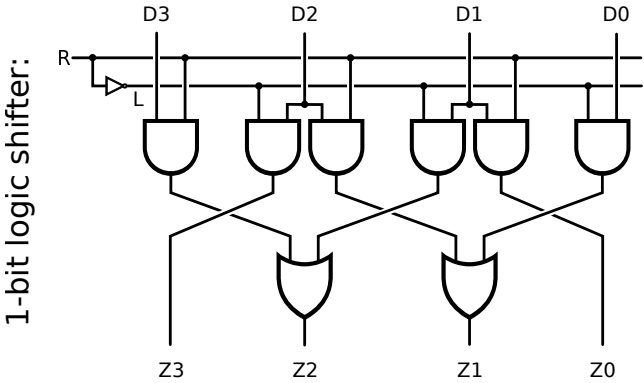


Figure 50: A 4-bit logic R/L-shifter. The input D is shifted right or left one bit depending on the mode bit (R). The opening bit is filled with a 0

right-shift the number -6 (2's-complement: 1010) becomes the number -3 (2's complement: 1101).

if $A=1$ (arithmetic R-shift): $Z3, Z2, Z1, Z0 = D3, D3, D2, D1$
if $A=0$ (logic R-shift): $Z3, Z2, Z1, Z0 = 0, D3, D2, D1$

The example is shown in the figure. It has to be noted that this only makes sense for right-shifts, since arithmetic left-shifts are done correct in logic mode anyway or cause overflow/underflow anyway. Also note that it does not work for -1 , which stays -1 forever upon such arithmetic right shifts.

Other important circuits are multiplexers and demultiplexers. A multiplexer (MUX) selects one of 2^n input lines by n control lines and places it on the output (for instance a bus). This is done in a SOP solution by 2^n AND-gates that each receives one of the input lines and a unique combination of direct/negated control lines. Only one AND-gate receives the control lines in the form of only 1s and this one thus lets through the associated input signal; all others output 0 because at least one of the control lines arrive as a 0 at the gate. Finally, an OR-gate sums it all up. $n - 1$ 0s and the selected data. The truth table below and Figure 52 clarify this.

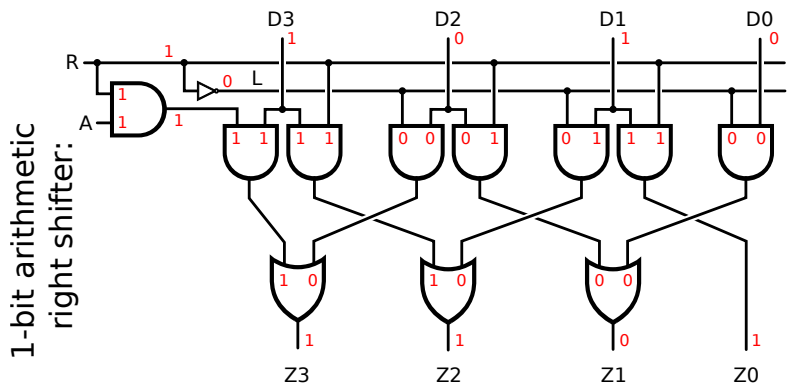


Figure 51: A 4-bit arithmetic right-shifter. The input D is shifted right or left one bit depending on the mode bit (R). In this case, when the mode A is set, it will do an arithmetic shift, which maintains the MSB. An example is shown how -6 (2's-complement: 1010) rightshifted into -3 (2's complement: 1101) is done. If A were not asserted, the leftmost AND-gate would produce a 0 and the Z3 bit a 0, making it a logic right-shift.

Multiplexer (MUX, Fig. 52)							
S1	S0	D3	D2	D1	D0	out	min-term
0	0	x	x	x	0	0	
0	0	x	x	x	1	1	$\overline{S1S0D0}$
0	1	x	x	0	x	0	
0	1	x	x	1	x	1	$\overline{S1S0D1}$
1	0	x	0	x	x	0	
1	0	x	1	x	x	1	$S1\overline{S0D2}$
1	1	0	x	x	x	0	
1	1	1	x	x	x	1	$S1S0D3$

x = don't care

exercise: Multiplexer

A multiplexer with 4 data lines and 2 selector lines can be used to implement any logic function of three variables A, B, C. Find a solution for the function $f(A, B, C) = \sum(2, 3, 5, 6)$.

Answer:
The truth table of this function is given below. If we connect A to S1 and B to S0 then we can connect the four possibilities to the data lines D3 ... D0. In the truth table below horizontal lines are placed

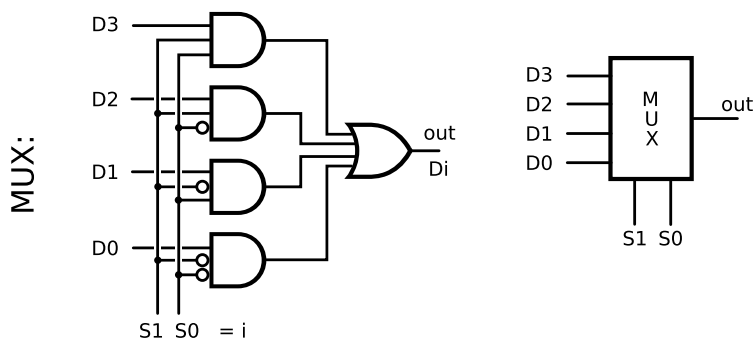
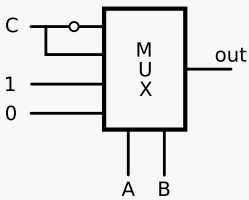


Figure 52: A 4-line multiplexer. One of the 4 data lines D3, D2, D1 and D0, appear on the output. The choice is determined by the control lines S1 and S0. These control lines appear on the AND-gates on unique direct/negated combination. In this way only one AND-gate receives two 1s from the control lines. This AND-gate becomes transparent for the data line. All other AND-gates output 0. The OR-gate sums all 0s and the selected data, the latter thus appearing on the output. Note also that, like any other SOP solution, this can be converted into a NAND-only circuit.

between inputs with combinations of A, B. The data at the selected port is then either 0 (if out = 0, independent of C), C (if out is equal to C), \overline{C} (if out is equal to \overline{C}) or 1 (if out = 1, independent of C).

A	B	C	out	
S1	S0			
0	0	0	0	D0=0
0	0	1	0	
0	1	0	1	D1=1
0	1	1	1	
1	0	0	0	D2=C
1	0	1	1	
1	1	0	1	D0= \overline{C}
1	1	1	0	



In this case, D0 is equal to \overline{C} , D1 is equal to C, D2 is equal to C and D3 is equal to \overline{C} .

Similarly, a demultiplexer (deMUX) puts the data on the selected output line with all other output lines equal to 0. See Figure 53. The truth table is given by

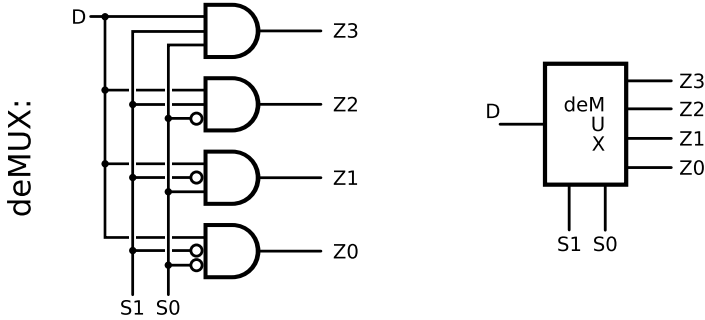


Figure 53: A 4-line demultiplexer. The data D appears on one of the 4 output lines $Z3$, $Z2$, $Z1$ or $Z0$; the other lines are 0. The choice is determined by the control lines $S1$ and $S0$. These control lines appear on the AND-gates on unique direct and negated form. In this way only one AND-gate receives two 1s from the control lines. This AND-gate becomes transparent for the data line. All other AND-gates output 0.

De-multiplexer (deMUX, Fig. 53)					
S1	S0	Z3	Z2	Z1	Z0
0	0	0	0	0	D
0	1	0	0	D	0
1	0	0	D	0	0
1	1	D	0	0	0

Very similar to a deMUX is a decoder (Figure 54). A decoder is a deMUX with the data equal to 1, and thus not needed to feed to the AND-gates. That means, that a decoder has one output line equal to 1 and all other equal to 0. This is used in things like selecting which of the memory chips can place data on the communication bus, all others remaining in tri-state. Only a chip that receives such a chip-select signal can talk.

Decoder (Fig. 54)					
S1	S0	Z3	Z2	Z1	Z0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

An encoder is a circuit that outputs in binary the information about which of the input lines is 1. So, if D_n is 1, it outputs n in binary format. The encoder normally assumes that exactly one input line is asserted (1). The hardware can then be simplified very much (see Figure 55). As a side effect, if there are two input lines asserted, it outputs the highest number. In a truth table

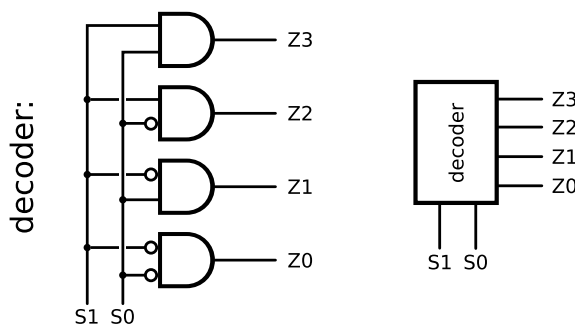


Figure 54: A decoder is a demultiplexer (Fig. 53) with the data line equal to 1, and thus not needed to supply to the AND-gates. This way one output line, determined by the selector lines S , is 1 and all the others are 0. This can be used, for instance as a chip-select signal, $S1$ and $S0$ function as 'address' of the chip to be turned active.

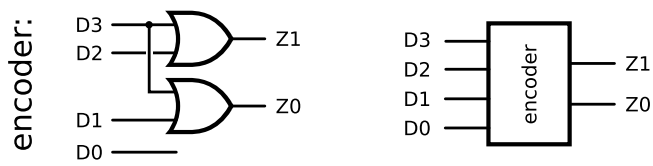


Figure 55: An encoder determines which of the input lines is 0 and returns the binary-coded number of that line, so if $D2$ has a 1, $Z1, Z0$ will be 1,0. In case of more than one 1, the highest counts. At least one input line should be 1.

Encoder (Fig. 55)					
D3	D2	D1	D0	Z1	Z0
0	0	0	x	0	0
0	0	1	x	0	1
0	1	0	x	1	0
1	x	x	x	1	1

x = don't care

Note also that the simplification is so much that $D0$ is not connected to anything, since in the truth table it only appears as a don't-care (x). The reader is urged to use the techniques of the previous chapter (esp. Karnaugh maps), more specifically, using a product of sums to check that the hardware of Fig. 55 indeed implements the truth table above.

Finally, a comparator compares two binary numbers A and B , and outputs if $A=0$, $A<B$ or $A>B$. The determination of all bits being equal ($A=B$) can be performed by a NOT-XOR-gate that outputs 1 if equal and 0 if dif-

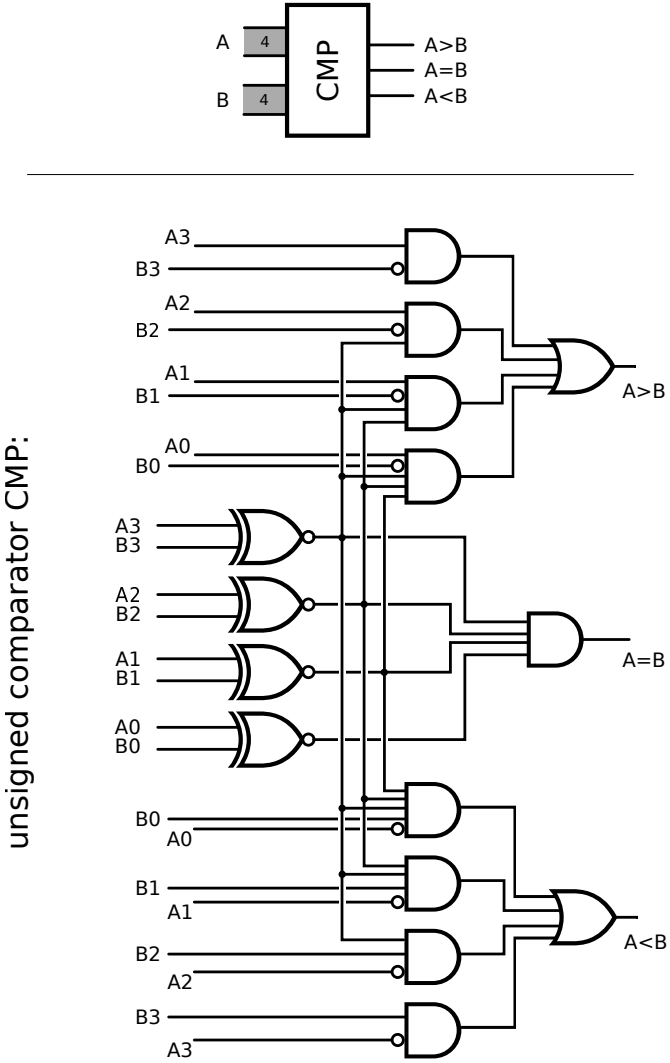


Figure 56: A comparator CMP compares two n -bit binary numbers and outputs 1 if they are equal and 0 if they are different (shown here for $n=4$). It also compares and determines if $A>B$ and $A<B$. Of course exactly one output is equal to 1.

ferent. AND'ing all these individual conditions results in the desired output, see the middle part of the circuit in Fig. 56 for a 4-bit version of it.

The other type of comparisons are a little more complex. For a 2-bit unsigned comparison (4-bit input) we can still easily do the SOP approach.

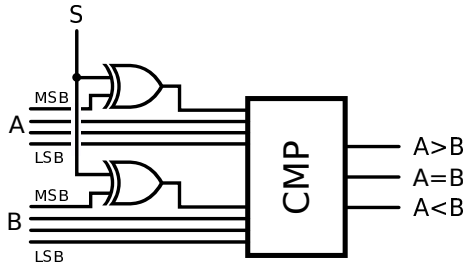


Figure 57: A comparator for signed and unsigned integers. When the S(igned) line is asserted, the MSBs are inverted before being fed into the logic comparator.

This was shown in the exercise on page 64 and the solution is $(A < B) = \overline{A}B1 + A\overline{0}B1B0 + A1A0B0$. A full unsigned 4-bit solution is shown in Figure 56. Take a look at the bottom part of the circuit in the Figure ($A < B$). It is OR'ing four conditions coming from AND-gates. The bottom one, involving the most-significant bits (A_3 and B_3), is simple: if the A_3 is 0 and B_3 is 1, then B is obviously larger than A and the output $A < B$ is true (1). The condition above it is also easy to understand. One input is an output from the $A = B$ section, namely comparing the MSBs A_3 and B_3 . This input is 1 if $A_3 = B_3$, so either $A_3 = B_3 = 1$ or $A_3 = B_3 = 0$. In this case, to find out if $A < B$, we now must compare the second bits, A_2 and B_2 . This is then done in the same way as A_3 and B_3 was done before, so this AND-gate produces a 1 if $A_3 = B_3$ and $B_2 = 1$ and $A_2 = 0$. OR'ing it in the final OR-gate produces a 1 at the output. The same strategy is then used for the other bits (1 and 0). With for bit 1 information coming from the NOT-XOR-gates equal to $A_3 = B_3$ and $A_2 = B_2$ and for the LSB (bit 0) equal to $A_3 = B_3$ and $A_2 = B_2$ and $A_1 = B_1$.

The condition $A > B$ can be derived from the conditions $A = B$ and $A < B$ both false, as is sometimes done in literature. Alternatively, a solution shown in the Figure, a circuit similar to $A < B$ can be constructed (just exchanging the roles of B and A). The former solution is obviously simpler but slower, containing less transistors, but more layers of them.

For signed comparisons the comparator at first thought would seem to be inadequate. However, remember that a 2's-complement signed integer of N bits has a negative weight for the first (most significant) bit only. The rest has the 'normal' (positive) weight:

$$-2^{N-1} + 2^{N-2} + \dots + 2^0.$$

Look at the table below, where unsigned integers and 2's-complement signed integers are ordered in value from smallest to biggest. From this table it is obvious that for 2's-complement signed integers we just have to

invert the MSB and can then use the unsigned coparator! For this we can use the same 'trick' as the one we used to make the adder into a subtractor, namely using XOR gates. In this case XOR'ing the MSB of A and B with the signed assertion line.

unsigned		signed		$\overline{\text{msb}}$
dec	bin	dec	bin	bin
0	0000	-8	1000	0000
1	0001	-7	1001	0001
2	0010	-6	1010	0010
3	0011	-5	1011	0011
4	0100	-4	1100	0100
5	0101	-3	1101	0101
6	0110	-2	1110	0110
7	0111	-1	1111	0111
8	1000	0	0000	1000
9	1001	+1	0001	1001
10	1010	+2	0010	1010
11	1011	+3	0011	1011
12	1100	+4	0100	1100
13	1101	+5	0101	1101
14	1110	+6	0110	1110
15	1111	+7	0111	1111

After inversion of the MSB, we can treat the numbers with our unsigned comparator. For instance, $-6 = 1010$, which becomes 0010, and $+3 = 0011$, which becomes 1011 and is thus bigger. The inversion of the MSB is done by feeding it together with a signal line (S) for signed comparison into an XOR gate, see Figure 57.



We now have all the components necessary to build a computer (Table XXVII): Memory. Logic functions (AND, OR, XOR). Arithmetic functions (ADD, SUB). Shifting functions (left and right). And comparisons (CMP: $A < B$, $A = B$, $A > B$). We can start combining them and make them do some serious things. We will see that by combining them a new property emerges, namely programming. We can make it execute a determined number of specific operations, called a program. This is where computing starts.

Table XXVII: Basic components of computers: Arithmetic and logic, control logic, and memory components.

Data flow control	Data processing Arithmetic & logic		Data storage Memory
MUX	adder/subtractor	AND, NAND	flip-flop
deMUX	L/R-shifter	OR, NOR	RAM*
coder	comparator	XOR	
decoder		NOT	

*: to be discussed in the chapter on memory (Ch. 7)

6 | Computers

In the previous chapters we have seen how we can start with simple electronic components and make fundamental logic gates out of them. Then, by combining them we made circuits that implemented any desired logic function. We learned special techniques to be able to help us in designed these circuits. We also saw how we got an emergent property at every step. From electronics to digital electronics (gates) we found Boolean logic. By adding feedback to the system, we found memory effects as an emergent property. In the next chapter we continued the integration and found more advanced components, such as the multiplexer and decoder. We did not learn what purpose they serve, but that will be made clear now.

We reached a point in which we can actually make the advanced processing units that are at the core of every computer. We can build a computer that can actually calculate things in a flexible way. It can be 'programmed'. All the components needed were explained in the preceding text. We just need to combine them.

6.1 Arithmetic and logic unit (ALU)

Let us start with designing a unit that does the actual calculation, all the arithmetic and logic word-size operations. This is the arithmetic and logic unit, or ALU for short. An ALU can perform a set of operations, the choice of which one is being executed is determined by a code supplied to it. Figure 58 shows an example of a 1-bit ALU that can perform four different operations on operands A and B: A AND B, A OR B, NOT A and A+B. All four operations are always performed, but one is copied to the output Z by the MUX. For instance, for the operation code F1,F0 = 1,1 the result of the operation A AND B appears on the output. For the arithmetic operation A+B, also a carry-in is taken into account and a carry-out is generated.

With this 1-bit ALU we can use the same technique as was used for the full-adder (Fig. 48), namely build an n -bit ALU by daisy-chaining n 1-bit ALUs. Figure 59 shows an example of such a technique in a 4-bit ALU.

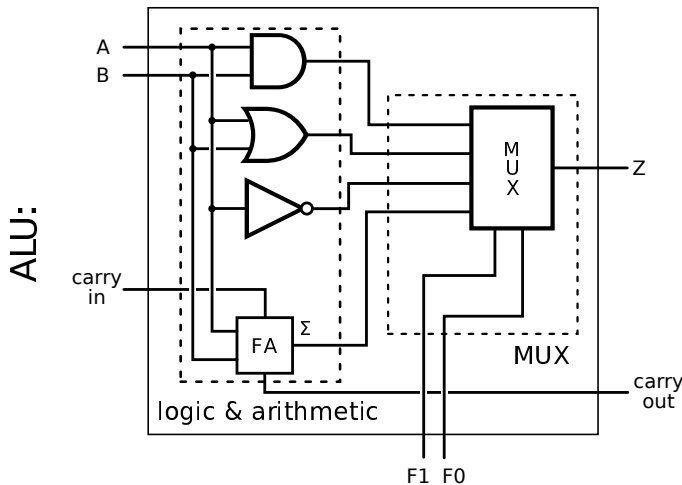


Figure 58: A rudimentary 1-bit arithmetic and logic unit (ALU). Depending on the 2-bit opcode F1,F0 one of the calculated products (A AND B, A OR B, NOT A, or A+B) appear on the output.

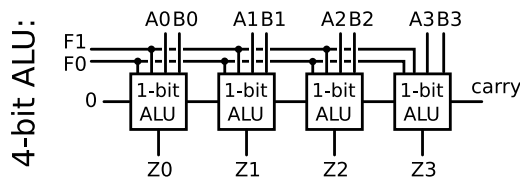


Figure 59: A 4-bit ALU made by daisy-chaining four 1-bit ALUs of Fig. 58.

Of course, real ALUs are much more complicated. They normally have a much larger set of instructions (and thus more operation code lines F). Moreover, not all instructions take the same amount of time of producing a result and it may be possible that some take more than one clock cycle to come up with the result at the output. However, we take this rudimentary ALU as a prototype to explain how a computer architecture can be build from known blocks.

The arithmetic function $A+B$ is effectively a ripple-carry adder and is therefore slow. ALU_i can only start calculating when the carry of ALU_{i-1} has settled. More advanced ALUs use flash techniques (for instance the carry-look-ahead adder of Fig. 49). Designing circuits with for instance $8/\text{input}$ truth tables, 8×8 Karnaugh maps, etc. They can calculate arithmetic sums much faster, at the cost of a design with a lot of transistors.

6.2 Central processing unit (CPU)

We are now going to join everything together in an integrated chip (IC) called the central processing unit (CPU), that is except the external (main) memory, which will be discussed in a separate chapter (Ch. 7). We start with the ALU. First we observe that it has two input data elements – so called operands – that it can process in a cycle. In some cases we will only need one (like for instance the 1-operand instruction NOT A), but in most cases operations consists of processing two input operands. That means that all high-level computation has to be broken down to such dyadic (two-input) operations. If we, for instance, want to calculate the 4-operand expression $y = ax^2 + bx + c$, we have to break it down into bi-operand operations. (Note: a smart way to do it, requiring only one register z would be $a \times x \rightarrow z$, $z + b \rightarrow z$, $z \times x \rightarrow z$, $z + c \rightarrow z$). The ALU is a two-input-operand-one-output-operand machine.

Thus we need a place to store the operands. A place where we can retrieve the input operands and store the output operand. While this can be external memory (some architectures indeed only have external memory), a better (faster) solution is to have memory elements close-by. This for the very simple reason that the computer speed is limited by the speed of light $c = 3 \times 10^8$ m/s. That means that if we are going to store and retrieve information at a distance of, say, 15 cm, communication takes about 1 ns, and our processor limited to 1 GHz. That is why it is advantageous to have the memory elements as close to the ALU as possible. Such memory elements we call registers. These are flip-flops or gated latches, D-type flip-flops, in the vicinity of the ALU, joined in the CPU IC. Main memory will be described later, rests to say that there also exists an intermediate type of memory – ‘cache’ – that is not directly next to the ALU, behaves as normal main memory. and is still inside the CPU.

Processing (running a program) consists of supplying the sequence of operation codes F and operands A and B to the ALU, and storing the results Z the ALU produces somewhere, either in the registers or in memory. In some architectures the ALU comes with an ‘accumulator’, an register where the last result of the ALU is stored. Apart from the data registers, the CPU also needs a place to store the actual operation that is performed (instruction register IR) and the program counter (PC) that points to the location in main memory of the instruction to be copied to the instruction register.

Most architectures then have these registers inside the CPU:

- Data registers. A place to store the two input operands A and B and the resulting value Z.
- An instruction register IR specifying what operation to perform, where

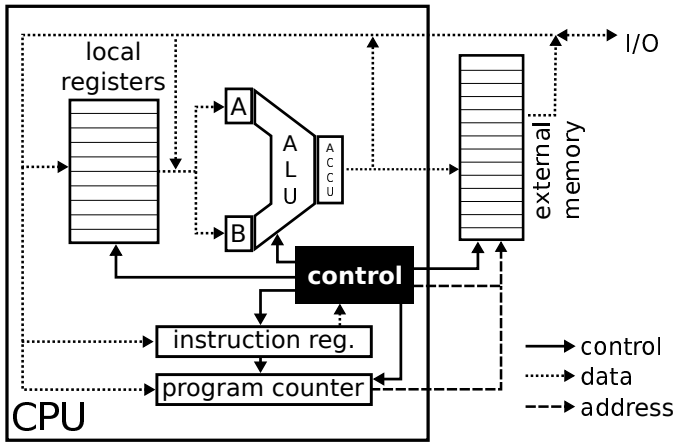


Figure 60: Schematic view of a typical CPU in a Von Neumann architecture. The architecture thus has four basic components: An ALU performing the arithmetic and logic operations, control logic, external memory, and I/O.

the operands can be found, and where to place the result. Similar to our 2-bit instruction $\{F1, F0\}$ of our 1-bit ALU.

- A program counter PC specifying where in memory the next instruction can be found.
- Flags inside a status register describing the state of the machine. For instance if the last result was negative or not or if a carry occurred.

This combination of control logic, program counter, instruction register, ALU and registers (possibly with cache) is called the central processing unit (CPU). A typical CPU is shown in Figure 60. It is also sometimes called a datapath. Technically speaking, a CPU is a datapath plus the controlling hardware.

The CPU (ALU plus control logic), plus memory, plus I/O defines the architecture. In the Von Neumann architecture, shown in Fig. 60, the memory is separated from the CPU and is external. Communication between CPU and memory takes place over a data bus, with the place (address) of the information in memory communicated either over the same data bus or by a separate address bus (shown).

6.3 Control logic

The controlling hardware is like a traffic controller that basically runs the program by sending strobing pulses to the individual components at the

correct times. Running a step in a program consists of

- Fetching the instruction from main memory. This means placing the program counter latch out of tri-state so that it sets the next-instruction's address on the address bus. The controller then enables the external memory chip which contains the address that can now write data (that is, the instruction code) on the data bus (dashed lines) and the instruction register receives a 'strobing' (gate) signal pulse. The instruction register now contains the operation code ('opcode') and a specification where to find the operands.
- The controller decodes the instruction and strobes the operands into the registers A and B of the ALU that then processes them.
- The controller selects (by a multiplexer) the result of the desired operation and places it at the output of the ALU. The ALU gets write permission (is placed out of tri-state) and places the data on the (internal) data bus.
- The controller strobes the result in the desired local register or places it on the external data bus to write it in external memory.
- The program counter is advanced to make it point to the next instruction in memory and a new cycle begins.
- The output can also be piped into the program counter, in which case the program does not continue in the next address but jumps to somewhere else in memory instead.
- Some instructions can be conditional, as in "if ($A < B$) then jump to address". Of course, coded in hardware. (If voltage at output of the comparison logic array is hi (COMP=1), then latch operand into PC).

Figure 61 shows an imaginary example of a very simple architecture, a 1-bit ALU with a 4-operations instruction set, with four 1-bit registers and main memory containing 8-bit instructions from which only the instructions can be read (contains no data, and no writing is possible; just for simplicity's sake).

The control-logic hardware is in this case simple; it converts the symmetric (50% duty cycle) clock into a sequence of short pulses by the invert-and-delay technique discussed in Fig. 36 on page 94: IR, AB, REGS, PC. At the IR pulse 8-bits of information in main memory are latched into the gated latch of the instruction register (IR), with the address of the information supplied to the 8 multiplexers by the program counter (PC). At the AB pulse, the two operand information bits of the two registers through multiplexers selected by the {A1,A0} and {B1,B0} bits of the instruction of the instruction register are latched to the ALU registers A and B. The

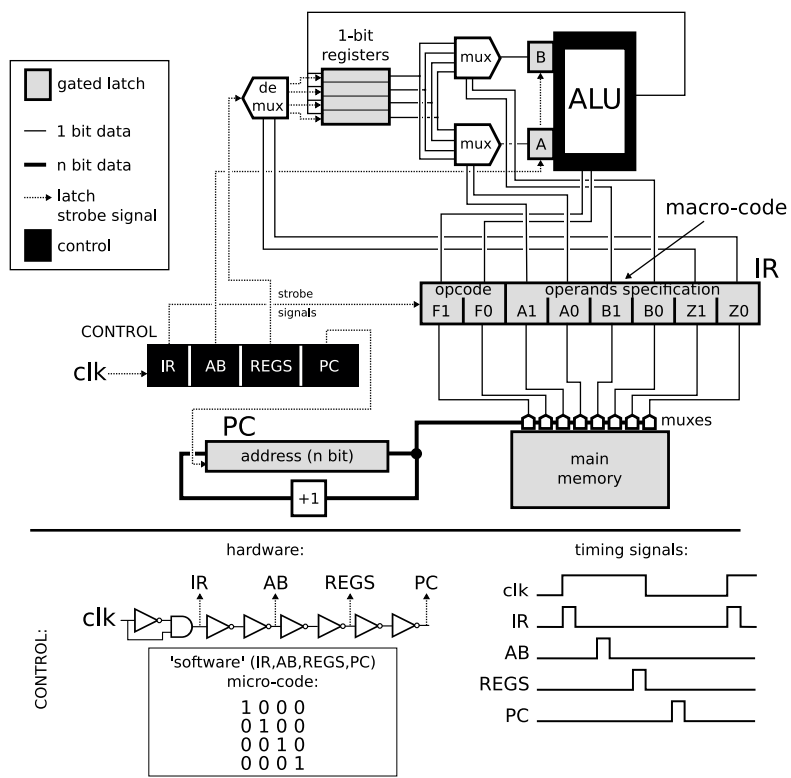


Figure 61: A rudimentary version of control logic of a 1-bit processor with four 1-bit registers, an 8-bit instruction register (IR), an n -bit program counter register (PC). ($2^n \times 8$ bits total memory size). The control logic is the black box generating 4 latch strobe signals, with the hardware inside the box and the timing signals coming out of it shown below.

ALU constantly performs all four possible operations, but only one appears on the output, namely the one selected by the opcode bits {F1,F0} of the instruction. After a while, the register selected by the {Z1,Z0} bits of the instruction register is receiving the REGS latch pulse to store the output of the ALU. Finally, the PC pulse increases the address with the magic '+1'-box that is not further explained here, but the PC in this rudimentary example can be simply a Moore-machine sequencer/counter; no jumps in the program can be made.

We can represent the control logic as hardware, we can also represent it as software, where the control-logic software program consisting of operations IR, AB, REGS, PC fixed in hardware can be written as

```

1000
0100
0010
0001

```

Such 'program' code is often called micro-code. Of course, real control logic is much more complicated than this example and the program can be rather complicated. It can be 'written' in hardware, or the program can be stored in ROM and copied to latches; maybe two signals are asserted at the same time. Maybe an engineer had an idea that the program counter can be increased already while the ALU is performing its operation and the control-logic code might be 0101, with the PC pulse coinciding with the AB pulse. The Figure just shows a very rudimentary imaginary example to give you an idea how it might work.

When the controlling consists of simple single-step processing this is called random logic. All operations are performed by single-step logic gates in the ALU. Alternatively, each instruction in the instruction register can consist of smaller micro-steps in the ALU performed by the controller. The program in the instruction register is called macro-code, whereas the instructions in the controller are micro-code. As example may serve an operation MUL \$3, \$2, \$1 (multiply the contents of register 1 to the contents of register 2 and place the result in register 3). In reality, the controller may translate this to many steps of shift-mask-and-add, since most ALUs cannot perform directly multiplications. (Just like humans cannot, by the way; we are also performing multiplications in various steps, as we have seen in the chapter on number systems, Ch. 2). In fact, micro-code is even on a much more basic level and consists of enabling outputs, etc. The controller will take care of running the micro-code and we will not go into detail about it. We can see it as a small CPU within the CPU.

6.4 Programming the CPU

We can now program this computer by placing a set of instructions in memory. Each of these instructions consists of what to do (the operation), with what to do it, and where to place the result (the operands). The controller will 'execute' the instructions by strobing input into flip-flop registers, multiplexing the chosen operation results to the output, and enabling components (taking them out of tri-state).

An ADD \$3, \$2, \$1 instruction (arithmetically add the contents of register 1 to the contents of register 2 and place the result in register 3) that will be loaded into the instruction register and executed by the control logic might look something like

100000	00011	00010	00001	000000000000
--------	-------	-------	-------	--------------

The first six bits represent the opcode (ADD), the next three sets of five bits specify the two input registers (often called 'source' and 'target', respectively) and the output register ('destination'). The last bits represent the type of adding, in this case two's-complement signed int. The controller interprets these bits and performs the right action by strobing the right components at the right time.

This is the so-called machine code, a representation of macro code in binary form. The code written in machine language. We can also represent machine code in hexadecimal to save space. In this case, we rearrange the 32 bit pattern shown above into 8 groups of 4 bits and convert each 4-bit group to a hexadecimal digit:

0b	1000	0000	0110	0010	0000	1000	0000	000
0x	8	0	6	2	0	8	0	0

The hexadecimal machine code thus 0x80620800. This is still a very awkward way of writing code. Fortunately engineers designed ways of writing much more comprehensible code, (macro) Assembly, which writes machine code in a easy readable way. We will treat Assembly in separate chapters in this book. We are now returning to the subject of how to do full arithmetic. Namely, how to do it on basis of the few instructions we have in hardware, ADD, SUB, AND, OR, XOR, and CMP.

6.5 Advanced arithmetic: Multiplication and division

Considering the fact that we have to our disposition only adding, subtracting, shifting, comparing, and the basic logic operations, how advanced arithmetic is done? For example multiplications and divisions.

Multiplication is done by the shift-and-add Russian-peasant algorithm presented in Chapter 2. The computer is performing this algorithm with binary numbers, which is very simple, as we have seen. The look-up table for binary number 1-bit multiplication of A and B is:

		A	
		0	1
B	0	0	0
	1	0	1

In summary: We shift operand A to the left and operand B to the right; if the LSB of B is 1, then add A to the sum. If not, then do nothing. Repeat until B is 0. An example of a long multiplication by this algorithm in binary multiplying $A = 13$ (1101) with $B = 6$ (011):

$$\begin{array}{r}
0\ 0\ 0\ 0\ 1\ 1\ 0\ 1\quad 13\ (A) \\
0\ 0\ 0\ 0\ 0\ 1\ 1\ 0\quad \times 6\ (B) \\
\hline
 0\quad (\text{operand A not added}) \\
0\ 0\ 0\ 1\ 1\ 0\ 1\ 0\quad (1\text{-left-shifted operand A added}) \\
0\ 0\ 1\ 1\ 0\ 1\ 0\ 0\quad (2\text{-left-shifted operand A added}) \\
\hline
0\ 1\ 0\ 0\ 1\ 1\ 1\ 0\quad = 78
\end{array}$$

No multiplication hardware is used for multiplications. Only shifting left and right, masking (AND'ing with the correct pattern, in this case: B AND 000...01 to determine the value of the last bit), and conditional additions. These were all described in the previous chapter (Chapter 5).

Note that this algorithm also works with negative numbers in 2's-complement, as long as we ignore any overflows. Take for example the 8-bit calculation of -3×3 :

$$\begin{array}{r}
1\ 1\ 1\ 1\ 1\ 1\ 0\ 1\quad -3\ (A) \\
0\ 0\ 0\ 0\ 0\ 0\ 1\ 1\quad \times 3\ (B) \\
\hline
1\ 1\ 1\ 1\ 1\ 1\ 0\ 1\quad (\text{operand A added}) \\
1\ 1\ 1\ 1\ 1\ 0\ 1\ 0\quad (1\text{-left-shifted operand A added}) \\
\hline
1\ 1\ 1\ 1\ 0\ 1\ 1\ 1\quad = -9
\end{array}$$

We do, however, need to use the *logical* right shift (and not the arithmetic right shift) for operand B, or the algorithm would never finish (right arithmetic shift of -1 results in -1). Once again we see here how brilliant the invention of 2's-complement is.



Division (of integers) is very similar to long division we learned to do on paper. In fact, it is again an algorithm based on shifting. In this case shift-compare-subtract. As an example, a division of 79 (1001111) by 6 (110), resulting in 13 plus a remainder of 1, is shown in Figure 62. To translate this to a computer (hardware) algorithm. If we perform operation a/b :

1. Left-shift one bit of **a** into a working register **x**
2. if (**x** >= **b**)
 - subtract **b** from **x** and left-shift a 1 into result register **y**
 - else
 - left-shift a 0 into result register **y**
3. if at LSb of **a**,
 - y**=**a**/**b**, **x**=**a**%**b** (remainder). READY
 - else
 - goto step 1

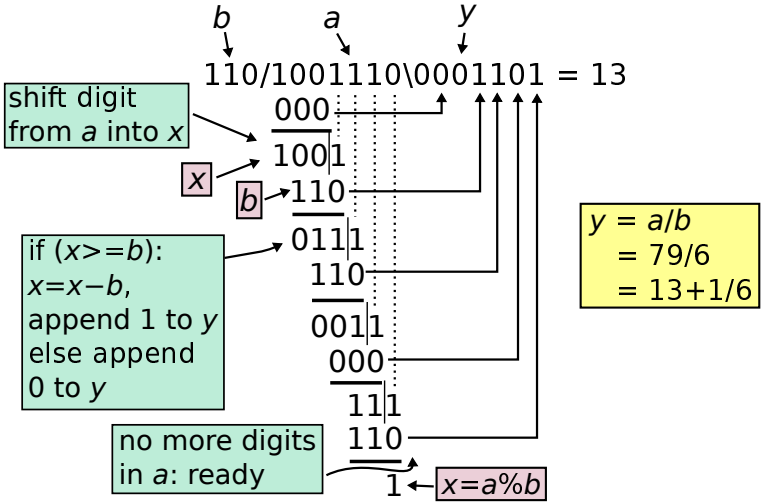


Figure 62: An example of a long division in binary.

As you can understand, it takes 32 left-shift-subtract cycles to divide two 32 bit numbers, compare this to the 32 right-shift and add cycles of multiplication. The interesting thing is that no additional hardware is needed to perform multiplications and divisions in a computer. They are all based on additions (with two's-complement numbers) and shift and mask instructions.

Of course, a lot of effort was spent on finding faster algorithms. A famous multiplying algorithm is named after Karatsuba: Imagine you want to multiply x and y , each $2m$ bits long. These both can be decomposed into

$$\begin{aligned} x &= x_1 \times 2^m + x_0 \\ y &= y_1 \times 2^m + y_0 \end{aligned}$$

each m bits long. Then a multiplication is

$$\begin{aligned} xy &= (x_1 \times 2^m + x_0) \times (y_1 \cdot 2^m + y_0) \\ &= z_2 \times 2^{2m} + z_1 \times 2^m + z_0, \end{aligned}$$

with

$$\begin{aligned} z_2 &= x_1 y_1 \\ z_1 &= x_1 y_0 + x_0 y_1 = (x_1 + x_0) \cdot (y_1 + y_0) - z_2 - z_0 \\ z_0 &= x_0 y_0. \end{aligned}$$

These multiplications can be performed on smaller (and faster) hardware. For instance we can do 32-bit calculations on 16-bit hardware (adders), since

the operands x_1 , x_0 , y_1 and y_0 are half the size of the full operands x and y . An example (for base 10 instead of base 2), $z = xy = z_2 \times 10^6 + z_1 \times 10^3 + z_0$:

$$12345 = 12 \times 10^3 + 345$$

$$6789 = 6 \times 10^3 + 789$$

$$z_2 = 12 \times 6 = 72$$

$$z_0 = 345 \times 789 = 272,205$$

$$z_1 = (12 + 345) \times (6 + 789) - 72 - 272,205 = 11,538$$

$$z = 72 \times 10^6 + 11,538 \times 10^3 + 272,205 = 83,810,205.$$

Where the multiplication by powers of 10 is done by left-shifting the appropriate amount of decimal cases.

Another approach of doing multiplications xy is by a look-up table that can be stored in code (microcode or macrocode). For 8 bit calculations (with 16-bit results) we'd need a two-by-two table of size $2^8 \times 2^8 \times 2$ bytes = 131,072 bytes. However, we can make use of the equivalence $(x-y)^2 = x^2 - 2xy + y^2$, or

$$xy = \frac{x^2 + y^2 - (x - y)^2}{2},$$

and use a one-dimensional table (i.e., vector) of squares only. For a multiplication of two operands x and y , we'd look up in the vector the squares of x , y and $x - y$, do summing and subtracting and divide by 2 at the end (a right-shift-one). A total of three vector-look-ups, two subtractions, one addition and a shift. For 8-bit integer multiplications we'd need only $2^8 = 256$ entries in the table, each 16 bit wide, giving a total of 512 bytes for our look-up vector. For 32-bit (4-byte) calculations we'd need $2^{32} \times 8$ bytes = 34,359,738,368 bytes, which is not very workable, making this only a solution for small integers.

6.6 Floating point; IEEE 754

As we have seen, computers are finite-state machines. Also numbers are limited by the size of storage elements. So, a byte can store positive integer numbers from 0 to 255, or from -128 to 127. As long as we stay within these limits, all possible numbers can be represented by these bit patterns. This is different for numbers from the real domain, \mathbb{R} . Even if we limit the range, for instance from 0 to 1, within that range the number of possible numbers is infinite. And thus they cannot be (all) represented by the finite-state memory elements. As we have seen in the chapter on numbers (Ch. 2), an elegant way of representing numbers from the \mathbb{R} domain is floating point. It has to be said beforehand that calculations in floating point are not (always) exact because of the number of states of a bit pattern is finite.

The IEEE 754 standard for floating-point numbers was developed to make it possible for engineers from various architectures to talk with each

Table XXVIII: Various lengths of IEEE 754 floating point numbers (float). The excess value used is equal to $2^{\text{nexp}-1} - 1$, with *nexp* the number of bits in the *exp* field. The (normalized) number is equal to $(-1)^{\pm}(1.\text{frac}) \times 2^{\text{exp}-\text{excess}}$.

floating point:

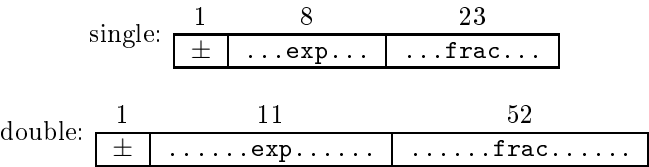
\pm	...exp...	...frac...
-------	-----------	------------

name	total size		\pm	exp		frac
	bits	bytes	bits	bits	(excess)	bits
half	16	2	1	5	(15)	10
single	32	4	1	8	(127)	23
double	64	8	1	11	(1023)	52
extended	80	10	1	15	(16383)	64
quadruple	128	16	1	15	(16383)	112
octuple	256	32	1	19	(262143)	236

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 is an industry standard described by the Institute of Electrical and Electronics Engineers (IEEE, pronounced as "eye triple ee") and has the following features for single floats and doubles (see Table XXVIII for other float types):



- It has three basic formats with different total bit-lengths: single (32 bits), double (64 bits), and extended (80) bits. As we will see, many architectures have hardware co-processors that can perform dedicated floating point operations with the first two types of numbers. Calculations with extended numbers often have to be emulated with software.
- One sign bit, at the position of the MSB. 0 = positive, 1= negative. It thus uses the sign-magnitude convention.
- 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, infinity and not-a-number).
- 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, `frac`, plus the leading "1." we call the significand: $s = 1.\text{frac}$; the final number thus being (for singles and doubles respectively).

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

(Note the strange mixed writing of the above, binary for the significand 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	000000000000000000000000

which translates into

sign: +

exponent: $\text{exp} - 127 = 126 - 127 = -1$

significand: $1.\text{frac} = 1.0$

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: a numeric data type that cannot be interpreted as a value, that is undefined or unrepresentable for instance $0/0$ or the square-root of a negative number). Together with the normalized numbers above they are described as:

	sign	exp	frac
Normalized:	\pm	— non-uniform —	— any —
	sign	exp	frac
Denormalized:	\pm	00...00	— non-zero —
	sign	exp	frac
Zero:	\pm	00...00	000000...000000
	sign	exp	frac
Infinity:	\pm	11...11	000000...000000
	sign	exp	frac
NaN:	\pm	11...11	— non-zero —

(In this 'non-uniform' means not all zeros and not all ones). Denormalized numbers use excess-126 instead of excess-127 for the exponent. The **exp** field containing only zeros, thus the exponent factor is always 2^{-126} . Moreover, denormalized numbers do not have an implied "1." in front of the fraction, but an implied "0." instead. The significand is thus $s = 0.\text{frac}$. On basis of this we can make the following observations:

- The largest positive normalized single-float number is in hexadecimal **0x7f7fffff**
exp=11111110, frac=11...11 [23 ones], with "1." in front, the significand is close to 2.0:

$$n \approx 2^{254-127} \times 2.0 = 3.403 \times 10^{38}.$$

$$n \approx 2^{\text{excess}+1}.$$
- The smallest positive normalized single-float number is in hexadecimal **0x00800000**
exp=00000001, frac=00...00 [23 zeros], with "1." in front, the significand is 1:

$$n = 2^{1-127} \times 1.0 = 1.175 \times 10^{-38}.$$
- The largest positive denormalized single-float number is in hexadecimal **0x007fffff**
exp=00000000, frac=11...11 [23 ones], with "0." in front, the significand is close to 1:

$$n \approx 2^{0-126} \times 1.0 = 1.175 \times 10^{-38},$$
 nearly equal to the smallest normalized number, and looking at the hexadecimal patterns we see they differ by 1. We see that the transition from normalized to denormalized numbers is seamless and the

Table XXIX: Range of IEEE 754 floating-point numbers using engineering notation (Exx represents $\times 10^{xx}$). nfrac is the number of bits of the frac field

Type	Largest ($2^{\text{excess}+1}$)	Smallest ($2^{-\text{excess}+1-\text{nfrac}}$)
half	6.5534E4	5.96E-8
single	3.4028235E38	1.4E-45
double	1.7976931348623157E308	4.9E-324
extended	1.189731495E4932	1.8E-4951
quadruple	1.189731495E4932	6.5E-4966
octuple	1.6113257175E78913	2.2E-78984

high-level programmer needs not to worry about the (de)normalization conventions of IEEE 754.

- The smallest positive denormalized single-float number is in hexadecimal

0x00000001

exp=00000000, frac=00...001 [22 zeros, one one], with "0." in front,

the significand is 2^{-23} :

$$n = 2^{0-126} \times 2^{-23} = 2^{-149} = 1.4 \times 10^{-45}.$$

$$n = 2^{-\text{excess}+1-\text{nfrac}}.$$

In which nfrac is the number of bits of the frac field.

It is clear from this that the *relative* error of numbers is rather constant for normalized numbers, but increases when we reach the lower limit of denormalized numbers. At the lower range, the distance between two numbers is equal to the lowest number and the relative error is 100%. Normalized numbers do not suffer from this effect and relative error is independent of the exponent in these numbers.

The range of single-float numbers are presented in Table XXIX, where also the range for doubles is shown. Note that calculations that result in a number smaller than the smallest value are rounded down to 0.0, while numbers larger than the largest number can cause an overflow error or are 'rounded up' to inf.

exercise: IEEE 754

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.



Multiplying floating point numbers is easy: multiply the two significands that are integers, and add the two exponents to result in the final significand and exponent, possibly after renormalizing the representation of the number.

$$\begin{aligned} f1 &= s1 \times 2^{e1-127} \\ f2 &= s2 \times 2^{e2-127} \\ f1 \cdot f2 &= s1 \times s2 \times 2^{e1+e2-254} \end{aligned}$$

and note that $s1 = 1 + \text{fraction1}$ and $s2 = 1 + \text{fraction2}$ so that $s1 \times s2 = 1 + \text{fraction1} + \text{fraction2} + \text{fraction1} \times \text{fraction2}$ which is a number between 01.0000... (2) and 11.11...1 (4)

Real hardware is a little smarter than that. That is why dedicated numerical co-processors exist (ex. 8087 from Intel). Take for example divisions in floating point. Basically the algorithm is that a division is a multiplication by its reciprocal:

$$\frac{a}{b} = \frac{a \times \text{reciprocal}(b)}{b \times \text{reciprocal}(b)} = a \times \text{reciprocal}(b).$$

The multiplication by a we can leave as the last step. The problem consists of finding the reciprocal $1/b$. We will first transform this into

$$\frac{a}{b} = a \times \frac{1}{b},$$

and continue with first solving this problem.

$$\frac{1}{b} = \text{reciprocal}(b) = \frac{1 \times \text{reciprocal}(b)}{b \times \text{reciprocal}(b)}.$$

So, it is all about finding the reciprocal of b by trying to make the denominator $d = b \times \text{reciprocal}(b)$ close to unity. In a co-processor the first approximation of the reciprocal is found in a look-up table. For instance, the first 5 bits (the look-up table is $2^5 = 32$ entries). The other bits are then found by iteration: The denominator d is close to 1 (with the objective of making it 1), so imagine we have a tiny error ε

$$d = 1 + \varepsilon.$$

Then, if we iterate to find a new reciprocal,

$$\text{reciprocal}(b)' = \text{reciprocal}(b) \times (2 - d),$$

then our new denominator d' becomes

$$\begin{aligned} d' &= b \times \text{reciprocal}(b)' \\ &= b \times \text{reciprocal}(b) \times (2 - d) \\ &= d \times (2 - d) \\ &= 1 - \varepsilon^2. \end{aligned}$$

This implies that we quadratically approach unity and find our reciprocal. A five-bit-accurate 1 in the denominator will become a 10-bit-accurate 1 after one more pair of multiplications.

An example: compute $1/b = 1.0/3.5$. The initial guess for $\text{reciprocal}(b)$ from the look-up table (entry 3; note that numbers larger than 10 or smaller than 1 can be scaled to fit within this table; a leading digit of 0 is not possible):

1	1.0000	6	0.1667
2	0.5000	7	0.1426
3	0.3333	8	0.1250
4	0.2500	9	0.1111
5	0.2000		

The starting value is 0.3333. Now, $d = b \times \text{reciprocal}(b) = 3.5 \times 0.3333 = 1.16655$. The next guess for $\text{reciprocal}(b)$ will thus be $0.3333 \times (2 - 1.16655) = 0.27778$. This is repeated and the result is 0.28549443, repeat and the result is 0.285714116. Not bad; my calculator tells me the 'real' answer is 0.285714285.

In any case, it is obvious that divisions are more complex, and generally a bit slower, than multiplications. This is good to know when you write

code in a higher-level language such as C++ or Pascal. It is better to write

```
a = b*0.2;
than
a = b/5.0;
or
a = b/(c*d);
instead of
a = b/c/d;
```

6.7 Advanced calculations

We have seen how a processor can do arithmetic adding. Then by applying the knowledge that the negative of a number is the 2's-complement of the number (inverting all bits and adding 1) with the same hardware we can subtract. Then we found out that multiplications are shift-add cycles and divisions are multiplications by the reciprocal. So, the only thing we need is shifting-and-masking and adding. But what if we want to do more complicated things, like generally evaluating functions $f(A)$? How do computers do that? We'll now see how this can be done too.

The first method is for a class of functions $f(A)$ for which we know the derivative of the function $f'(A)$, or a problem that can be inverted into such a situation. Take for example the calculation of the square-root x of an input value (argument) A , or in other words

$$x = \sqrt{A}.$$

How is this implemented into the processor (microcode)?* This 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 . Figure 63 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 x_0 . Successive iterations will lead to the x -value at which the function is zero. This will then be the square root of the argument. In other words, at each step

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

*Note that some architectures, like x86, have a square-root macro-assembler instruction (see **FSQRT** on page 316).

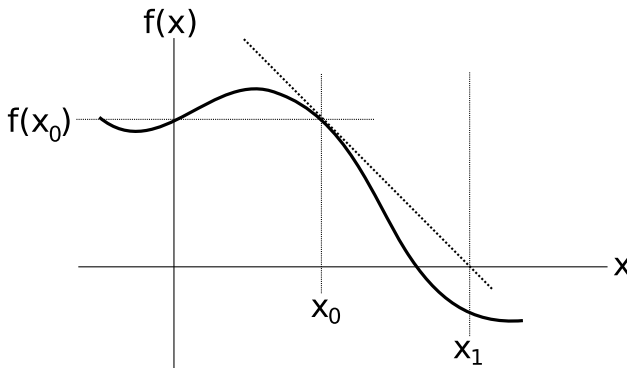


Figure 63: Method of Newton-Raphson for finding zeros in functions. Starting at a point x_0 , a new estimation x_1 is found based on the function value and its derivative at x_0 , namely $x_1 = x_0 - f(x_0)/f'(x_0)$.

In this case

$$\begin{aligned} x_{i+1} &= x_i - \frac{x_i^2 - A}{2x_i} \\ &= \frac{1}{2} \left(x_i + \frac{A}{x_i} \right). \end{aligned}$$

These iterations use simple multiplications, divisions and subtractions, all implemented in hardware and part of the 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. We continue until the difference between two iterations is less than a tolerance value, δx :

$$\Delta x = |x_{i+1} - x_i| < \delta x.$$

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, $\sqrt{30.0} = 5.48 \pm 0.01$.

Another set of problems does not have a known derivative, or these derivatives are as intractable as the original problem of finding the function. Take for example the function $f(x) = \sin(x)$. Its derivative is $f'(x) = \cos(x)$ and this does not help very much; we'd be as far away from home as where we were.

In this case we can use Taylor expansions. And exactly the thing the mechanical calculator the Difference Engine of Babbage was good at (See p. 72 and the dedicated section at the final chapter of this book). As an example, the sine function can be Taylor expanded to (x in radians), see Figure 64:

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} \dots$$

In general, any function $f(x)$ can be Taylor-expanded to

$$f(x) = S_0 + S_1x + S_2x^2 + S_3x^3 + S_4x^4 \dots$$

to minimize the number of multiplications, this would be translated by the computer into

$$f(x) = S_0 + x \cdot (S_1 + x \cdot (S_2 + x \cdot (S_3 + x \cdot (S_4 + \dots))))).$$

The constants S_i are stored somewhere in the computer (probably in a look-up table in the CPU) and the function evaluation is converted into a series of multiplications that the computer can deal with very well. In some cases special properties of the functions are used to facilitate the calculations. For example, it is known that $\sin(-x) = -\sin(x)$ and $\sin(\pi - x) = \sin(x)$ so we only have to calculate the value between 0 and $\pi/2$, the rest can be derived from that, see Figure 64.

The Taylor expansion above is done relative to $x = 0$, but this is not always the best case. In our sine example above, it would be much better to Taylor-expand the function exactly halfway the interval of interest, that means around $\pi/4$ since the maximum distance to the desired function is then only $\pi/4$ and the series converges much faster. The series would be

$$f(x) = T_0 + x' \cdot (T_1 + x' \cdot (T_2 + x' \cdot (T_3 + x' \cdot (T_4 + \dots))))),$$

with $x' = x - \pi/4$ and the constants T_i obviously different than the S_i constants.

In other cases this is even more obvious. Calculating the logarithm does not make sense in a Taylor series around 0, because it is infinite there. We'd better Taylor-expand around $x = 1$. Moreover, also in this case we can reduce the range of necessary function evaluation, since $\ln(ex) = 1 + \ln(x)$, so we can first determine in what range x is and add or subtract as many times 1 needed, while simultaneously adjusting x to a range between 1 and

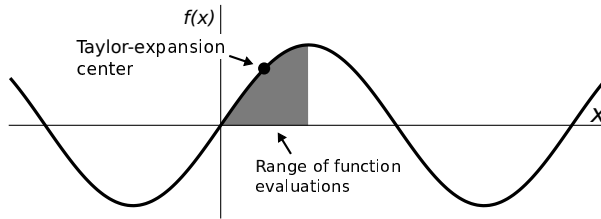


Figure 64: Method of finding a function value for $f(x) = \sin(x)$. Only a selected range is needed because the others can be found by mapping, $\sin(-x) = -\sin(x)$ for example. In the range the Taylor-expansion is done from the middle, so with $x' = x - \pi/4$.

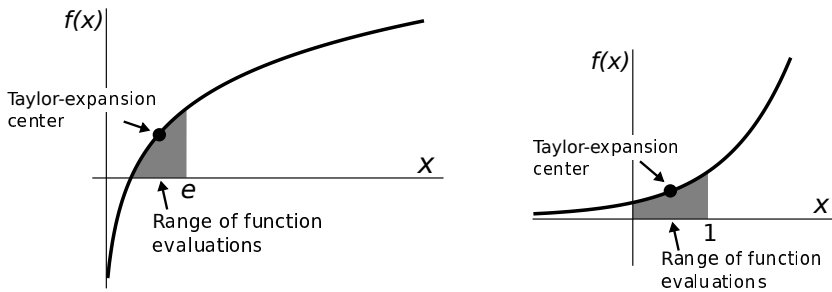


Figure 65: Taylor expansion of a logarithmic function and an exponential function.

e (see Figure 65). Even better, we do the logarithm calculation on the significand only and then add a value by looking at the exponent of the IEEE 754 number. Moreover, we can make use of the mathematical equality

$$\log_b(x) = \frac{\log_a(x)}{\log_a(b)},$$

and also calculate logarithms with other bases once any \log_a is implemented, for example $\ln = \log_e$. And if architecture engineers are smart, they'd start with the $\log_2(x)$ implementation, because then the exponent directly translates into an added number to the final result,

$$\log_2(1.\text{frac} \cdot 2^{\text{exp}}) = \log_2(1.\text{frac}) + \text{exp}.$$

and 1.frac is already nicely within the desired range from 1.0 and 2.0.

For the exponential functions, we can use the opposite, adjust the **exp** part of the float number by looking how far the argument is away from 0 – that is, the integer part of A , and then do a real calculation for an argument between 0 and 1. (See Figure 65).

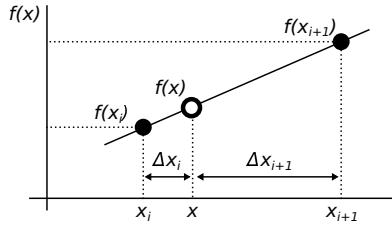


Figure 66: Any function value (\circ) can be approximately found by interpolation between two closest known function values (\bullet) calculated or known in another way.

Now that we have the logarithm and exponential functions, we can find any exponent x^a by the simple rule

$$x^a = e^{\ln x^a} = \exp(a \times \ln(x)),$$

which is the `powf(x, a)` function in C. (Note that the integer version `pow(i, n)`, i^n , is much simpler).

Another technique that is often used is interpolation, see Figure 66. A table of a limited number of function curve points can be stored in a look-up table and then the function value for any argument found by simply interpolating between two known points, assuming the function to be locally linear:

$$f(x) = \frac{f(x_i) \cdot \Delta x_{i+1} + f(x_{i+1}) \cdot \Delta x_i}{\Delta x_i + \Delta x_{i+1}}.$$

These methods are often done inside hardware (micro-code) or implemented in Assembly. More advanced and less-often-used functions are written in higher level program languages like (especially) FORTRAN and available in pre-compiled libraries. In general, readers might want to consult classic works such as *Numerical Recipes* in C, Pascal, Fortran.

7 | Information and memory

Memory is the capacity to store information temporarily or permanently. In the framework of computer architecture this means inside the CPU, but also in things like DRAM (dynamic RAM) chips and even external things like hard disks, SD (secure digital) memory cards, etc.

Before we continue, it has to be said again that the information stored is in a different domain than the physical state of the memory, be it a voltage at a gate, or a charge in a DRAM or hole in a punch card or ticker tape. One thing is a voltage, another thing is the information this voltage stores and represents. The latter is not part of the physical domain, but the virtual world of mathematics. How we humans *conventionally* interpret the physical state.

As an example, an 8-bit memory container can store a bit pattern that can either represent

- An unsigned short int or byte 0 ... 255
- A signed short int $-128 \dots +127$
- An ASCII (American Standard Code for Information Interchange) character (NULL, ... 'A', 'B' ... 'Z', ... 'a' ... 'z' ... See the appendix at the end of the book)
- A set of 8 state flags (printer on, paper empty, toner low ...)
- Two BCD digits, '00' ... '99'
- Anything else we can come up with. Pixels, audio volumes, etc.

It also has to be noted here right from the start that at the level of computer architecture, in contrast to the level of higher-level programming languages such as C, Fortran, Pascal, Basic, etc., there does not exist the concept of

variables. The only thing that exist are the memory elements and their addresses. Physically the contents of the memory and the control lines to have access to that specific memory.

Let's first take a look about the information itself and how to quantify it.

7.1 (Quantifying) Information

The smallest piece of information is a bit. It has two possible states. It can be 0 volt and 5 volt or 0 coulomb and 1 pico-coulomb in a physical sense. And we can represent it conventionally as 0 and 1 or false and true. In general, it can store the answer to a question that can be answered with "yes" or "no". But, not all questions are equal. Take a look at these three statements (meaning a "yes" to a question)

"It rained yesterday in London"

"It rained yesterday in Rome"

"It rained yesterday in Riad"

It is obvious that not all of these sentences contain the same amount of information. Take for example the first one. Most people will say "Duh! It always rains in London. Where is the news?!" We start getting a feeling here about what information is. That we are somehow surprised with the answer. The last sentence, about rain in Riad (a city in the middle of a desert) surprises us much more. So we feel it contains more information, although all sentences come from a type of question that has yes/no answers ("Did it rain in ..."). We can actually write this in a formal way, in what was done by telecommunications pioneers such as Shannon, Hartley and Khinchin:

The amount of information h in a message is proportional to the logarithm of the probability p of having received that message.

$$h = -\log p.$$

While the Information Theory does not specify a base for the logarithm, and any base will do since it is only about proportionality, for us informatics a base-2 seems adequate. Therefore, from now on, when we talk about logarithm, we mean one based on 2. Moreover, we define the unit of this information as 'shannon' (Sh) or 'bit' (b). Other units of information are the 'hartley' (Hart. Logarithm base 10, also called a 'dit', short for decimal digit) and the 'nat' (logarithm base e).

Rain in London is very frequent so the probability is close to 100% and the information in the message is nearly zero. Riad, however, is a dry place and the info that it rained there is very high. Imagine it was only 1%

probable. Then the amount of information is 6.6 bit. Strange this may seem, since it is a one-bit-question, does it rain in Riad?

These are *a posteriori* observations. How much information is in a message that we just received? More interesting from an informatics engineer's point of view is: How many bits of information will I need to send or save a message that I do not know yet? An *a priori* question. In that case we talk about the entropy of a message, the uncertainty we have about the possible answer.

The amount of entropy H of a message is proportional to the weighted average of possible amounts of information it can contain.

In other words, if there are n possible answers, each with p_i probability of occurring, the Shannon entropy is defined as

$$H = - \sum_i^n p_i \log p_i.$$

Take for example the flipping of a coin, it has two possibilities – heads and tails – each with 50% probability of occurring (if the coin is fair). The Shannon entropy of a coin flip is thus

$$H = -0.5 \log 0.5 - 0.5 \log 0.5 = 1 \text{ b.}$$

Moreover, if we get the result of a coin flip, for instance "tails", the amount of information is also $h = -\log 0.5 = 1$ bit. This is because the probabilities were symmetric, all p_i are equal to $1/n$. Then the *a posteriori* information is equal to the *a priori* entropy:

$$\begin{aligned} h &= -\log \left(\frac{1}{n} \right) = \log n, \\ H &= - \sum_i^n \frac{1}{n} \log \left(\frac{1}{n} \right) = \log n. \end{aligned}$$

The Shannon entropy is maximum for a system with all-equal probabilities. In case the probabilities are not equal, the entropy is always lower! Take for example a biased coin, with heads occurring with 75% probability and tails with 25% probability. The entropy is then $H = 0.81$ bit, 0.19 bit less than that of a fair coin.

$$H = -p \log p - (1 - p) \log(1 - p),$$

with p the probability of heads. This function has a maximum at $p = 0.5$, which is the case for a fair coin. (See Figure 67).

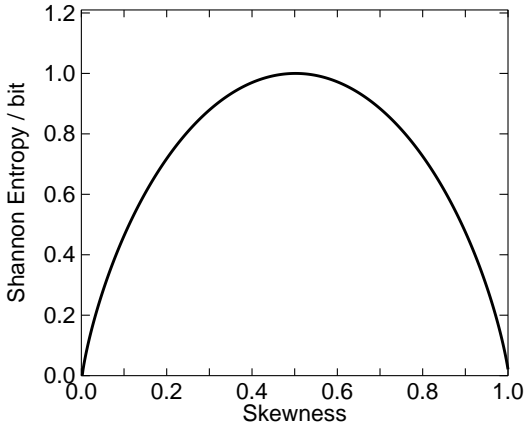


Figure 67: Shannon entropy H as a function of skewness p in a binary event, for example a coin flip. If the coin is always heads ($p = 0$) or always tails ($p = 1$) the information in a coin flip is zero. For a fair coin, $p = 0.5$ the Shannon entropy of a coin flip is maximum.

exercise: Odds

The 'odds' at a betting agency for a football match between Argentina and Brazil are given by

ARG win (1)	draw (x)	BRA win (2)
2.37	2.87	2.46

Meaning that, for instance, if you bet 1 euro on Argentina to win and Argentina wins, you get 2.37 euros. Assuming that these odds are proportional to the reciprocal of the probabilities of happening, $o_i = a/p_i$,

- a) What are the probabilities of the outcomes of Argentina winning, a draw and Brazil winning, p_1 , p_x and p_2 , respectively?
- b) What is the profit margin for the betting agency?
- c) What is the Shannon entropy of the match?
- d) If I tell you now that Argentina has won, what is the amount of information I just gave you?

Answer: $p_i = a/o_i$. Then $a/2.37 + a/2.87 + 1/2.46 = 1$, therefore $a = 0.849706$, and $p_1 = 35.85\%$, $p_x = 29.61\%$, $p_2 = 34.54\%$. The profit margin is $1 - a = 15.0\%$. $H = 1.58$ bit. $h_1 = 1.48$ bit.

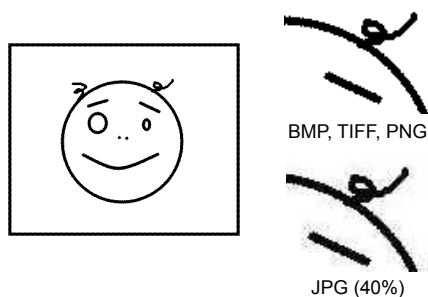


Figure 68: An image originally in BMP format can be compressed without loss into TIFF or PNG format, or with some losses [see in the right part a zoom-in] to JPG.

The difference between entropy and information, and the skewness of probabilities can be used to compact data. If not all bit patterns have the same probability, the information in the pattern is less than 1 bit per bit and compacting can be done. A bitmap (bmp) image that is fully black (0s for red, green and blue for all pixels, 000...0000) can be compressed into a much smaller lossless gif (graphics interchange format) or tiff (tagged image file format) or png (portable network graphics) image, or a lossy jpg or jpeg (joint photographic experts group) image that does not fully reconstruct to the original image. See Figure 68. The file sizes are

<i>format</i>	<i>size</i>
bmp	352 kB
tiff	468.2 kB
gif	5.8 kB
png	3.5 kB
jpg (40%)	11.7 kB

As can be seen, not all compacting algorithms are good for all types of images; for the image of Figure 68 tiff actually increases the size, and the lossy jpg is larger than the lossless png. Generally saying, for graphic art png seems the best, while for photos jpg works best.

A file that cannot be compressed has equal probabilities for all possible symbols contained therein. We call such information streams *ergodic**. Obviously, the algorithm for compacting the data depends on the type of data. Zip files are good for general data, png for images, etc.

Rather than using h , we can also define and quantize 'information' as the amount of uncertainty – that is, Shannon entropy – that is *removed* by the

*Ergodicity goes a little further than this. Imagine a stream of bits 010101010... This obviously has equal probabilities of 0 and 1, yet is not ergodic, since the next bit can be predicted perfectly on basis of the previous one and thus contains no information.

message. The output of a NAND gate (if all input combinations have the same 25% probability) is like our biased coin from above, biased towards 1 (75%), and the Shannon entropy is 0.81. Before I tell you the output of the NAND gate, you have 0.81 bit uncertainty about the output value. For you, the NAND-gate system is like a quantum mechanic probability wave function of a superposition of states (the NAND gate representing our Schrödinger cat) being both 0 and 1 simultaneously. When I then tell you that, after all, it was 1, the uncertainty fully disappears. The state 'collapses' into the known state, namely 1. The probability distribution is now: 1 = 100%, 0 = 0%. The Shannon entropy is then

$$H = -0 \log 0 - 1 \log 1 = 0,$$

thus $\Delta H = 0.81$ bit of uncertainty has been removed. A priori, an AND gate has 2.0 bit of input entropy and 0.81 output entropy. So on average 1.19 bit of information is lost in the AND'ing process. With the output of an AND gate we cannot determine what the two input bits were. Well, in one case we could (an output of 0 implies both input bits 1, but on average, in entropy terms, 1.19 bit is lost). Note that, in contrast to computing, useful communication (thus excluding prime time talk shows) consists of removing uncertainty, and maximizing ΔH per bit of data sent. We want to be able on basis of our received data to 'recover' what the data was emitted on the other side of the communication channel. Not so for computing.

In fact, computers are destroying information by the sheer way they work. If I write a computer program that processes all data (current and historic) from all weather stations around the world and weather balloons and then come up with a single prediction for the temperature tomorrow in my city, obviously a lot of bits of information are simply lost in the process somewhere and they are not recoverable (if not stored). We may hope that a tiny bit of uncertainty is removed by the weather prediction, but unfortunately even there the reduction in Shannon entropy is meager to say the least. Modern weather forecasts can basically predict it will rain tomorrow in London. So, maybe we should see a program as an artist sculpting a statue from a block of marble, cutting away everything that is not needed, to leave a beautiful piece of art. The laws of thermodynamics state that the pieces of marble can never be put back into the original block. It is a one-way process. Likewise, computing can produce beautiful results, but never again can the original state be recovered.

As an example may serve the AND-gate that take two bits of information, assuming they are independent and unbiased, the amount of information they bring is 2 bit (4 possibilities – 00, 01, 10 and 11 – each with 25% probability). At the output of the AND-gate, however, we have only 0.81 bit entropy (75% 0, 25% 1). A half-adder has two bit input entropy.

Table XXX: Amount of entropy H at basic circuits

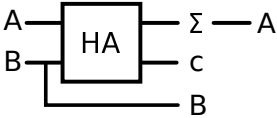
Operation	Input entropy	Output entropy
(N)AND	2.00	0.81
(N)OR	2.00	0.81
XOR	2.00	1.00
HA	2.00	1.50
FA	3.00	1.81

A	B	\sum	c
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

There are three possible outcomes for \sum and c; the combination $\{\sum = 1, c=1\}$ is not possible. The combination $\{\sum = 1, c=0\}$ has double probability. The output entropy is thus 1.5 bit. See Table XXX for the input and output Shannon entropies of some basic circuits. The AND-gate is communicating and erasing 0.81 bit of uncertainty. The rest, 1.19 bit, is lost forever, *if we do not store the input bits*. An AND-gate, on average, destroys 1.19 bit of information. Seen at it from another way: if we write a program that calculates the average of one hundred numbers, at the end we know one number only. We cannot invert the calculation and reconstruct the one hundred numbers. I'd know the average of the numbers but not the individual numbers. Or simpler, with two numbers. If I tell you the sum of two numbers turned out to be 8, you do not know if the numbers were 2 and 6, or 3 and 5, etc.

Yet, in some cases we can recover input data if we remember part of it. We can recover the rest on basis of the output data, if the sum of output entropy and the entropy of the remembered input part are equal or bigger than the total input entropy. An example is the arithmetic operation sum $A+B$. If we know the sum, say 10, and we know that B was 4, we can recover the information that A was 6.

This can be shown on the lowest hardware level. Take for example the half-adder operation on A and B, in which we overwrite the original value of A with \sum (after all, we have to store it somewhere), but keep the value of B:



The entropy at the input of the half-adder is two bits ($H = 2.00$). We might think that at the output we have three bits entropy, since we have three bits output. However, when we look at the truth table:

input		ouput		
A	B	A	B	c
0	0	0	0	0
0	1	1	1	0
1	0	1	0	0
1	1	0	1	1

We see that the output bits are not independent. We see that for the new A: $p(A=0) = p(A=1) = 0.5$ and the entropy for A is thus 1 bit. The entropy for B is also 1 bit because the conditional probabilities are $p(B=0|A=0) = p(B=1|A=0) = p(B=0|A=1) = p(B=1|A=1) = 0.5$. Yet, the carry c is then fully redundant – contains no new information – since it is fully predictable on basis of the new A and B, as the table above clearly shows. In fact, $c = B \text{ AND NOT } A$. (With A the new value of A). The output of a half-adder, when we remember one of the original inputs, has two bits of entropy; no information is lost. We can omit the carry from the truth table, which now becomes in terms of old and new values fro A and B as given below:

old		new	
A	B	A	B
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	1

The fact that no information is lost also means that on basis of the new values the old values can be reconstituted, the truth table of this reconstitution given by:

new		old	
A	B	A	B
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	1

Which, in fact, is a symmetric operation (compare the above two truth tables). Forward (old \rightarrow new) is the same as (new \rightarrow old). If we constantly keep one operand available (such as B in the example above) then we can run our operations backwards and calculate back to the beginning situation.

Note that this is a property of the XOR gate that has 1.00 bit of entropy and together with the independent bit of one input that was unaltered we can recover the original state. This cannot be done with any of the other standard gates. For example, the AND gate has 0.81 bit of entropy at the output and the original state cannot be recovered:

A	B	A·B	B
0	0	0	0
0	1	0	1
1	0	0	0
1	1	1	1

We can see here that an output of A AND B = 0 and B = 0 could mean that the original A was either 0 or 1. The information about A is lost in that case. The reversibility only works for XOR. This is relevant because XOR is the basic ingredient of arithmetic, as shown in the half-adder and full-adder.

As a curious consequence of this, we can exchange the value of two variables without the need of a temporary auxiliary variable, simply by arithmetic. Imagine we want to swap the value of variables `a` and `b` in programming. Most programmers will do this by

```
temp = a;
a = b;
b = temp;
```

However, we can also do this by

```
a = a+b;
b = a-b;
a = a-b;
```

because one operand is preserved in the sum-and-attribute operations (+ and = operations). For these operations we ignore the overflows and underflows that may occur, which are the full-number multi-bit equivalents of the carry at the one-bit level. An example is

instruction	a	b
a=3; b=5;	3	5
a = a+b;	8	5
b = a-b;	8	3
a = a-b;	5	3

As seen in Table XXX, an XOR-gate has a full bit of information at the output (as much as one input bit), we note that only 1 bit of information is lost by the XOR'ing. Thus the XOR operation is non-lossy in the context of the discussion here, and we can also exchange two 'variables' (bit patterns) by XOR operations that are significantly faster than the arithmetic operations above:

C language:	MIPS Assembly:
<code>a = a^b;</code>	<code>xor \$t0, \$t0, \$t1</code>
<code>b = a^b;</code>	<code>xor \$t1, \$t0, \$t1</code>
<code>a = a^b;</code>	<code>xor \$t0, \$t0, \$t1</code>

with \wedge the bitwise XOR operation. This is interesting. Information flows from `a` to `b` and from `b` to `a` without mixing halfway. This is like having two jars of wine in front of you, the left one with white wine, the right one with red wine, and being able to exchange them without the help of a third jar, and not winding up with rosé in the process. It seems to go against the laws of thermodynamics in terms of entropy. Such tricks, as the one above, are what Computer Architecture is all about. With a little bit of thinking we came up with an idea to speed up exchanging of information. Imagine this could be useful in a two-register architecture (like MOS 65xx that has only X and Y registers) where XOR'ing inside the CPU outperforms access to external memory. The only thing we are interested in is increasing speed and reducing resources needed.

7.2 Information sizes

As seen in the previous section, the minimum amount of information on a discrete, binary computer is 1 bit. It stores the answer to a question with a yes/no answer, i.e., two possibilities (irrespective of their probabilities). We can now start joining bits to have larger units, with more possibilities.

When we join four bits, this is what is called a nibble and has $2^4 = 16$ possibilities. Often it is used to store a BCD (binary-coded decimal) that has 10 possibilities, 0 ... 9, in which case we waste some combinations that will never be used (see page 18).

Joining two nibbles, giving a total of eight bits, is called a byte and often (for some historical reason) the referential unit of memory size. A typical computer (today) has some 4 gigabyte of memory. We will use here the symbol 'b' for bit and 'B' for byte. In a byte we can store information of an answer to a question that has 2^8 possible answers, or 8 questions each with 2 possible answers. It has to be noted that the number of possible answers – or states – is limited. Generally speaking memory with a size of n bits has 2^n possible states. We always have to bear this in mind. Even when we look

<i>name</i>	<i>bytes</i>	<i>number of possible states</i>
bit (b)	-	$2^1 = 2$
nibble	-	$2^4 = 16$
byte (B)	1	$2^8 = 256$
kilobyte (kB)	$2^{10} = 1,024^*$	$2^{(2^{13})}$
megabyte (MB)	$2^{20} = 1,048,576$	$2^{(2^{23})}$
gigabyte (GB)	$2^{30} = 1,073,741,824$	$2^{(2^{33})}$
terabyte (TB)	$2^{40} = 1,099,511,627,776$	$2^{(2^{43})}$

*: $2^{10} \approx 10^3$

Table XXXI: Memory sizes.

at the entire computer, with 4 GB (2^{32} B = 32 Gb = 2^{35} b) of storage, it is still a finite-state machine. A 4 GB computer has $2^{(2^{35})}$ possible states.

The most important aspect of information size of a computer architecture is: What is the size of the data the instructions work on? That is, if it performs an ADD operation, what is the size of the operands? We call this the word-size. Whereas bit, nibble, byte are standard concepts, the word size depends on the architecture. The first architectures, like the Intel 4004, used nibble-size registers and operations. Later architectures (like the 6510 used in the Apple and Commodore C=64) used byte-sized words. There is a tendency to increase the word size. A Motorola 68000 had sixteen/thirty-two technology meaning 32-bit registers and internal bus (and sixteen-bit external bus). Modern computers using AMD technology have 64-bit buses. It is unlikely that the word size will increase beyond this, because information sizes of 64 bit are more than adequate in most applications.

7.3 External memory

The external memory (outside the CPU), random access memory (RAM) is normally organized in packages of bytes. That is, the distance between addressable information is 8 bit. If address 0x10000000 points to a place in memory, 0x10000001 points to data 8 bits further on. ('0x' means hexadecimal). However, few architectures nowadays use 8-bit sizes. Take for example MIPS, the internal architecture of the MIPS processor is 32 bit (ALU operations, registers as well as the instruction register). That means that inside the CPU calculations are done 32 bit. In fact, the external bus over which the data is communicated is 32 bit, so that 32 bits are copied simultaneously in one clock cycle from/to the CPU to/from memory. We

call this 32-bit/32-bit architecture with both the external bus as well as the operations taking place in the CPU being 32 bit.

There is now a problem that is caused 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.

One immediate problem that may arise is that some data are smaller than 32 bits and only part of the data need to be fetched. This is a tiny problem because the controller can just fetch 32 bit anyhow and just ignore the part not needed. However, a more serious problem exists if we want to directly fetch data that is in an address not divisible by 4 (32 bits divided by 8 bits) and sometimes the hardware cannot deal with that. A fetch instruction to address 0x10010001 might not be possible. In such cases we must apply data alignment and place the data in the next address divisible by, in this case, 4, so 0x10010004, and waste 3 bytes of memory space.

This can also happen when the data is an instruction of program code. If an instruction is smaller than the 4 bytes, we must data-align the next instruction to start at an address divisible by 4. This can be done by placing no-operation (`nop`) (pseudo)instructions in our Assembly source code. Note that in some cases `nop` can also be used to delay the program. Maybe because the data from external memory has not arrived yet. This is of course, more the case in micro-code rather than macro-code.

Another problem that occurs is that, even if the data is of the correct size (for instance 4 bytes) we do not know *how* the information is stored into memory. Of the 4 bytes to store in a memory write, where do the most-significant byte (MSB) and least-significant byte (LSB) go physically in memory?

There exist two possibilities, as shown in Figure 69a 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 book of Jonathan Swift's book *Gulliver's Travels*. In the country of Lilliput 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.

In our case it is also completely irrelevant; as long as it is consistently done by the hardware, we couldn't care less how the information is stored, until we start looking at part of a word. Or when we communicate with a computer using a different endianness. 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. Figure 69b shows an example. The big-endian computer on the left has stored the information of JOHN DEFOE, age 23, office 264 into memory

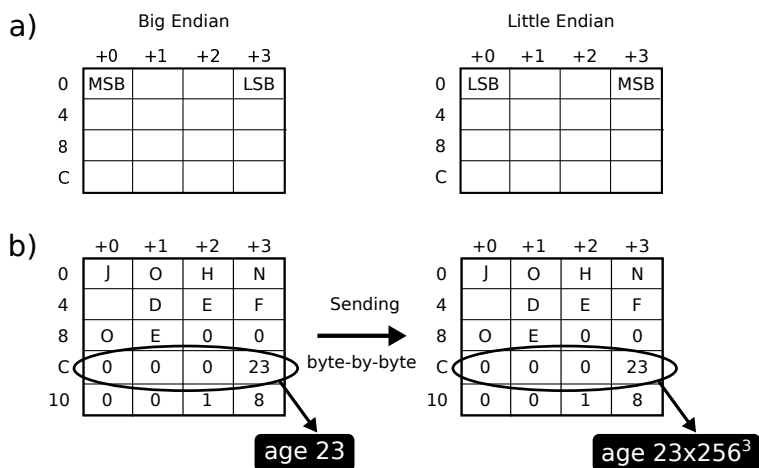


Figure 69: a) The difference between Big Endian and Little Endian storage of information is caused by the fact of having a different size CPU and communication bus architecture compared to that of main memory. In this case, MIPS has a 32-bit architecture while the addressing distance of memory is 8-bit. 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.

(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 O, etc. The little-endian computer receives a J and stores it in address 0, etc. So far so good. At the end it has completely copied the file of John Defoe. Now the right computer wants to print the age of John. The poor guy has suddenly aged quite a lot. The LSB (23) has become the MSB, and Jim is now $23 \times 256^3 = 385,875,968$ years old, and the building in which he works must be rather big, his office being $8 \times 256^3 + 1 \times 256^2 = 134,283,264$. In other words, we have to keep these things of endianness in mind when communicating with other computers. Note that also the placement of ASCII characters can be reversed, changing the name into "nhoJfed ". (Note the extra space). All depends on the architecture.

✱

We can now look at how external memory is made up physically. Figure 70 shows a 4 by 4 memory chip (4 words of 4 bits each). At the top it is connected to a data bus of 4 bits from which it can read and to which it

can write. On the left an address bus of 2 bits determines which of the four 4-bit words is selected for reading or writing by means of a decoder. The main component is a D-type flip-flop that copies the input at its D-terminal to its Q-output when a clock pulse is received. Any flip-flop receives this clock pulse if all the following conditions are met: the chip is selected (CS asserted), the chip is programmed to write (RD cleared), and the correct address is selected (determined by the four AND gates on the left). On the other hand, if output is enabled (OE asserted), no flip-flop receives the clock, but the output of the four selected flip-flops is copied to the data bus. In that case, also the chip has to be selected (CS=1) and read disabled (RD=0). Tri-state output makes sure that there is no data conflict with other chips.

We can use this chip of four-by-four memory or 2 address bits and 4 data bits, let's call it M2x4 (Figure 71), to design a larger memory layout made up of many of such chips. The essential idea is that CS (chip select) activates the relevant chip and the address lines activate the relevant bit within the selected chip.

But let's advance a little; the above example was only for purposes of explaining how it works in principle. Imagine we have 1-GB (8-Gb) chips that can communicate 32 bits simultaneously on the bus. It has 30 address bits (1 GB), so let's call it a M30x32 chip for convenience. We want to use them in a 32x32 architecture with a total of 4 GB addressable memory (RAM). That is, 32 address lines and a 32-bit data bus. We would need 4 such chips and thus use 2 address lines for chip-select logic. The rest of the address lines are fed to all chips.



In the above (specifically Fig. 70) we assumed that the memory elements were made up of D-type flip-flops and this kind of memory we call static RAM (SRAM). Static because the memory keeps its value (as long as power is supplied); no action has to be taken if no changes are needed. As we have seen, a flip-flop is based on half a dozen of gates and each gate on half a dozen of transistors, we can easily imagine a flip-flop to be made of thirty transistors or more. New technology makes use of dynamic RAM (DRAM). In this technology, a memory element is made of a single transistor connected to a capacitor, the latter storing an amount of charge that represents the memory state. Since a capacitor is basically a metal-insulator-metal device, it resembles very much a metal-oxide-semiconductor transistor. In fact, it can be made from a MOS transistor by joining the source and drain terminals, so effectively a DRAM bit is made of two MOSFETs, see Figure 73.

One of the transistors serves as an interrupter/switch; when the enable (gate) signal is high, it shorts the source to the drain. At that moment charge can be moved in or out the second transistor that functions as a capacitor. Once the write cycle is completed, the enable line is cleared and

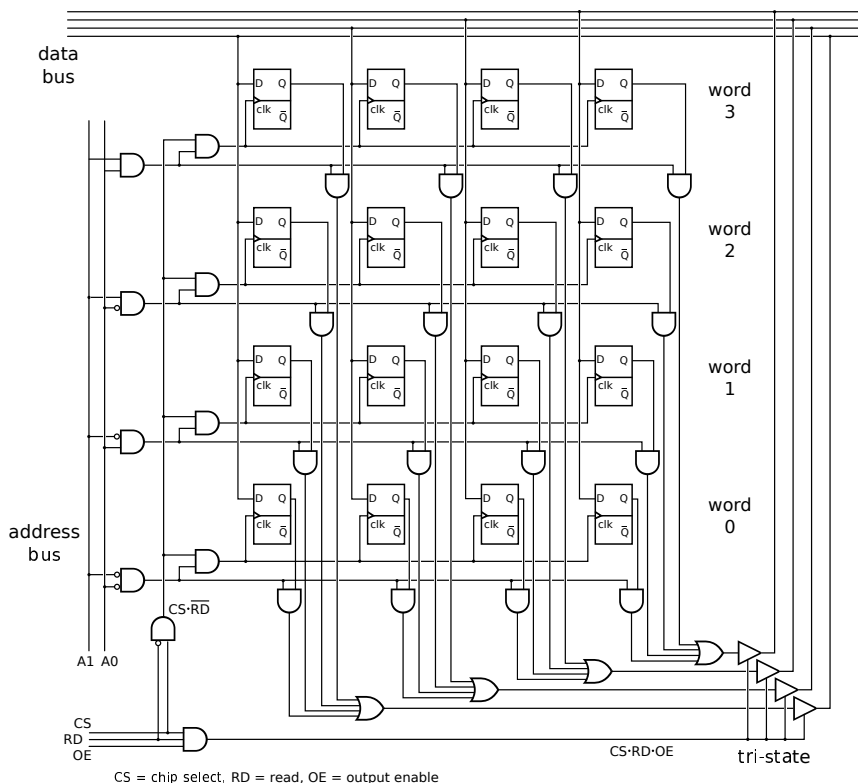


Figure 70: Physical construction of a memory chip. A 4-word chip (2 address bits) each word (row) made up of 4 bits (4-bit data bus). Three control lines determine the flow of information: Without chip select (CS) asserted, *this* chip will do nothing, not input, nor output. Read (RD, a signal to all chips) will determine the choice between a read and a write operation. Together with output enable (OE, all chips) these signals will make the chip a talker on the data bus by controlling the tri-state. The four leftmost AND-gates form a decoder (Fig. 54), on basis of the two address lines ('address bus') activating one row (word). The clock signals at the flip-flops are gated by this and CS and \overline{RD} to latch-in data from the data bus. Together with the vertically-oriented AND-gates and OR-gates at the bottom the decoder also forms a MUX (Fig. 52), selecting which data row (word) is ready to be placed on the data lines ('data bus').

the charge state fixed. Note that the write data (D) and read data (Q) lines are the same, in contrast to SRAM flip-flops.

Similar to the integration of flip-flops into SRAM chips, these dynamic memory elements can then be combined into a DRAM memory chip as

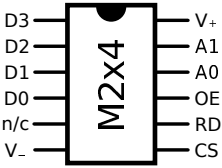


Figure 71: A four-by-four memory chip of Figure 70. Two address pins (A1, A0) determine which four bits of the four addresses will be put on the 4-bit data bus, or copied from the data bus (D3... D0). CS selects the chip, RD determines between reading and writing, and OE allows the chip to talk on the data bus. One pin is not connected (n/c)

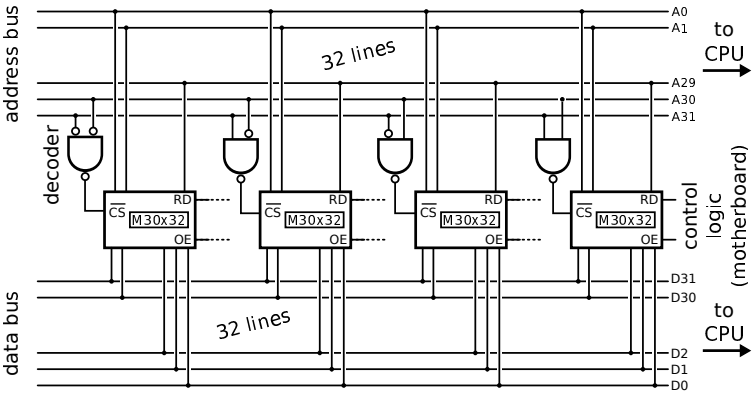


Figure 72: A full memory configuration. Four 1 GB-chips are joined together. A decoder determines which of the four chips is selected on basis of the two most-significant address bits. All the others remain inactive. The other 30 bits of the address are decoded to find the wanted bits inside the chip and place them on the data bus or to store received bits into.

seen in Figure 74. In a normal read or write, the control circuit asserts the correct address lines to enable the correct word for reading or writing. In idle mode, the control logic constantly keeps cycling all words, reading their bits and capping the charge in the capacitors by a read-write cycle. Hence the name *dynamic* RAM or DRAM. This extra, rather complicated, hardware is a small price to pay compared to the tremendous simplification of the memory elements (compare Fig. 74 with Fig. 70); with the same amount of transistors many more memory elements can be made.

The above memory elements, both SRAM as well as DRAM, are so-called volatile memory, which means that information is lost when power is switched off. In some cases we want the information to be permanent, or semi-permanent. The latter meaning that information is not lost upon

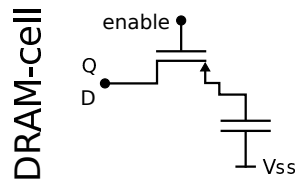


Figure 73: DRAM 1 bit cell. Once a 1 is placed on the enable pin, the transistor opens and the capacitor becomes accessible. It means that we can read it (Q) or we can write into it (D), the same pin serving both functions.

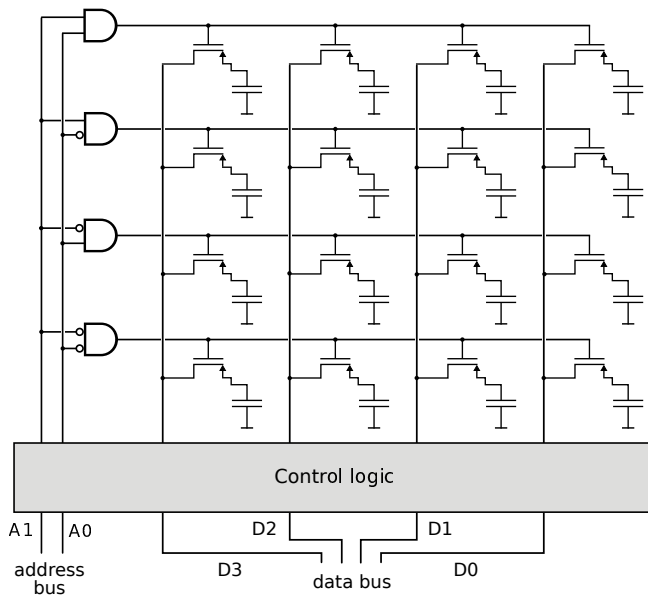


Figure 74: DRAM chip. A control circuit, when no external request for read or write is received, constantly cycles all memory addresses, reads them and rewrites them to recharge the capacitors.

a power cycle, but even so information can be written in it. In the case the information is permanent and cannot be erased or overwritten, we call it read-only memory (ROM). The technology is not very complicated and resembles dynamic RAM, but with the transistors not connected to capacitors, but directly to ground (logical 0). Moreover, the transistors data lines are also connected to a pull-up resistor, see Figure 75. That means that if there is no other thing connected to a vertical data line, the voltage on the line will be V_{dd} (logical 1). If the horizontal line coming from the address decoder is asserted (1), all transistors connected to this line are switched on

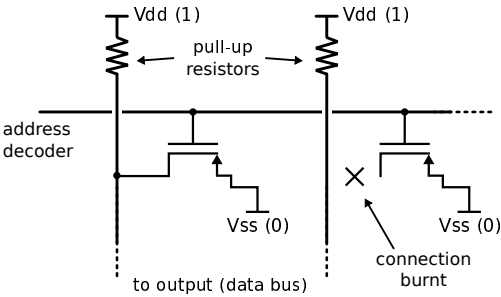


Figure 75: Two bits in a ROM memory chip. If the horizontal line coming from the address decoder is asserted (1) the transistors switch on and output 0 to the vertical data lines. However, if the transistor is burnt off (\times), the output will be 1 because of the pull-up resistor.

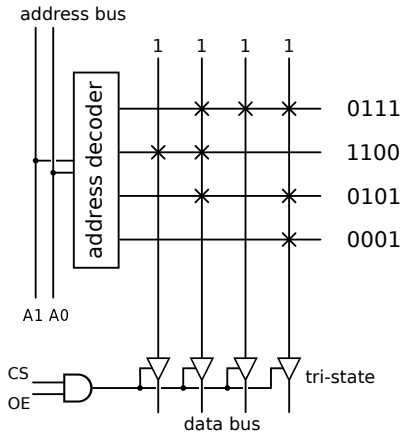


Figure 76: A 4x4 ROM chip based on the configuration described in Fig. 75. A ' \times ' indicates a burnt-away transistor and thus an output of 1. Without a \times at the intersection between the data line and address-selected word line the output will be zero.

and the voltages on the vertical data lines are low (V_{ss} , logical 0) because it 'wins' over the V_{dd} connections because it has lower resistance. We can burn away the connections of the transistors (indicated with a \times). In those cases, the data output remains high (logical 1). So, if we burn the connection (\times) we get a 1 and when the connection exists we get a 0.

We can use this approach in a full read-only memory chip. This is shown in Figure 76. A cross (\times) in this Figure indicates a burnt transistor connection and thus a logical 1 at the output (if that word is selected by the address decoder).

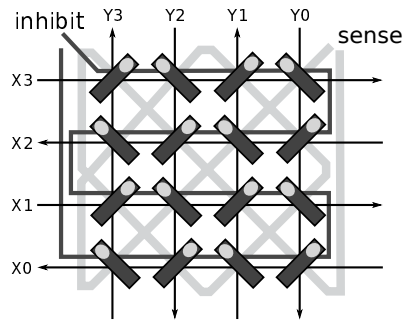


Figure 77: Magnetic core memory.

In other technologies, the user can write codes once after the virgin chip comes from the factory. This is s-called PROM, programmable ROM. Other versions also exist where the code can be erased by exposure of the chip to ultraviolet light (EPROM, erasable programmable ROM) and in some cases it can be done electrically (EEPROM, electrically-erasable programmable ROM). The latter is done by having a floating gate at the transistors that can store charge by applying a high voltage to it. This comes very close to the CMOS memory that exists in many modern camera's. In that case the charge at the floating gate comes from the light.

*

From a historic perspective it is interesting to mention magnetic core memories. They were based on the semi-permanent magnetizability of ferrite cores, see Figure 77. Through each core pass four wires; X and Y 'address lines', a 'sense' wire, and an 'inhibit' wire. The polarization of these cores could be programmed by passing a current through their center. Because this process was non-linear (a recurring *essential* element in digital electronics), the core is only programmed when the sum of magnetic fields from the currents of all wires passing through them is above a certain threshold. To select a memory location $\{X,Y\}$ the corresponding lines are driven at half current – 'half select'. Only the combined magnetic field where the X and Y lines cross is sufficient to change the state of that core; other cores will see only half the needed field, or no field at all. The direction of the sum current then determines the bit value, for instance resulting clockwise magnetic field as '1' and anticlockwise as '0'.

To read a core state, the circuit tries to write a value to it, for instance a '0'. If the state was already '0', it stays '0'. However, if the bit was previously '1', the core changed magnetic polarity and this change causes a voltage pulse on the sense line. The detection of the pulse meant the state was read as '1'. It is obvious that after a read, the state is always '0' – memory access is then called a 'destructive read' – and thus the state '1'

has to be rewritten if that was the read state. To write a '1', the currents in the X and Y lines are driven in the opposite direction, to cause an opposite magnetic field compared to the (write '0') reading phase currents. To write a '0', together with the X and Y currents an equal intensity 'inhibit' current is supplied by the inhibit line so that the total current is only half of the threshold voltage to program the core.

The magnetic-core memories had an access frequency of about a MHz, and some kilobits of total memory. Because of their high cost of fabrication, by the end of the 1970s they were no longer used and are only described here for historic reasons, and to show that transistors are not the only way to implement (binary) logic.

7.4 Memory speed-up

As we have seen, external memory – outside the CPU – consists of either D-type flip-flops (SRAM) or capacitor-transistor pairs (DRAM), with the latter being significantly slower but also significantly cheaper. Inside the CPU, there are ultra-fast registers (D-type registers).

Apart from that, most modern advance architectures have so-called cache memory. To explain why this exists, let's do a calculation. Executing a program in a computer consists of fetching instructions and data from memory and storing data resulting from calculation back into memory. The bottleneck in the overall speed of the computer – the so-called Von Neumann bottleneck – is the speed of communication between CPU and memory and this has real physical limitations. To give you an example, if the memory card is 10 cm away from the CPU, it takes $(0.1 \text{ m})/c$ time for the request for information to reach the memory chip, and an equal time for the data (or instruction) to be sent to the CPU, even if the presentation at the output of the memory chip is immediate. With c the speed of light (equal to $3 \times 10^8 \text{ m/s}$) that is twice 0.33 ns and the maximum processor speed would be $1/(0.66 \text{ ns}) = 1.5 \text{ GHz}$.

To circumvent this, engineers have invented cache memory. That is a relatively small amount of memory inside the CPU that holds a copy of external memory. In most cases we are working on a tiny part of the vast external memory, so if we keep a copy of this locally and every now and then synchronize the two (external memory and cache), enormous speed increments can be achieved. Of course, the price to pay is a larger overhead – control logic – inside the CPU. To make things even more complicated, since the closer the memory is to the ALU, often several levels of cache are present in the CPU. Each level further away from the ALU, bigger in size, and slower. A complicated control logic (running on microcode) is actually doing predictions (heuristics) about what memory is going to be needed in the future.

Another technology is to avoid the CPU altogether. Imagine we are reading a file from a hard disk, maybe a word processing program or something of the kind. Instead of the CPU reading a byte from disk and writing it in memory (twice using the data bus; a read from disk and a write to memory), in direct-memory access (DMA) a channel is opened between hard disk and memory and a stream of data is poured directly into memory, without any byte ever going through the CPU; the CPU merely sets up the channel by communicating to the controller on the motherboard what it wants. Apart from the fact that it is directly twice as fast, it also liberates the CPU for doing other things. This is especially possible when the CPU has a cache and needs no connection to the data bus, which is occupied by the DMA transfer.

7.5 Software aspects of memory

After having described the hardware aspects of memory, let us now take a look at the 'software' aspects. The way the memory looks like in terms of organization. Remember, Computer Architecture deals with both hardware and software, and this also applies to memory.

The first thing to observe is that the external memory holds both the data to be processed, as well as the program code (instructions). We call this a Von-Neumann architecture (or Princeton architecture). The technical description is "any stored-program computer in which instruction fetch and data fetch cannot occur at the same time because they share a common bus". If we want to load a piece of code in the instruction register, or a piece of data in a data register, these bits are transferred over the same (data) bus. Contrasting this is a Harvard architecture that has two separate buses for data and program instructions. Before Von Neumann data was stored in memory (for instance in magnetic cores) and the program was mechanically entered by engineers by setting switches. Von Neumann realized this is too cumbersome and inflexible. Why not have the program residing in the same memory so that it was *physically* indistinguishable from data.

In a Von Neumann computer it is possible that a program changes itself, since both reside in the same physical memory and use the same physical communication bus with the CPU. I would strongly advise everybody to avoid this 'dirty programming' technique, which fortunately is anyway made impossible by most of the assemblers. That is, it is a software limitation (of the Assembly compiler) and not a hardware limitation.

On the other hand, lifting this limitation allows for the concept of mixing data with code, as in object-oriented programming (OOP). As we know, a structure in C (or record in Pascal) consists of a set of data of different type. Each piece of data in a so-called 'field'. In object-oriented programming, an 'object' can contain data fields ('properties'), as well as function code

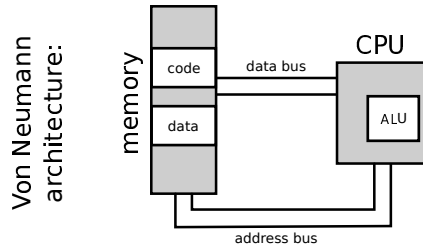


Figure 78: A typical Von Neumann architecture. Both code and data reside in the same memory and are transferred to the CPU by the same data bus, but they are in different parts of memory, the code segment and data segment relatively.

(‘methods’). So, instead of `printf("%s", stringptr)`, with `printf` the function and `stringptr` the (pointer to the) data, we would have a `String` object that if we want to print it we’d call the print ‘field’: `myString.print`. The point is that data and code are logically mixed and can even be so physically (although technically speaking still not necessarily).

Classically, while data and code reside in the same physical memory, they do so at clearly separated addresses. We normally have a data segment as well as a code segment. And the code acts solely on the data. A program reads and writes in the data segment. However, this cannot be an absolute requisite, because if that were the case, how does a program get loaded into memory?! It must be so that a program (which we call the operating system, OS) loads the code from disk and puts it in memory. Now, for that OS program, the code of the program is *data*. It operates on it, albeit rather simple operations (load-store, if no DMA is used). Obviously, a program (at least those of the OS) must have a permission/possibility to write in the code segment.

The address of an amount of data can also be called data. This type of data or variable we call a pointer. We already know this from C language. If we declare an array – of chars, called a string, or maybe a two-dimensional array of floats – the compiler remembers the pointer to the start of the array. Take for example the C declaration

```
float a[3][3];
```

What the compiler does (translating it to Assembly or directly to machine language) is

- Virtually, by the bookkeeping of the compiler, reserve space in memory so that next declarations are saved further down in memory. This ‘reserving’ is nothing more than advancing the compiler status value of the next-available address by the amount needed. Declaring a float,

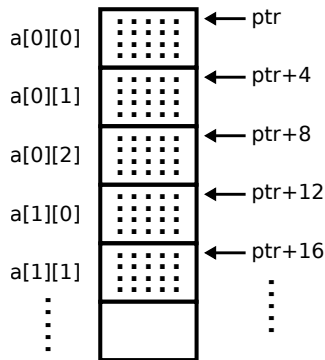


Figure 79: The mapping of a multidimensional array onto linear memory. The compiler or Assembly programmer has to *calculate* where elements reside. In this case an array `a[][3]` of floats. Note that the first dimension size does not need to be known for the calculation.

for example, advances this value by the size of a float (4 bytes). (Real memory allocation is done by the operating system; at the time of loading the compiled program the OS is allocating memory for it, for both the code and the data segment. Dynamic memory allocation is done at run-time by the program by calling the OS through the `malloc()` function).

- Remember the starting address of the array in a label-look-up table, so that every time it now finds a reference to the array in the source code, it substitutes this address
- Remember the specifications about the dimensions (except the first one, as we will see) as well as the size of one element (for instance, a float is 4 bytes).

If a user is making a reference to an element of the array, the C compiler adds code to calculate the physical address of the element. In Assembly *we* have to do this calculation!

How is this done? Remember that memory is a linear vector; a memory element only has one index/address. Therefore we – the Assembly programmers, not the higher-level programmers – have to calculate the address, to do a mapping from n -dimensional array to 1-dimensional memory space. Figure 79 shows how this is done. Assuming that the first index of the array (the line number) has higher significance than the last index (column) and the array starts at address `ptr`, float element `a[i][j]` resides at

$$\text{address} = \text{ptr} + \text{sizeof(float)} * (\text{numcol} * i + j)$$

As you see, we need to know the number of columns (`numcol`) but we do not need to know the number of lines (`numlin`) in the array. Now you know why in C we can pass arguments to functions as, for example, in this function declaration:

```
int function(int p[][3]){
    p[1][2] = ...
}
```

after which we can use element `p[i][j]`. We can, but we do not *need* to specify the number of lines, since it is not needed to calculate the address of any element in the array. Of course, the programmer has to be careful not to overwrite the adjacent objects. Assembly permits everything. (YWIYGI paradigm; you wanted it, you got it). The `function` merely receives a pointer `p` and the compiler (sic) adds code for calculating where element `p[i][j]` resides. For that it needs to know the number of columns. An alternative form would be letting the C-programmer calculate it

```
int function(int *p, int numcol)
```

and the C-programmer has to calculate the address; `p[i][j]` is in reality `p[j*numcol+i]`. Try it out! This last version above is the closest thing to Assembly. By now you should have understood that the following is not allowed:

```
int function(int p[3][])
```

because the address of elements cannot be calculated. Note also that

```
int function(int x, int y, int p[y][x])
```

is not allowed (in C) because, although it makes perfect sense for humans, a compiler does not know how to deal with it statically. Only with the introduction of dynamic programming languages such as C++ are the above declarations permitted. An object int-array with dimensions `x` and `y` will be created at run-time, the time at which the size of the object is known.

7.5.1 Heap and stack

There are two way things can be placed in the data segment in memory: heap or stack. The heap is what we can call 'classic' memory. Any object can be referenced at any time. It is classic RAM (random-access memory). When a program is compiled, memory is reserved for the objects in the data segment and pointers to the objects saved, as described above. No value is saved in it (no initialization* of 'variables' is taking place). This way of data

*Initialization is assigning an initial value to a variable.

organization on the heap is called static memory allocation. The amount of reserved space is constant during the program. It can be rather inefficient, because we might not know how big the data is that we are going to process and we would then always have to reserve space with the maximum size we can imagine, even if we are just going to find the average of an array of 2 elements, we'd have to do the equivalent of declaring an array of a million elements (and thus occupy the data segment with a million zeros or irrelevant leftover 'garbage' values), reserving space for a million elements in RAM.

Memory can also be reserved at run time. The procedure is similar to the one above and is called dynamic memory allocation. We have a static pointer-type variable in the heap and make a system call to the operating system to reserve space for us on the heap at *run time*. The operating system will return the address of the start of the allocated memory to the calling program and that program can now use the memory at its leisure. This is a dynamic way of reserving space and that enables consuming the memory with exactly the amount necessary, never claiming unneeded memory. The equivalent in C is the `malloc()` function call. And, as we know from our C lectures, don't forget to release the memory once it is no longer needed. Fortunately, the C compiler is adding code to our program to do that at the exit of our program. If even that fails, the operating system keeps track of what process reserved what memory and if the process or thread crashes, the operating system cleans up our mess.

A third way of reserving space in memory is by placing the objects on the stack. This is the most flexible and allows for recursive function calls, where each function call gets its own set of local variables. It works as follows.

A stack is a place in memory where we only have access to the top. We can place a new item on the stack, or we can remove the top items, but otherwise we cannot do anything. We do not have access to items below. Now, imagine what happens when we call a function `sum` that has three local variables `a`, `b` and `c`.

```
int sum(){
    int a;
    int b;
    int c;
    <...code...>
}
```

The moment – at runtime, so dynamic allocation – the function is called, the local variables are placed on the stack at the address pointed to by the stack pointer (`$sp`) and the stack pointer is adjusted to the last item on the stack. The variables work as one item and can all be accessed by the function called. See Figure 80. They stay on the stack until the end of the function, whence they are 'deleted' (that is – why waste effort – simply the stack pointer is adjusted). It is obvious from the way the stack works

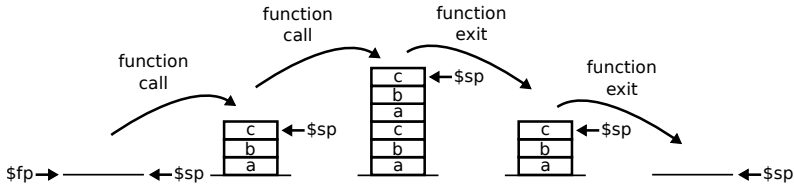


Figure 80: When a function is called that has three local variables, *a*, *b* and *c*, these are dynamically (at run time) placed on the stack at the place pointed to by the stack pointer *\$sp*. Once the function terminates, the items are removed from the stack. This allows for recursive functions, where every function call has its own private set of variables. The frame pointer *\$fp* always points to the bottom of the stack and keep an eye on avoiding stack underflows (removing an item that does not exist on the stack).

that only the last item can be removed. That is why this type of memory organization is called last-in-first-out (LIFO).

The interesting thing is that this allows for recursive function calls (functions calling themselves). As can be seen from the figure, each function call has its own private set of variables. Moreover, it cannot access the variables from the function call below it. Recursivity is difficult to implement without the concept of a stack and was thus an emergent property when the stack was invented. Moreover, the idea of local variables is also easily implemented. The main program (the one that calls the function) does not have access to the local variables of the function, because they do not exist yet! The functions do have access to the global variables because they are placed on the heap and exist as long as the entire program is running until it is 'killed' by the operating system.

Placing an item on the stack is called 'push' and removing it 'pop'. A function call pushes local variables on the stack and leaving a function is popping them off the stack. Often a frame-pointer (*\$fp*) is also used. This is a constant pointer (within the context of a program) and allows for checking if we do not get stack underflow (trying to pop off the stack more items than exist there).

Some program languages, such as Forth and Postscript (a printer language), uniquely use the stack and can be called stack-oriented programming. It is quite an interesting concept but takes some time to get used to. An example is the Forth code

```
2 3 4 + * .
```

which means: 1) put 2 on the stack, 2) put 3 on the stack, 3) put 4 on the stack, 4) pop the top two elements (4 and 3) from the stack, add them and put the result (7) on the stack, 5) pop the top two elements (7 and 2) from the stack, multiply them and put the result (14) on the stack, 6) pop the

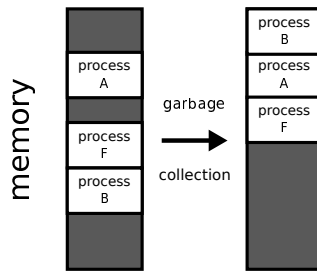


Figure 81: garbage collection. Processes and threads are relocated in memory to leave less small gaps and leave space for big programs.

top element from the stack and print it. The result, obviously, is 14. (Try it; `gforth` is freely available in most Linux distributions. If you try it, also try an extra 'print' represented by the dot. As you will see a "stack underflow" will occur).

The stack itself is created somewhere in RAM. Our program has to specify how much stack it will need and the compiler adds a `malloc()` at the beginning of our program to reserve that amount of space in memory, and saves a pointer to the bottom and top of the stack. In fact, if we have stack-checking option switched on in our compiler, code is added to our program that verifies if the actual stack pointer is not outside these limits, generating a stack-overflow exception in case it is outside this range.

7.5.2 Garbage collection, paging, and overlays

In modern computers many programs are running 'simultaneously' (that is to say they are all residing in memory and get allotted time slices of processing). These are called processes or threads, the difference between them being that threads share memory while processes do not. So a process can have many threads, but a thread belongs to a single process.

The problem with threads and processes is that they are created and become extinct dynamically. Every time a thread is created (by the operating system), space is reserved for it in memory (the heap). And because the threads and processes are not necessarily destroyed in the (reverse) order they were created, gaps can fall in the memory space. To clean up this swiss-cheese structure, the operating system every now and then performs what is called a 'garbage collection'. It relocates all processes and threads so that they are nice adjacent and no holes exist. See Figure 81. Now if a big program starts, there will be place for it in memory. This concept is very similar to disk defragmentation techniques that were very common in older hard disk architectures.

Of course, moving a program from one place in memory to another has

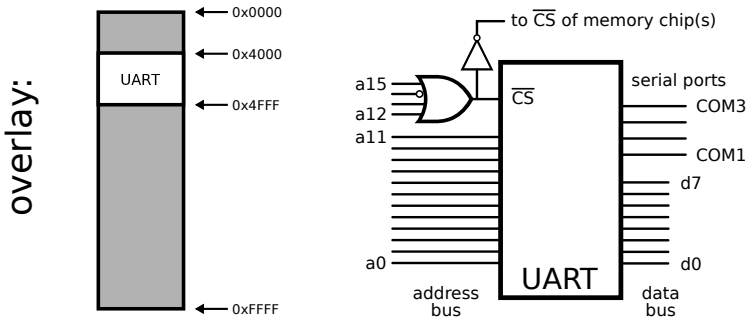


Figure 82: An overlay consists of shadowing the RAM memory and activating a hardware chip for certain addresses only. In this case a UART.

to be done with care. In case the program has absolute addressing methods, these might have to be translated and the address references updated. Fortunately, most programs use relative addressing modes and we can simply shift the entire program to a new place; garbage collection does not affect the functioning of the programs. See for example the jump and branch instructions of MIPS (Sec. 10.3 of Chapter 10). The jump instruction (*j*) is relative to the page of the program counter, while branching instructions (*beq*, etc.) are relative to the program counter. That way moving a program to another memory location can be done without having to change the code.

Another way of avoiding problems when moving processes in memory is by having dedicated hardware to translate logic addresses, as referenced by the code, into real physical addresses by a so-called memory-management unit (MMU). The code makes a reference to a certain address and the MMU translates this into a physical address. This way, when a process is moved in memory, the code can stay the same, only the MMU has to be informed about the new location of the process in memory for it to be able to do correct address translation. The x86 Intel family used such paging techniques but recent versions (AMD 64) have dropped most functionality of this type.

However, such techniques also allow for something that is called virtual memory. When an address is referred that is outside the range of physical addresses they can be loaded from the disk instead, possibly loaded entire pages at a time (for instance 4 MB) instead of loading individual bytes. This applies to data as well as code and was invented by researchers at Manchester.

Another technique often used with respect to memory organization is overlays. When a certain range of addresses is referenced, instead of the data coming from and going to the memory, it actually goes to other hardware, which then deals with the data instead of just storing it in memory. As an example might serve an UART, universal asynchronous receiver/transmitter,

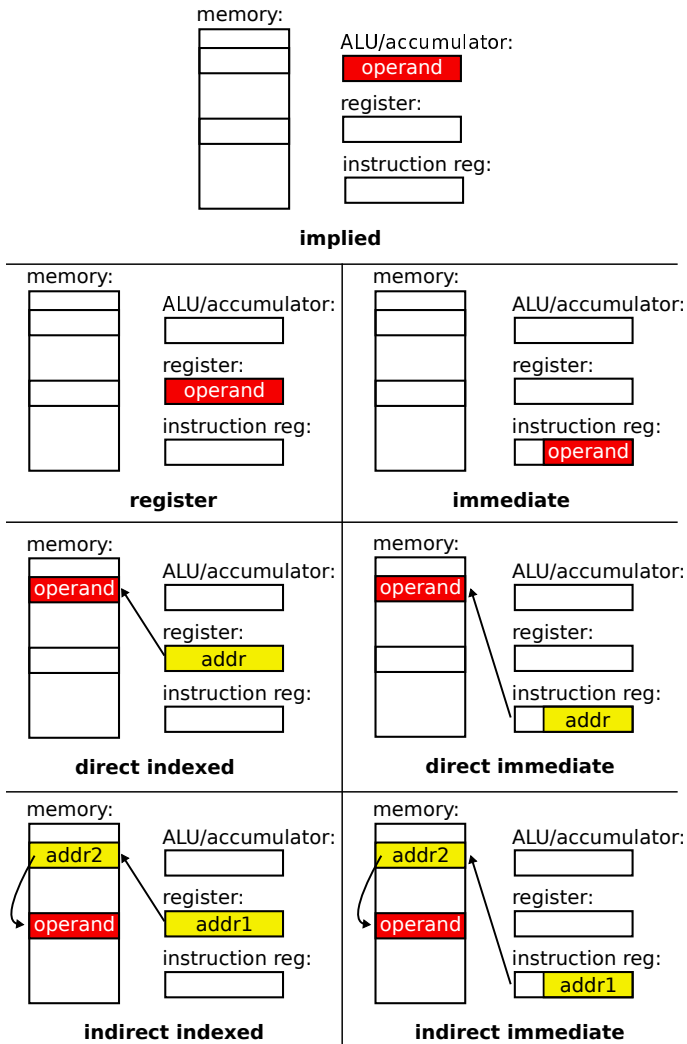


Figure 83: Some examples of addressing modes in Assembly. How the operand that will be used in the operation is referenced.

better known as the hardware that takes care of serial communication (COM ports, etc.). Figure 82 shows this more clearly. If the first bits of the address are 0100 (0x4), the memory chip is deselected and the UART chip selected. Then writing in address 0x4000 may be transmitting a character on serial port COM1, reading from 0x4001 receiving from that COM1, writing in 0x4002 transmitting through serial port COM2, etc. Other addresses may

be used to set up the communication ports (bit rate, parity, stop bits, etc.) This system is called overlay because the UART overlays the memory, as it were. The overlay hardware can be switched on and off by the controller, or by writing in a specific memory address. When switched off, the underlying memory becomes visible and accessible again.

7.5.3 Addressing modes

To finish the chapter and come full circle back to information, let's take a look at how a program (or better to say an Assembly program) can access the operands. There are basically five addressing methods:

- Implied: No operand needed. As in the MOS 65xx instruction `INX` (increase the accumulator register by 1).
- Register: The operand is in the register specified. In some architectures this can also be in the accumulator.
- Immediate: The operand is part of the instruction code and is in the instruction register.
- Direct: The operand is in the address specified. This address can be part of the instruction ('direct immediate') or a register specified ('direct indexed').
- Indirect: The operand is in the address that is stored in the address specified (that can again be immediate or indexed).

Some architectures have limited addressing modes, while others, can have very complex instructions. Figure 83 is not exhaustive.

8 | Hardware/software aspects

When a theologian and a philosopher argued about the creation of the universe, the theologian said "The universe is so complex, it must have been God as creator". To which the philosopher answered, "O, and *who* created the complex entity of the Creator?!"

This small anecdote brings us to a pertinent question: How do we start up the computer? Where does it get its program, or first program from? After we have a program in memory, this program can be something that loads other programs, but how do we start it all, the 'divine spark'? What happens when we press the power button and switch on the computer? How can it be that after a while we have a screen with an operating system running? (And in Windows that while can be a little longer compared to Linux). The RAM that stores the program, as we have seen in the previous chapter, consists of flip-flops and dynamic RAMs, either losing all information when the power is removed. That means, when it is switched on it is all filled with 0s (or maybe 1s), maybe all instructions being `addi $0, $0, $0` with the program counter pointing at address 0.

The computer needs a way to pull *itself* out of a zero/reset state. This procedure we call bootstrapping, named after the story of Baron von Münchhausen who pulled himself and his horse out of the swamp he was sinking into by the straps of his boots.

The way a computer does this is by having a minimal program (maybe some kB only) in unerasable read-only memory (ROM). This code has as only task loading the real code into memory. It is so tiny that it cannot even do this simple task. What it is doing is actually loading the small program into memory that then loads the operating system. This small program resides on a hard disk or maybe a USB pen and is called a boot loader. This is a very flexible way of organizing the software that will run on the computer, specifically the operating system because changes can be easily made to the operating system. Bugs can be fixed, functionality (and new

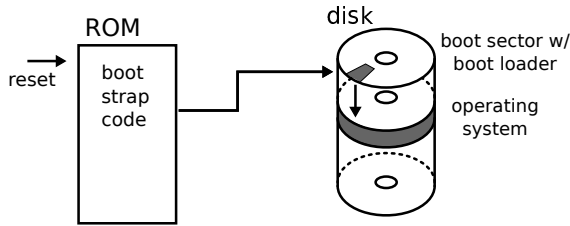


Figure 84: Bootstrap code in ROM is executed that has as only task to load the bootloader from disk. The bootloader in its turn loads the operating system. The operating system loads the user programs to be executed.

bugs) can be added. The only requisite is that the bootloader resides on a specific place on disk, the boot sector. See Figure 84. Once the operating system is loaded, it takes care of loading the user programs.

The bootstrap code in the computer is part of the BIOS ROM (basic input/output system), a special chip with the most elementary operating code. It contains all the system calls like input and output (screen, mouse, disk, keyboard, timer). Also sometimes called firmware (part of the operating system that is non alterable, or not easily alterable). Modern computers have an UEFI (Unified Extensible Firmware Interface) instead of a BIOS, which has higher security and some added features, such as support for larger disk sizes (UEFI is not limited by the 2.2 terabytes of BIOS as it can support 9.5 zetabytes).

8.1 Interrupts

The BIOS also takes care of hardware interrupts. These are electrical signals to the processor that an important event occurred that needs immediate attention. The processor interrupts the current task – hence the name ‘interrupt’ – so that the event can be handled in a timely manner. It deals with the interrupt at an IRQ (interrupt request). The CPU saves the state it is in (program counter, register values, etc.) and starts executing the code of the interrupt handler by setting the program counter to the appropriate interrupt service routine (ISR). In more modern computers, besides processor interrupts generated inside the processor, and hardware interrupts with an electric signal on a pin of the CPU, there also exist software ‘interrupts’, which are called ‘exceptions’ and are treated by event handlers. The idea is the same, the normal flow of the program is interrupted by something that requires attention. The starting addresses of the ISRs are written in an interrupt vector table, or interrupt descriptor table in BIOS. It specifies for every type (value) of interrupt – both hardware and software – where in

memory the interrupt handler routine resides.

The most basic interrupts are processor interrupts, those that are generated by the processor itself. An example is division-by-zero, a state that has to be resolved or the program will not run correctly. Other such interrupts are breakpoints and processor single-step. (See Appendix E for an overview of such processor interrupts in the Intel x86 architecture).

The Intel x86 architecture also has hardware interrupts that are generated by external hardware components such as disks and keyboards (pin 18 of the chip shown in Figure 108). They occur, for instance, when a key is pressed or a mouse is moved. As an example, for Intel x86 architecture the following hardware interrupt requests (IRQs) exist:

IRQ 0	System timer	IRQ 8	Clock
IRQ 1	Keyboard controller	IRQ 9	ACPI (advanced)
IRQ 2	Cascaded from IRQ 8-15	IRQ 10	Peripherals (SCSI, etc.)
IRQ 3	Serial port 2	IRQ 11	Peripherals (SCSI, etc.)
IRQ 4	Serial port 1	IRQ 12	Mouse
IRQ 5	Parallel port or sound card	IRQ 13	CPU float
IRQ 6	Floppy disk	IRQ 14	ATA (HD/CD)
IRQ 7	Parallel port	IRQ 15	Secondary ATA

(ATA, SCSI, ACPI explained later on).

Apart from these hardware interrupts generated by external hardware or by the processor itself, there also exists so-called software interrupts, also known as 'exceptions'. A program can 'throw' an exception. One can imagine operating system function calls, as in for instance writing to screen. The distinction between interrupts, exceptions and system calls in some cases quite vague, but one can imagine an hierarchy, a priority of importance of the system to deal with the interruptions.

A common problem is, what to do when an interrupt interrupts another interrupt? Even though interrupts have priority (a lower-priority interrupt cannot interrupt a higher-priority one) the system can freeze up. It is a quite complicated task that in x86 architecture is managed by a specially dedicated chip, PIC (programmable interrupt controller), integrated on the motherboard.

Many exceptions (especially the software ones) can be left unattended, but some of them must be dealt with. We call these 'trapped exceptions'. A trap handler must take care of it. You will notice it when you leave out a trap handler in your higher-level language program. The compiler will refuse to compile your program. In earlier days it would compile and then when the trap event occurred the program would terminate and control returned to the operating system. An example is a division-by-zero error, or null-pointer assignment (trying to write in address 0x0000...0), or I/O errors

(file not found or not allowed to write to), or a stack overflow. Less severe are range-check errors (like the addition of two integers that does not fit in an integer).

In summary, many events can cause interruptions of normal program flow. The simplest (hardware) ones are that data from the disk are ready or a keystroke occurred or a mouse moved. Some events are more critical than others. Some are hardware triggered with directly an electric signal to a pin of the CPU, others are software triggered. In normal conditions, flow of the program is interrupted by the PIC or CPU, meaning the state of the CPU is saved (program counter, registers, etc.) and the event dealt with. After it has been dealt with, the interrupted program continues.

Interrupts can also be triggered by software. See Appendix F for a small list of x86 architecture software interrupts. An interesting one is MS-DOS interrupt 0x21 (or 21h in x86 jargon). This interrupt is actually a set of interrupts, the one we want is specified by the AH register. It mostly deals with I/O functions, but also includes a way to change any interrupt to make it point to our own interrupt handler. By placing 0x25 in register AH, the interrupt we want to change in AL and the address pointer (segment and offset) of our handler routine in registers DS:DX and then triggering an interrupt 21h we will set the entry AL in the interrupt vector to our interrupt handler routine. We could also directly 'poke' this into the interrupt vector table, the address of AL-interrupt routine is given by `offset = 4*intnum`, `segment = 4*intnum+2`.

The idea of interrupts is very useful. The alternative to interrupts is what is called 'polling'. In that case, the CPU itself is regularly checking if there is nothing important going on. The advantage is that the hardware and organization are much simpler and it is more flexible. The disadvantage is that the CPU wastes a lot of time on checking things that do not need attention. With interrupts, CPU time is never wasted. Events will only be processed when they occur, which maybe once in a million years, or maybe even never.

8.2 Bus

We have already seen it mentioned in the section on computers how a bus connects various devices over the same set of physical tracks. A bus is a set of communication lines between various components or parts of the computer or parts inside the CPU. Typically, a bus connects a CPU with the external memory, with a graphics card, a hard disk, and external components like USB pens. Because all devices connected to it can put data on the bus the bus is multi-directional. Imagine a CPU writing to a USB pen and later the data from the USB pen is copied to memory. That means the devices are connected to the bus by tri-state (see Chapter 4). Theoretically it would also

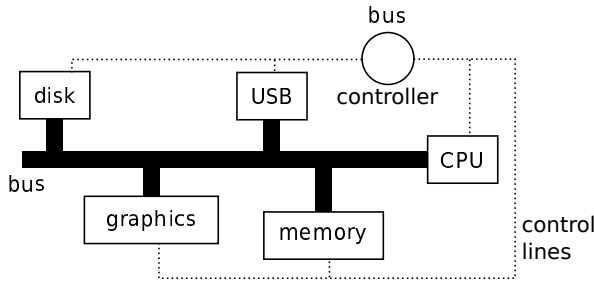


Figure 85: A simple diagram of a bus.

be possible to do with multiplexers but that is too complex a technology. The logical layout of the bus looks like something in Figure 85. Many devices connected to it. A bus controller (not necessarily the CPU) manages it.

The bus controller determines which device is allowed to ‘talk’ and has low-resistance output by asserting its OE (output enable) line. All the others do either nothing or are listeners (when their chip select lines CS are asserted), in any case they have high resistance tri-state at the output. There can be many listeners, but only one talker. In normal circumstances there is one talker and one listener.

Intel x86 and AMD 64 architectures even have a hierarchy of buses (see Figure 86). Well-known bus architectures are VME (Versa Module Europa), ISA (Industry Standard Architecture), EISA (Extended ISA; for 386), Microchannel, PCI (Peripheral Component Interconnect), SCSI (‘scuzzy’, Small Computer System Interface), ATA-PI (Advanced Technology Attachment - Packet Interface), SATA (Serial ATA), PATA (Parallel ATA, better known as IDE [Integrated Drive Electronics]), IEEE487 (Institute of Electrical and Electronics Engineers, standard 487), PCI-express, AGP (Accelerated Graphics Port). They differ in speed and control command language. A motherboard can even have more than one of those buses. To connect them to the main bus, a modern motherboard has two controllers, called North Bridge and South Bridge. The Northbridge (or memory controller hub) connects to the CPU through a front-side bus (FSB) and to RAM and through a fast PCI-Express bus to advanced graphics cards. It also connects to a Southbridge (or I/O controller hub) that in turn connects to slower components such as PCI buses, USB interfaces, ISA buses, IDE hard disks, the BIOS, Ethernet connections and legacy devices.

The CPU itself also has internal buses. The FSB mentioned above is the interface with the outside world and connects internally to the bus interface and level 2 cache. This interface connects to the inside of the CPU, namely the CPU core (with ALU and registers) and level-1 cache, through a back-side bus (BSB).

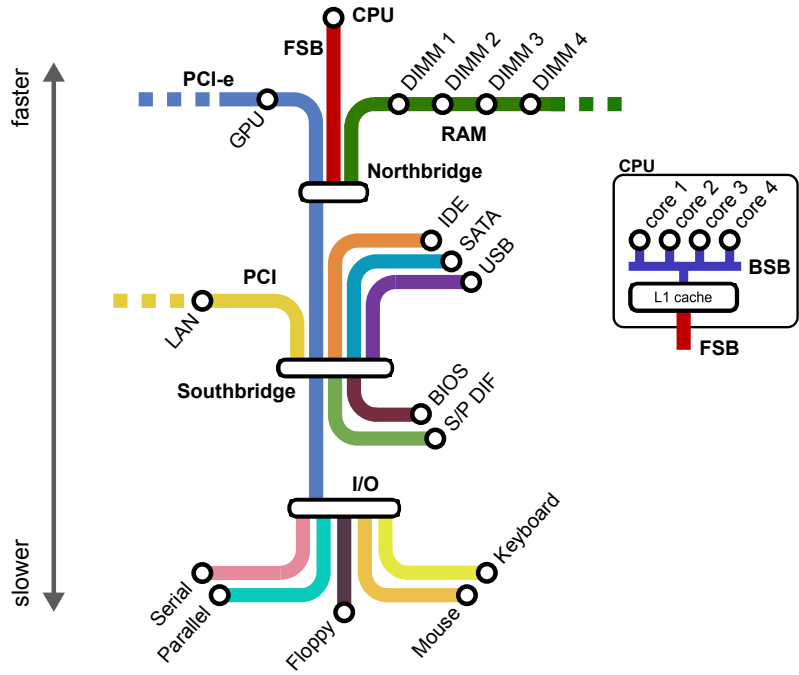


Figure 86: A diagram of an Intel-x86/AMD64 motherboard bus hierarchy.

Further down the hierarchy we can find drivers for external equipment such as USB hubs and Internet interfaces (Ethernet and WiFi). The distances are getting greater and the devices connected (and bus technology) thus slower. The SATA hard disks are relatively slow (compared to memory and advanced graphics such as PCI-e and AGP), and even further down the hierarchy we find the very slow BIOS, legacy ports (such as serial and parallel ports, floppy disks and (non-USB) mice (not 'mice', by the way) and keyboards.

8.3 Communication

From there devices can be connected off-board and distances get larger and speeds lower (remember Einstein's speed-of-light limitation). Moreover, simple metal track on plastic PCB (printed circuit board) no longer works. Typical distances become 1 meter for USB to 100 meters for Ethernet cables. Such cables, moreover being external, work like antennas and catch a lot of interference and noise. In fact, given here without proof, the Shannon-Hartley channel capacity of the communication channel (in bits-per second)



Figure 87: Twisted pair. The same signal is sent over both wires, but with opposing current polarity. Because the noise has the same polarity it can be filtered out by a differential amplifier. (Image: Wikipedia).

is given by

$$C = \Delta f \log_2(1 + S/N)$$

with Δf the physical bandwidth of the channel in hertz, S the signal strength (watt), N the noise level (watt) and S/N the signal-to-noise ratio. To increase the communication speed, we can thus increase the channel width (if it does not start overlapping with other channels), increase the signal strength (if people do not start getting paranoid about radiation and we have enough money and power available) or reduce the noise. To overcome the problem of noise, two things are normally done. One hardware and one software:

Cables come in pairs, so called twisted pairs. One wire carries a (current) signal ($I_1 = +S$), while the other carries the same signal, but with an opposite sign ($I_2 = -S$). Because wires are close together, they catch the same noise δN . A differential amplifier, A , then amplifies the signal and eliminates the noise: $A \times (I_1 - I_2) = A \times ([+S + \delta N] - [-S + \delta N]) = 2AS$. As an example, a standard USB cable has one twisted pair (for bidirectional data), a +5 V power supply, plus ground, giving four wires in total. Ethernet (RJ45) is similar. It has more wires but only one twisted pair for data called H and L.

The other way to overcome the problem of noise is by software. The technique is to add redundant information to the data packages. This redundant information is fully predictable when we know the other bits of the package and thus have no entropy (see the section on information and entropy, Section 7.1). Yet, if something goes wrong in the communication, and bits arrive wrong, the other side can detect and sometimes even correct errors.

In a general cyclic redundancy check, CRC, n bits of redundancy are added. Similar algorithms, like MD5 add redundancy in a cryptographic way to protect the data from intentional alterations rather than protect it from unintentional errors as in CRC. The simplest form of CRC is parity check. Parity checking is thus CRC1. For every package of bits an extra (redundant) bit is added to make the total number of 1s even. It contains no new information, but is just added as a check. A 1 is added if the other data bits have an odd number of 1s and a 0 is added when the number of 1s was already even. Example of such an E81 (even parity, 8 data bits, one stop bit) serial communication string is

01110110 1 0

The first 8 bits are the data, the next 1 is the parity bit (a 1 to make the number of 1s even, namely 6) and the 0 is the stop bit (used for synchronization). Now if something goes wrong in the communication, it can be detected. We do not know *what* went wrong, but know that *something* is wrong if we receive a package

01100110 1 0

The parity does not check. The receiver obviously does not know the original package. (If the package was known to the receiver, it would contain zero information, because the probability of the [intended] message being that package is 100%; why send a message the other side already knows?!) But something is wrong because the parity does not check (five 1s, which is odd, if not to say strange). The receiver cannot recover the data. It does not know *which* bit is wrong (may even be the parity bit!) The only way to proceed is to ask the sender to resend the message. This is only possible in bidirectional communication channels, and can be rather slow, especially when we are in a stream of data (the first bit not yet arrived and the n^{th} bit already sent). But what about one-to-many unidirectional channels, such as radio and television broadcasts or compact disks? moreover, the above scheme can detect *single* errors. What if there were two errors – two bit flips – in communications?

01101110 1 0

is accepted; after all, the parity bit checks (six 1s in total).

To overcome this we can use something that is widely used in unidirectional communications and is actually able to not only detect errors but also correct them. It is called 7.4 Hamming and is a form of forward error correction (FEC). It works as follows: Apart from four data bits, three redundant parity bits (without information) are sent. To determine what the three parity bits are we use a schematic shown in Figure 88. We draw three circles and place the data bits d1..d4 in the four intersection zones. Then we find the parity bits p1..p3 by making all circles have an even number of 1s. The example shows how to the data pattern 1000 the parity pattern 110 is added. Now if we receive a pattern we find the intersection of all circles where the parity is wrong and this is the bit that must have flipped during communication. An example, if we receive 1001110, we know the error must have been the 4th data bit.

This scheme can correct single-bit transmission errors 100% successfully. Multiple-bit errors can still go undetected and uncorrectable. While the Hamming 7.4 and similar FEC techniques are widely used in communication, modern computers also start using it for internal communication, for instance on the buses described above. The reason is that then the clock frequency can be increased, because it permits a small tolerance on errors because a small bit-error rate (BER) will not be fatal. Shannon and Khinchin

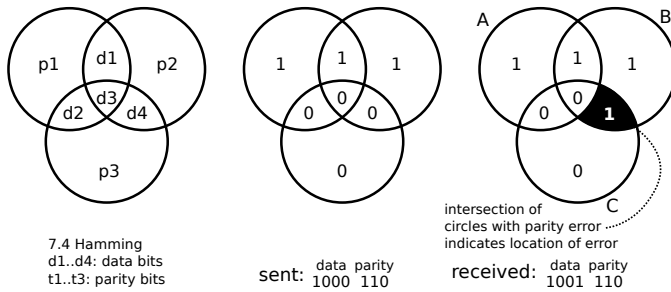


Figure 88: 7.4 Hamming coding (forward error correction, FEC). To the four bits of data, d1..d4, are added three parity bits, p1..p3, so that the parity is even in all circles. When a message is received it can be checked on errors and single errors can even be repaired; the error is the bit in the intersection of all circles with an odd number of 1s. Here indicated a data bit d4 was transmitted wrongly.

have proven that theoretically an errorless channel of communication can be established using adequate error-correction techniques and has a channel capacity given before. I happily refer the reader to works on telecommunications.

9 | Architecture of MIPS

We now come to a specific architecture, namely that 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. Figure 89 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 sophisticated 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 at 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 to mathematically add two operands.
- 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 that we use the naming `$rs` for the source and `$rt` the target register. For instance we can add the source register `$t0` to the target register `$t1`.
- One of the operands can also be 'immediate', which means that the source value is part of the instruction and should be copied from the immediate-field of the instruction register.

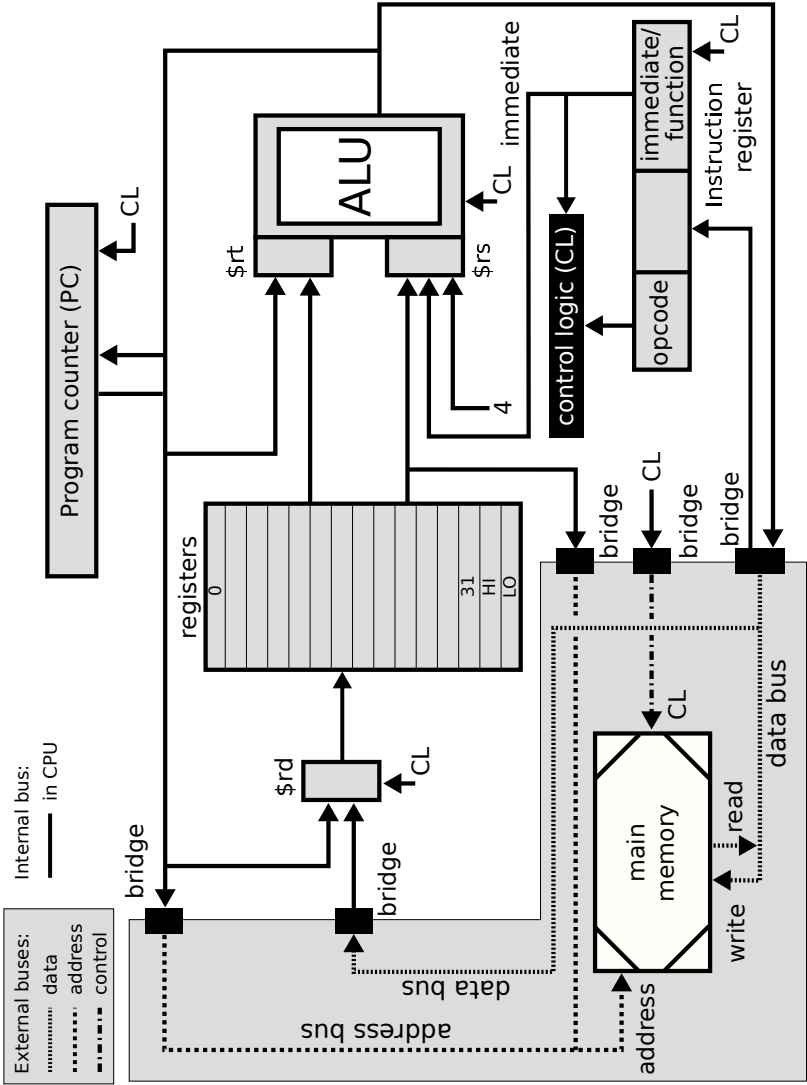


Figure 89: Basic data flow diagram of the MIPS architecture.

- 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 conditional or unconditional jump of the program, making the program counter point at another instruction. If the program counter was not changed by such a jump instruction, by default the program counter is increased by 4 (see the '4' source value

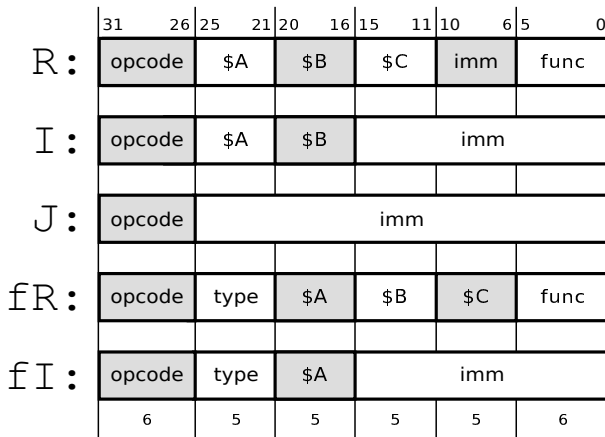


Figure 90: Possible formats of the MIPS instructions. Three different types, R, J and I. All instructions start with a 6-bit opcode, R (register) instructions specify a source (\$rs), a target (\$rt) and a destination (\$rd) register. Some further specify a number (num), for instance the amount of bits 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.

at the ALU in the Figure).

- 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 in main memory instead or the source operand is loaded from memory and stored in the destination register.

See Figure 90 for the possible formats of the instructions.

Before we can start programming the MIPS architecture, we also need first to describe the register file here in the end of this chapter. 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 XXXII):

- 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

Table XXXII: MIPS registers

Integer 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

Floating-point registers:

\$f0	\$ft0	\$f8	\$fs0	\$f16	\$fa6	\$f24	\$fs8
\$f1	\$ft1	\$f9	\$fs1	\$f17	\$fa7	\$f25	\$fs9
\$f2	\$ft2	\$f10	\$fa0	\$f18	\$fs2	\$f26	\$fs10
\$f3	\$ft3	\$f11	\$fa1	\$f19	\$fs3	\$f27	\$fs11
\$f4	\$ft4	\$f12	\$fa2	\$f20	\$fs4	\$f28	\$ft8
\$f5	\$ft5	\$f13	\$fa3	\$f21	\$fs5	\$f29	\$ft9
\$f6	\$ft6	\$f14	\$fa4	\$f22	\$fs6	\$f30	\$ft10
\$f7	\$ft7	\$f15	\$fa5	\$f23	\$fs7	\$f31	\$ft11

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
```

is implemented with (for instance)

```
addu $rd, $zero, $rs
```

Fortunately, MIPS compilers understand `move` and we can freely use this instruction. We have to realize that `move` is not implemented in MIPS hardware, 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`, as we will see in the next chapter.
- 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 (Section 10.8 of Chapter 10).
- 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 10.8 of Chapter 10. The kernel registers, global pointer and frame pointer will not be discussed in this book.
- Apart from these general 32-bit registers there are two registers exclusively used for arithmetic, namely HI and LO that are each 32 bit wide and store the results of integer multiplications and divisions.
- For floating-point operations there exist 32 32-bit floating-point registers labeled simply from `$f0` to `$f31`, or once again divided in temporary registers (`$ft`), saved registers (`$fs`), or arguments (`$fa`).

The question of MIPS being Little Endian or Big Endian we leave unanswered here. For us it can be 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 a problem might arise when we communicate between computers of different architectures.



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 powerful instructions, but each one 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 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.

What does strike us here is the total absence of programming concepts familiar to us in high level programming languages such as C. Things such as variables, arrays, structures, function calls, I/O, for-loops, etc. All these concepts will be built *on top* of the layer of Assembly and the architecture, but are not part of it. In the next chapter we will actually implement all these concepts in Assembly and will that way naturally migrate into the higher programming level, especially C (which is the programming language closest to Assembly). An example is variables. There do not exist different variable types (`int`, `float`, etc.). That is a concept of high-level programming. The only thing we will have in Assembly is 32-bit CPU registers and memory addresses and their contents. Concerning the latter, everything we have to the memory are addresses ('pointers'). We could say that everything is pointers. Always. As we will see, our Assembly code will have to do all the bookkeeping, keeping track of what information resides where. After having learned this level, you will much more appreciate the higher programming level. It saves a lot of time.

In summary, what Assembly consists of is

- Arithmetic (addition, multiplication, division, masking, shifting)
- If-then (branching)
- Copy (from memory to registers and back, and between registers)
- Pointers (memory addresses)
- Data (integers, floats, and pointers)

Note that pointers can also be data and we can do arithmetic with them, as if they were simple integers. In fact, there is no physical difference between 'normal' data (integers and floats) and pointers. A pointer is merely a 32-bit pattern *interpreted* as an address.

It is now time to take a look at a specific MIPS assembler. We do that in the next chapter.

10 | MARS: MIPS Assembly language implementation

It is now finally time to start coding! In this chapter we will do that. We will learn here how to write in Assembly. We already knew the first-generation processing (the electronics and logic gates), as well as having an idea of the second-generation level (machine language). Moreover, we also know the fourth-generation (imperative programming like C and Pascal). It is now time to bridge the final gap by the third-generation level, namely assembly, and we use a specific architecture with that, namely MIPS with MIPS Assembly.

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 addresses, pointed at 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 H). That means that specifying which one to use takes 6 bits ($2^6 = 64$). 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'. Note, however, that some instructions use additional bits to define a sub-function, thereby leaving less space for the rest of the instruction. An example is the opcode 000000 that is used for a variety of mnemonics, as can be seen in Appendix H.

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. Note that MIPS therefore does not have indirect addressing mode (the operand in an address that is in an address pointed at by a register, see Figure 83).

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 (pointers).

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

```
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. The labels are forgotten when executing the 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,

```
mydef .eqv 64
```

It just remembers the combination label and text "64" in the table. Every time the label `mydef` is encountered in the code, it is substituted by the text "64" and then interpreted by the assembler. This text can be anything you want, as long as it is legal MIPS code.

In this case, after our three declarations the compiler label-look-up table now looks like this, two pointers to memory (one to data and one to code) and one text value:

<i>Label</i>	<i>Value</i>
<code>myword</code>	0x10010000
<code>mycode</code>	0x00400000
<code>mydef</code>	ASCII "64"

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 mydef`, which the compiler might even accept if the value coincidentally is within the code segment 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).

In the next sections we will see how MIPS can implement basic concepts of higher 'imperative' programming languages such as C. We will see

- Input/output
- Arithmetic
- Jump and branch (`goto` and `if-then`)

Table XXXIII: Some compiler directives

Anywhere	
#	ignore rest of line (comment)
<i>example:</i> # this text will be ignored	
.eqv label text	store (label,text) in compiler table
<i>example:</i> .eqv four 4	
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
<i>example:</i> numbers: .word 1, 2, 3	
<i>example:</i> numbers: .word 0:129 # reserve 130*4 bytes space	
label: .ascii "string"	store ASCII string in data segment and store (label,address) in compiler table
<i>example:</i> name: .ascii "Peter"	
label: .asciiz "string"	same as above but add NULL (0x00, EOS) to end of string
label: .space n	reserve <i>n</i> bytes in memory and store (label,address) in compiler table
<i>example:</i> myarray: .space 200	
Code segment	
label:	store (label,next code address) in compiler table
<i>example:</i> main:	

- Loops (for, while, do-while)
- Arrays and structures
- Floating point arithmetic
- Functions

10.1 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 (think of calculating the first 1000 digits of π), 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 resemble the software interrupts (exceptions) that were described in Chapter 8 and in MIPS are called system calls which are summarized in Appendix J. The method consists of choosing the system call we want by placing the correct number of the system call in register `$v0`, place any arguments in the a-registers, and then issue a `syscall`. As Appendix J shows, printing a string is MARS system call number 4 and we have to place the address of the null-terminated string in register `$a0`. That's all there is to it. The program below, `hellow.asm`, shows how this works. First we 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 stops execution. Figure 91 shows how this looks in the MARS environment.

```
#####
#   MIPS Assembly 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 --
```

Analyzing the program we see that assembly does not have any variables. The only thing assembly has is values – bit patterns – and these can also be

interpreted as addresses (pointers). When we 'declare' a 'variable' – label – such as `hellow`, what, in fact, is done is

- Space is reserved in the data segment (that will be residing in main memory on the heap at runtime) 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 corresponding value.

In this case the label-look-up table looks like this (see panel Labels in Figure 91. Note that any `.eqv` definitions are now shown in this look-up table):

<i>Label</i>	<i>Value</i>
<code>start</code>	<code>0x00400000</code>
<code>stop</code>	<code>0x00400010</code>
<code>hellow</code>	<code>0x10010000</code>

The first two labels are pointers to instructions in the code segment. The last is a pointer to the string in the data segment. 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):

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

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, for example adding the source register to the `$zero` register and putting the result in the destination register.

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, $t0, immediate`
store the 32 bit value `immediate` in register `$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

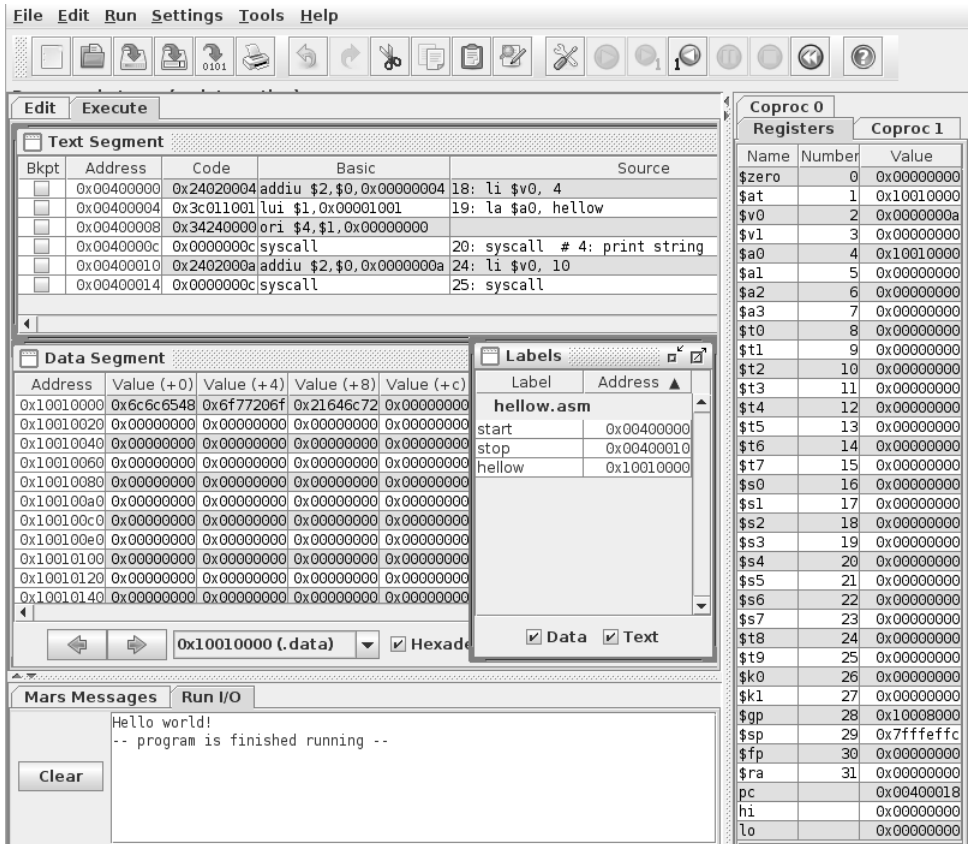


Figure 91: Example of the compiled and ran program hellow.asm.

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
```

thus translates into the two instructions

```
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' and the value is now an address.

The instructions also 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, and they have two addressing modes, immediate and direct.

- `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 Figure 91. We see that the compiler has translated our

```
li $v0, 4
into
```



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

which accomplishes the desired effect, adding 4 to the 0 of register `$zero` 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, $0, 0x00000004`. (See the Figure ; the column named 'Basic'). If we look in the appendix with MIPS instructions (Appendix H), we see that `addiu` has an opcode equal to `001001`, after which follow five bits for the source register, five bits for the target register and 16 bits for the immediate value: the source register is `$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 `00000000000000100`, so the final instruction is:

mnemonic	source	target	immediate
<code>addiu</code>	<code>\$zero</code>	<code>\$v0</code>	4
<code>001001</code>	<code>00000</code>	<code>00010</code>	<code>00000000000000100</code>

Regrouping in units of 4 bits and converting to hexadecimal:

	<code>0010</code>	<code>0100</code>	<code>0000</code>	<code>0010</code>	<code>0000</code>	<code>0000</code>	<code>0000</code>	<code>0100</code>
0x	2	4	0	2	0	0	0	4

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)

<code>6c</code>	<code>6c</code>	<code>65</code>	<code>48</code>	<code>6f</code>	<code>77</code>	<code>20</code>	<code>6f</code>	<code>21</code>	<code>64</code>	<code>6c</code>	<code>72</code>	<code>00</code>
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.

In this example we used console output, printing a string to the screen by `syscall 4`. We can also output (and input) to (from) file. For that we use I/O system calls (see Appendix J) `syscall 13` (open file), `syscall 14` (read from file), `syscall 15` (write to file) and `syscall 16` (close file). An example of writing `Hello world!` to a file named `hello.txt`:

```
#####
#   MIPS Assembly program that writes "Hello world!"   #
#       to a file 'hello.txt'                         #
#####

.data

filename: .asciiz "hello.txt"
outbuffer: .asciiz "Hello world!\n"

.text

    la $a0, filename    # string with filename
    li $a1, 1           # mode: 1 = overwrite
    li $a2, 0           # mode2: 0 (always)
    li $v0, 13          # syscall 13 = open file
    syscall

    move $a0, $v0        # file descriptor
    la $a1, outbuffer    # address of string to write
    li $a2, 13           # number of chars to write
    li $v0, 15           # syscall 15 = write to file
    syscall

    li $v0, 16          # syscall 16 = close file
    syscall

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

10.2 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 L0, 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? Or to have an immediate output operand?) 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 version (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	4	0x00000002	2
	0000...0100		0000...0010	
sra 1	0x00000004	4	0x00000002	2
	0000...0100		0000...0010	
sll 1	0x00000004	4	0x00000008	8
	0000...0100		0000...1000	
srl 1	0xffffffffc	-4	0x7fffffff	2147483646
	1111...1100		0111...1110	
sra 1	0xffffffffc	-4	0xffffffff	-2
	1111...1100		1111...1110	
sll 1	0xffffffffc	-4	0xffffffff8	-8
	1111...1100		1111...1000	

Shift instructions, where the number of bits to shift is not an immediate value contained in the instruction, but given in a register instead, are specified by adding a 'v' to the mnemonic, which indicates 'value'. 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, as in a rotation. Such rotate instructions are not part of MIPS, but our MARS compiler can translate the pseudo-code easily

mnemonic	opcode	function	description
rol \$rd, \$rt, n	-	-	rotate \$rt left <i>n</i> bits
ror \$rd, \$rt, n	-	-	rotate \$rt right <i>n</i> nits

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

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

which indeed does the job.

exercise: Arithmetic

1. Write a MIPS program that asks two numbers and prints their sum, difference, product and quotient.
2. Write a MIPS program that asks two numbers and calculates and multiplies them, with this multiplication not done by a `mult` instruction, but by shifting and summing. Make it also work for negative numbers.

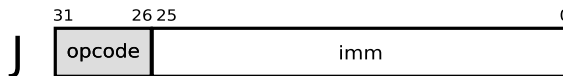
10.3 Jump and branch; (goto, if ... then goto)

The next instructions we will look at are simple unconditional jumps and conditional jumps (see Figure 92). 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 → pc`



Since the program counter (`$pc`) is like any other 32-bit register, the equivalent of a jump instruction is `la $pc, address`. Note also that the jump address is 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 specifying the address (of 32 bits) and there thus exists a problem.

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 in immediate jumps are thus relative to the program counter (`pc`) when jumping. To be more precise, the address jumped to is on the 'page' specified by the first 4 bits of the actual program counter (`page = pc AND 0xF0000000`, coding for 16 pages); the final address to jump to is this page plus the 26 immediate bits (unsigned integer) of the instruction multiplied by 4 (left-shifted two bits):

$$\underline{\text{page} + 4 * \text{immediate}(26) \rightarrow \text{pc}}$$

or (binary)

$$\underline{\text{pppp} \text{iiiiiiiiiiiiiiiiiiiiiiiiiiiiii} 00 \rightarrow \text{pc}}$$

with `pppp` the first 4 bits of the program counter and `ii...i` the 26-bit immediate value. As an example, if we have a jump instruction at address `0x00400038`

`j 0x0040002c`

the address to jump to is `0x0040002c = 0000 0000 0100 0000 0000 0000 0010 1100`, the page (of both the jump instruction and the one to jump

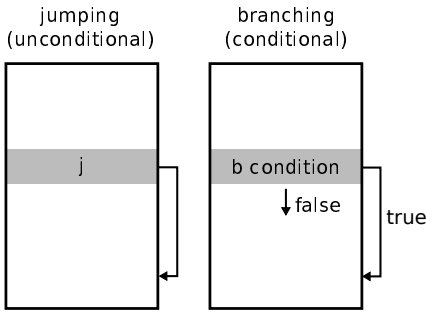


Figure 92: Difference between (unconditional) jumping and (conditional) branching.

to) is the first 4 bits: `pc-page = 0000`. The rest of the destination address is divided by four and serves as the immediate value of the operation: `immediate = 00 0001 0000 0000 0000 0000 1011`. The jump opcode = `0000 10`. The total instruction at address `0x00400038` is thus

opcode		immediate						
0000	10 00	0001	0000	0000	0000	0000	0000	1011
0x	0	8	1	0	0	0	0	c

This way, the first bits of the program counter represent something like a page we are working on, of which in MIPS there exist $2^4 = 16$, 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 a full 32-bit address in a 32-bit register and issuing a jump-to-register (`jr`).

The advantage of paging (or addresses relative to the program counter) is that a program can be relocated in memory without having to change the code. The most convenient therefore would be jumps relative to the program counter, in which case the program can be placed anywhere in memory, without restrictions and recalculating instructions.

*

Conditional jumping is called 'branching', for which we have a set of instructions that are generally described by:

- **branch-condition address**
This puts the value of the address into the program counter when the condition, comparing two values, is true, otherwise it defaults to adding 4 to the program counter, like for normal instructions:
condition is true: `label → pc`
condition is false: `pc+4 → pc`

It is thus equivalent to the BASIC instruction

IF condition=TRUE THEN GOTO label

The six branch instructions that are implemented in hardware are:

```

beq $rt, $rs, addr : Branch to addr if operands are equal
bne $rt, $rs, addr : Branch to addr if operands are not equal
bltz $rt, addr     : Branch to addr if operand < 0
bgtz $rt, addr     : Branch to addr if operand > 0
blez $rt, addr     : Branch to addr if operand ≤ 0
bgez $rt, addr     : Branch to addr if operand ≥ 0

```

All other branching variants are pseudo instructions that might (or might not) be implemented by the assembler, or can be implemented by us in macros (see Appendix H):

```

blt $rt, $rs, addr : Branch to addr if $rt < $rs
bgt $rt, $rs, addr : Branch to addr if $rt > $rs
ble $rt, $rs, addr : Branch to addr if $rt ≤ $rs
bge $rt, $rs, addr : Branch to addr if $rt ≥ $rs
beqz $rt, addr     : Branch to addr if $rt = 0
bnez $rt, addr     : Branch to addr if $rt ≠ 0

```

As an example,

```
beqz $t0: Branch if $t0 = 0
```

is implemented as

```
beq $t0, $zero
```

which obviously does the trick. And

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

is implemented in two instructions as

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

Here the instruction `slt` (set if less than) is writing the result of the evaluation of the condition — 'less-than' — in a specified register

- `slt $rd, $rt, $rs`: Set `$rd` (to 1) if `$rt` is less than `$rs` (reset 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 (`$rd` replaced by an immediate value) apart from the direct addressing mode above, and has versions 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 the condition is true, the program continues at address `(pc+4) + immediate×4`,

with `pc` the address of the branching instruction, and `immediate` a 16-bit signed integer. The address to branch to thus becomes:

$$\overline{(pc+4) + 4*immediate(16) \rightarrow pc}$$

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 direct-addressing-mode 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 assembly an 'else' clause, nor does the concept of multiple choice (like 'switch' in C, or 'case ... of' in Pascal) exist. All conditional branching in Assembly 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 assembly:
    beq $t0, $t1, label
    instructions-when-false
```

They are in opposite order, jump-over-when-false vs. jump-over-when-true.

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 Assembly 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 "Give a number"
li $v0, 5
syscall                # read int into $v0
move $t0, $v0
li $v0, 4
syscall                # print "Give a number"
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 --

```

10.4 Loops; (for, while, do-while)

In high-level programming we normally have the concept of loops to our disposition, 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 is ever 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 Assembly, but it is not very difficult to implement with the branching instructions learned above. The code below gives an example of how to implement a for loop, printing n times a certain text.

Source code:

```
#####
# MIPS Assembly 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", "Hello world!");
# 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 \$1,0x00001001	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: Prime numbers

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

10.5 Masking

A special application of the bitwise `and`, `or` and `xor` instructions is masking. Imagine if we have a bit pattern of 8 flags, for instance the printer status, and we want to see if one status bit is set, for instance if the printer is out of paper. How do we do that? Very simple. We use a bitwise `and` instruction with the flags status register as first operand and for the second operand a value consisting of all zeros, *except* at the place corresponding with the status bit of interest, which is 1 instead. This 'isolates' the bit we are interested in and we can now either directly check if the resulting entire value is zero or not, or we can right-shift the bit under scrutiny to the least-significant bit (LSB) position and compare the result with 1. If this int value is 1, the flag bit was set, if the resulting int is equal to 0 the flag bit was not set. As an example, imagine we have a status register equal to 00010110 and we want to know if bit b_2 (3rd bit from the right) is set we use

00010110

Reading a bit:

00000100 and

00000100 srl 2

00000001

This result is equal to one and thus the bit was set. (Alternatively, we could have stopped after the `and` instruction and decided that because this value is not zero, the bit must have been set). The mask, the second bit pattern above, as it were, lets through only the relevant bit and 'masks' the other bits.

Setting a bit to 0 or 1 can also easily be done, as well as inverting a bit:

Setting bits (OR-1):

00000001 sll 7

10000000

00010110 or

10010110

Erasing bits (AND-0):

11111110 rol 1

11111101

00010110 and

00010100

Toggling bits (XOR-1):

00000001 sll 2

00000100

00010110 xor

00010010

It is obvious that various bits can be addressed like this simultaneously.



On basis of this we can write a program that multiplies two numbers without using the `mult` instruction. Remember that multiplications consist of shift and add, in the so-called Russian-peasant algorithm. An example is shown here multiplying two positive numbers (max 65535, to not have overflow). It saves the result in `$l0`:

```
#####
# MIPS Assembly program to do integer multiplications #
# without using the MUL instruction                    #
# Russian-peasant algorithm                            #
#####

.text

li $t5, -10    # A
li $t6, -5     # B
```

```

    move $t0, $zero # $t0 = result = $t5 * $t6

loop:
    beqz $t6, ready
    and $t1, $t6, 1 # mask to only have LSB of $t6
    beqz $t1, continue # if LSB=0 no adding needed
    addu $t0, $t0, $t5 # add $t5 to result
continue:
    sll $t5, $t5, 1
    srl $t6, $t6, 1
    j loop

ready:
    li $v0, 1 # print int
    move $a0, $t0
    syscall

return:
    li $v0, 10
    syscall

```

Another example, of how to do divisions, is shown here below. In this case, the operand *a* in the division *a/b* is stored in *\$t1* and shifted into *\$t5* by masking (*\$t3*). Every time the result in *\$t5* is bigger than the *b* operand (*\$t2*), an operation is performed, subtracting *\$t2* from *\$t5*, and a 1 is shifted into the result register *\$t6*, otherwise a 0 is shifted into it. The mask bit – starting on the leftmost position (could be 2nd position. Why?) – is then shifted to the right. When the mask bit is shifted out of *\$t3*, the division is ready. The result is in *\$t6*, with the remainder in *\$t5*. For compatibility, these are then copied to the *\$lo* and *\$hi* register, respectively. To make the algorithm work with negative numbers, first the sign information is saved (in *\$t9*), and then the positive values of the operands are used. For inserting 1s into registers, *ori 1* is used rather than *andi 1*, because OR-immediate is a faster operation in many architectures:

```

#####
#   MIPS Assembly program to do integer divisions   #
#   without using the DIV instruction               #
#####

.text
# directly inserting operands here
li $t1, 1000 # operand a
li $t2, -3   # operand b

# convert everything to positive and remember in $t9 the
#   final sign in upcoming result (and in $t8 sign of a):
li $t9, 0 # 0=positive result, 1=negative result
bgtz $t2, posb

```

```

    sub $t2, $zero, $t2 # t2 = -b
    li $t9, 1
posb:
    li $t8, 0
    bgtz $t1, posa
    sub $t1, $zero, $t1 # t1 = -a
    li $t8, 1
    sub $t9, $t8, $t9 #t9=1-t9, if (t9==1) t9=0; else t9=1;
posa:

# initialize registers:
li $t3, 0x80000000 # mask
li $t6, 0          # result
li $t5, 0          # shift-in register of $t1 operand

top:
    beqz $t3, done # if mask is shifted empty, terminate
    sll $t5, $t5, 1 # shift left $t5 1 place; make place for new bit
    sll $t6, $t6, 1 # also in the result
    and $t4, $t1, $t3 # is isolated masked bit set in operand a?
    beqz $t4, insertzero # if not 'insert' 0
insertone: # else insert 1 in shift-in register
    ori $t5, $t5, 1
insertzero: # nothing to do here
    # now $t5 has new bit of $t1 (a) shifted in on the right
    # compare it to $t2 operand b
    bltu $t5, $t2, nosubtraction # if big enough, subtract and insert
                                     # 1 in answer $t6

subtraction:
    subu $t5, $t5, $t2
    ori $t6, $t6, 1
nosubtraction: # else 'insert' 0 in answer $t6
    srl $t3, $t3, 1 # move mask bit 1 place to right
    j top

done:
    beqz $t9, saveandterminate # results should be negative?
    sub $t6, $zero, $t6
    sub $t5, $zero, $t5
saveandterminate:
    mtlo $t6
    mthi $t5
    li $v0, 10
    syscall

```

An engineer must admit it is beautiful. Doing divisions without a division operation. Just like we can do multiplications without a multiplication algorithm. All based on additions and subtractions. And subtractions were just additions with the two's-complement of the second operand (Fig. 48). And additions themselves could be carried out by a ripple-carry adder consisting of a dozen transistors per bit. In the extreme case, a single 1-bit full-adder could do the job. And the full-adder was based on simple Boolean logic we

designed with Karnaugh maps, etc. We have already come a long way since we introduced Boolean logic, but we can still go much further.

10.6 Variables, arrays and structures

Access to main memory we have with MIPS instructions load word (`lw`) and store word (`sw`), where we can copy from registers to memory and back. There also exist versions copying less information at a time (`lb`, `sb`, `lh`, `sh` for bytes and halfwords). With this we implement the concept of variables of imperative programming, a concept that does not exist at the level of assembly:

```
.data

# 'declare' variables a, b and c (each occupying 4 bytes)
#  int a, b, c;
var_a: .space 4
var_b: .space 4
var_c: .space 4

.text

#  C language: c = a + b;
lw $t0, var_a
lw $t1, var_b
add $t2, $t1, $t0
sw $t2, var_c
```

The first part is the declaration of the variables, which means simply reserving space for them and remembering where that space is in memory (the assembler will take care of that by creating items in the label look-up table). This space is in the data segment and the declaration is thus done in the `.data` part of our program. Note that it does not assign a value 0 to the variables. The memory might contain garbage. In fact, *we have to assume it contains garbage!* Otherwise our program will run a hundred times correct, and then one day suddenly produce garbage.

The second part is attributing values to the variables and retrieving their values. (The first time a value is attributed is called initialization, and no variable should be left uninitialized). This is done in the code segment (`.text`) part of our program. A compounded C instruction

```
int a = 0;
```

might thus be a little confusing, since half of the instruction is a declaration and half a value-attribution. Half of it is in the data segment and half in the code segment. As long as we remember this, no harm is done. These are examples of writing shortcuts in C, of which it is famous. Closer to Assembly would be

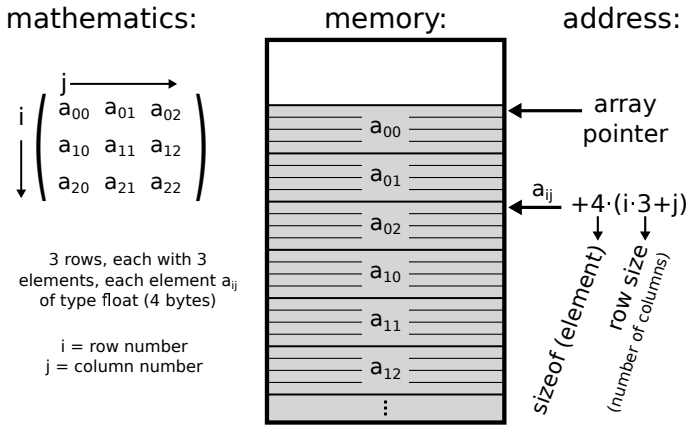


Figure 93: An array in programming is a mapping of a multi-dimensional object, such as a mathematical matrix shown here, to a linear memory 'vector'. We have to calculate the position of the element in memory based on its indexes. The address of element a_{ij} can be found as: $\text{address} = \text{arraypointer} + \text{elementsize} \times (i \times \text{elements_per_row} + j)$.

```
int a;
a = 0;
```

Modern assemblers, like our MARS, can, in fact, do a combination of declaration and attribution as well:

```
var_a: .word 0
```

This reserves space (a 32-bit, 4-byte, word) in memory and fills it with 0.

✱

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

In assembly 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 Figure 93). 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 $\text{vectoraddress} + i \times 4$. A two-dimensional array can be implemented as well, but the calculations are slightly more complicated. The element a_{ij} , with i the row number and j the column number can be found by the row number i multiplied by the row size (number of

columns, dimj) and adding the column number j , multiplying this sum by the element size and adding the result to the base address, see Figure 93. The address of a_{ij} is thus the following C-expression:

```
&a[i,j] = base + (i*dimj+j)*sizeof(element)
```

The example below 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 $x_0, y_0, z_0, x_1, y_1, z_1, x_2 \dots$, starting from address `array`. This means that to find the coordinate x of point i in space, we calculate `array+3×4×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 Assembly program that shows how to implement an array   #
#####

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

# an array of 10 coordinates (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

    li $t8, 999999   # t8 stores minsum

    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 --
```

Assuming the lowest index in any dimension is 0, the address of an element with size `elementsize` for a scalar, a vector, a matrix and a rank-3 tensor all starting at base address `base` can be calculated as follows:

entity	element	address
scalar	a	<code>base</code>
vector	a_i	<code>base + i×elementsize</code>
matrix	a_{ij}	<code>base + (i×dimj+j)×elementsize</code>
tensor	a_{ijk}	<code>base + [(i×dimj+j)×dimk+k]×elementsize</code>
:	:	:

exercise: Array address

In case of a three-dimensional object a of doubles (8 bytes) with dimensions $\text{dimi} \times \text{dimj} \times \text{dimk} = 3 \times 4 \times 5$ starting at address

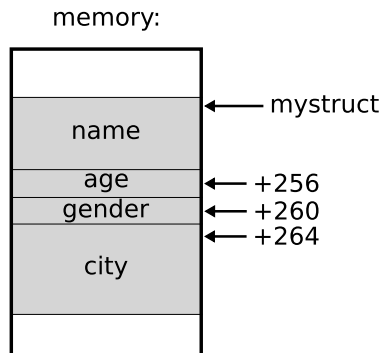


Figure 94: An example of a structure (set of informations of different types) used for the programming example.

0x10010000, what would be the address of element a_{123} ?

✱

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 Figure 94):

Source code:

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

.data
nameprompt: .asciiz "name:"
ageprompt:  .asciiz "age:"
genderprompt: .asciiz "gender:"
cityprompt:  .asciiz "City:"

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

```

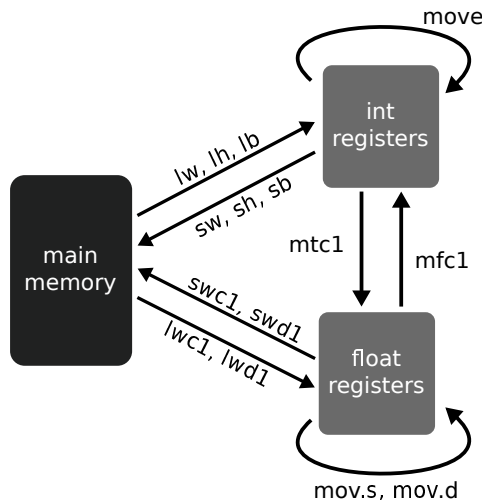


Figure 95: MIPS move instructions to copy from and to main memory, the integer registers and float registers.

10.7 Floating point

In the chapter on computers (Ch. 6) the IEEE 754 standard was introduced for floating point numbers. As discussed, since all information is integer (both the fraction and the exponent), we could implement this with integer operations. However, 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, with the higher-number register containing the most-significant bits. So, for instance, the pair `{ $f4, $f5 }` can store a 64-bit double-precision float with `$f5` containing the sign bit, the eleven exponent bits and the twenty leftmost fraction bits. (The syntax for such instructions only indicate the first of the two registers, which should always be an even-numbered register; `$f0, $f2`, etc.)

MIPS comes with a set of instructions specifically for floating point operations and exchanging bit patterns between the float-registers and regular int-registers. Let's first look at how to move information about, see Figure 95

The first set is main memory access, and this has only direct-addressing mode (no immediate versions), see Appendix H:

- `lwc1 $fd, offset($rt)`: copy 4-byte contents of address pointed to

by `$rt+offset` into register `$fd` of the co-processor.

- `lwd1 $fd, offset($rt)`: copy 8-byte contents of address pointed to by `$rt+offset` into registers `{ $fd, $fd+1 }` of the co-processor.
- `swc1 $fs, offset($rt)`: copy 4-byte contents of register `$fs` of the co-processor into consecutive addresses pointed to by `$rt+offset`.
- `swd1 $fs, offset($rt)`: copy 8-byte contents of registers `{ $fs, $fs+1 }` of the co-processor into consecutive addresses pointed to by `$rt+offset`.

Copying data within the co-processor, like copying in the main processor, is done with move instructions:

- `mov.s $fd, $fs`: Copy contents of single-register `$fs` into single-register `$fd`.
- `mov.d $fd, $fs`: Copy contents of double-registers `{ $fs, $fs+1 }` into double-registers `{ $fd, $fd+1 }`.

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

- `mtc1 $rs, $fd`: (Move to). Copy contents of int-register `$rs` into register `$fd` of the co-processor.
- `mfc1 $rd, $fs`: (Move from). Copy contents of float-register `$fs` into int-register `$rd`.

There are no 'immediate' versions of loading values into floating-point registers. We have to load-immediate the bit pattern into an int-register and copy it from there to the desired float-register, it thus effectively being part of the code segment:

```
.eqv pi 0x40490fdb
.text
li $t0, pi
mtc1 $t0, $f0
```

or have the float in the data segment and load it

```
.data
pi: .float 3.14159265358979323846264338327
.text
la $t0, pi
lwc1 $f0, 0($t0)
```

Data conversion:

- `cvt.T0-TYPE.FROM-TYPE $fd, $fs`: Convert `$fs` from format `FROM-TYPE` to format `T0-TYPE` (either 's', 'd', or 'w') and store the result in `$fd`.
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.

Example:

Source code:

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

.data
newline: .asciiz "\n"

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

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

li $v0, 4
la $a0, newline
syscall            # print new line

# check what the hex code is for the float number:

mfc1 $a0, $f12     # move bit pattern to int processor
li $v0, 34
syscall            # print as hexadecimal

li $v0, 10
syscall
```

Output:

```
9.0
0x41100000
-- program is finished running --
```

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.SIZE $fs, $ft`: Sets condition flag true or false. COND: 'eq', 'lt', or 'le'. SIZE: '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 address:** Jump to **address** if condition flag in co-processor-1 is true. Like for integer -condition branching, only jumps local addresses. Example:

```
bc1t mylabel
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 address: Same as **bc1t**, but branches when condition is false.

Floating-point arithmetic is done by the same mnemonics as was used 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 \$fd, \$ft, \$fs:** Perform arithmetic operation **OP** ('add', 'sub', 'mul' or 'div') of type **TYPE** ('s' or 'd') on **\$fs** and **\$ft** and store the result in **\$fd**. Example:

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

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

That's all there is to floating point in MIPS. 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 (See page 144 of Chapter 6). In this example we calculate the square root. This is a floating point function that is not implemented in hardware, so we have to calculate it with software. This is not so difficult, as will be shown. Calculating the square-root of A consisted of finding the zero of the function $f(x) = x^2 - A$ by successive iterations, every iteration starting from the last approximation and predicting the zero from the function value and derivative at that point. Below here is the entire Newton-Raphson root-calculation in MIPS, with 0.000001 precision.


```
#####
# 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
half: .float 0.5

.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: x_i

lwc1 $f3, half
mov.s $f2, $f0      #x0 = A as seed
dowhile:
    # new xi = (xi + A/xi)/2
    div.s $f5, $f0, $f2 # $f5 = A/xi
    add.s $f5, $f2, $f5 # $f5 = xi+A/xi
    mul.s $f5, $f5, $f3 # $f5 = (xi+A/xi)/2 = new xi
    sub.s $f4, $f2, $f5 # $f4 = new xi - old xi
    abs.s $f4, $f4      # $f4 = |new xi - old xi| = dx
    mov.s $f2, $f5
    c.lt.s $f4, $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: 5.4772253
-- program is finished running --
```

exercise: Float-array calculation

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
```

They are organized according to:

```
x1, y1, z1,
x2, y2, z2,
etc.
```

10.8 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 Figure 96. 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

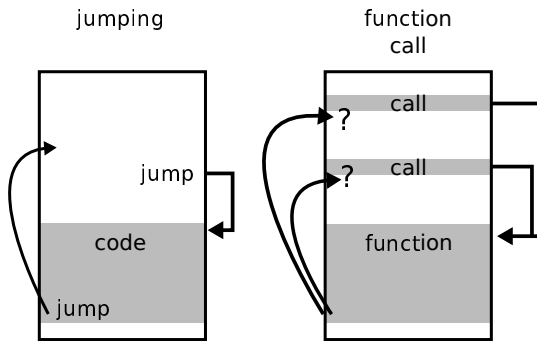


Figure 96: Difference between simple jumping (GOTO) on the left and function calling (GOSUB) on the right. For function calls we have to remember what was the address of the code that was interrupted by the function call, otherwise we do not know where to return to.

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.

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), see Figure 96.

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 Figure 89. 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-to-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 MIPS Assembly, 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* of 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 Assembly 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 Assembly 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 and they

thus have to be saved somewhere, and the stack is an ideal place for that. 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 stack pointer register (\$sp). Pushing and popping is thus implemented as (an example of storing \$rt):

- push:


```
addi $sp, $sp, -4
sw $rt, ($sp)
```
- pop:


```
lw $rt, ($sp)
addi $sp, $sp, 4
```

If we want to save/retrieve multiple items at once, we can do it with things like this, grouping them to save on stack advancing instructions:

```
addi $sp, $sp, -12
sw $t0, 0($sp)
sw $t1, 4($sp)
sw $t2, 8($sp)
```

```

...
lw $t0, 0($sp)
lw $t1, 4($sp)
lw $t2, 8($sp)
addi $sp, $sp, 12

```

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 *must* 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 Assembly 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 multiply        # 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: -----#

multiply:
#####
# 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
multiply_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 Assembly 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 --
```

10.9 Macros (pseudo-instructions)

The instruction set of MIPS is rather limited, as we have seen. This means that some very obvious instructions are not implemented. An example is the copy instruction. Believe it or not, but there does not exist hardware to copy the contents of one register directly to another register. The trick, as we have seen, is to add the zero register (\$zero) containing all zeros to the source register and place the result in the destination register. These things are needed when using a RISC approach with few instructions. We thus have now a copy instruction, curiously named `move`, that copies the content of one register to another. Given the fact that OR or XOR is simpler than ADD in terms of hardware, the move instruction

```
move $rd, $rs
```

can better be translated into

```
or $rd, $zero, $rs
```

Likewise, as we have seen, the load-immediate value is not implemented, and, as a fact, cannot be implemented, because a 32-bit value word *plus* the 6-bit opcode cannot ever fit in a 32-bit instruction. Therefore, only a load-*upper*-immediate instruction (`lui`) exist. This places the 16-bit halfword immediate value contained in the instruction into the upper 16 bits of the

destination register. 16 bit (value) plus 6 bit (opcode) plus 5 bit (destination register specification) do easily fit in a 32-bit instruction. The lower 16 bits halfword would then require a similar load-lower-immediate `lli` instruction. However, once again the engineers of MIPS were very smart, realizing that this function is redundant, since it is equivalent to

```
ori $rd, $zero, halfword
```

So, the instruction

```
li $rd, word
```

can be translated into the two MIPS instructions

```
lui $rd, halfword1
```

```
ori $rd, $zero, halfword2
```

with `halfword1` equal to `word` logic-shifted-right 16 bits, and `halfword2` equal to `word AND 0x0000FFFF`, calculations the assembler must perform.

We can call these instructions pseudo-instructions; they are not directly implemented in hardware, but can easily be translated into real instruction(s). An assembler such as MARS we are using (or alternatively SPIM, another well-known MIPS assembler) supplies a list of pseudo-instruction mnemonics and their implementation. Some of them may be directly translated into machine code, while others are translated into other instruction(s) before being finally translated into machine code. In any case, it is free to the developer of the assembler to chose the names for the mnemonics. What is not free to chose is the resulting binary machine code (opcodes, etc.), which has been designed by the MIPS processor developer. If I were writing an assembler, I'd rather chose `copy` for the mnemonic instead of `move`, since `move` hints at that it disappears at the source, which it doesn't. However, 'move' seems to be rather standard in the computer architecture world.

Many assemblers also allow for designing our own pseudo-codes. These are called macros and come in handy when we have code we often write. In MARS, a macro is written by

```
.macro macroname (%parameter1, %parameter2, ...)
    MIPS code here
.end_macro
```

A simple example without parameters is an 'instruction' to end our program

```
.macro done
    li $v0, 10
    syscall
.end_macro
```

and we can now simply terminate our program with the instruction

```
done
```

To show how a macro can take parameters, we implement a terminate-with-return-value macro:

```
.macro return (%exitcode)
    li $v0, 17
    li $a0, %exitcode
```

```
    syscall
.end_macro
```

This can now be used to terminate our program with an error code, maybe

```
    return (-1)
```

In this case the `%exitcode` is verbatim copied in the interpretation of the instructions within the macro. If you use it with an integer, as in `return(1)`, it will result in an instruction `li $a0, 1`. If you use it with a register, as in `return($t0)`, it will translate into `li $a0, $t0`, which does not make sense and will generate an error at assemble-time.

Appendix H shows a list of some useful macros/pseudo-instructions implemented in MARS, and written by the author. First the instruction from incrementing and decrementing registers, useful when implementing for-loops.

```
.macro inc %reg
    addi %reg, %reg, 1
.end_macro

.macro dec %reg
    addi %reg, %reg, -1
.end_macro
```

After which we have the two pseudo-instructions

```
inc $rt
dec $rt
```

which we might put on our personal instructions reference card if we wish to do so. Maybe the most useful are stack-instructions `push` and `pop`, placing registers on the stack and removing items from the stack respectively:

```
.macro push %reg
    addi $sp, $sp, -4
    sw %reg, 0($sp)
.end_macro

.macro pop %reg
    lw %reg, 0($sp)
    addi $sp, $sp, 4
.end_macro
```

But macros can even be very sophisticated, including nesting (macros using other macros) as this example of a full for-loop printing the squares of numbers from 1 to 10 shows:

```
.macro square (%intreg)
    mult %intreg, %intreg
    mflo %intreg
.end_macro

.macro printsquare (%intreg)
```

```

    move $a0, %intreg
    square $a0
    li $v0, 1
    syscall
    li $a0, 0x20 # space
    li $v0, 11
    syscall
.end_macro

.macro for (%regi, %fromi, %toi, %whattodomacro)
    ori %regi, $zero, %fromi
    forloop:
        %whattodomacro (%regi)
        addi %regi, %regi, 1
        ble %regi, %toi, forloop
.end_macro

    for ($t0, 1, 10, printsquare)

```

Which will output 1 4 9 16 25 36 49 64 81 100 . (Note that it is not really a C for-loop, which, as we know, verifies the condition at the beginning of the loop, making it possible that the `whattodomacro` is never even run once. A good exercise is to convert the code above to make it C compatible, `for(i=1; i<=10; i++) printsquare(i);`).

Note that these are not functions, although they may very much look like them; the for-loop example above comes very close to a program in C. Yet, these are simply macros that are transcribed by the assembler into real MIPS code and then translated into machine language. Nothing more. At runtime the macros are gone, while C-functions are part of the executable code.

10.10 Extensive example: Gauss method for solving equations

With our knowledge of MIPS we can now work on more complex matters. As an example we will write a program for implementing the Gauss method for solving equations. As we have learned from linear algebra, we can solve a set of n equations with n incognitos using this method. We are going to do exactly the same here now. (A note up front: we will use the informatics convention that indexes start with 0). Detail: we only consider 'nice' sets of equations here that can be solved and give no trouble in rounding. As a

test, we will solve the following set of equations:

$$\begin{aligned} 2x + y - 3z &= -1 \\ -x + 3y + 2z &= 12 \\ 3x + y - 3z &= 0 \end{aligned}$$

which, as we can easily find out, has the solution $x = 1, y = 3, z = 2$. How does the method of Gauss work? We start by constructing an $(n + 1) \times n$ matrix with the coefficients and the independent values. In this case a 4×3 matrix:

$$\begin{pmatrix} 2.0 & 1.0 & -3.0 & -1.0 \\ -1.0 & 3.0 & 2.0 & 12.0 \\ 3.0 & 1.0 & -3.0 & 0.0 \end{pmatrix}.$$

Step 1: on line 0, divide all elements by a_{00} ($= 2.0$):

$$\begin{pmatrix} 1.0 & 0.5 & -1.5 & -0.5 \\ -1.0 & 3.0 & 2.0 & 12.0 \\ 3.0 & 1.0 & -3.0 & 0.0 \end{pmatrix}.$$

We now have a desired 1 on the diagonal.

Step 2: With the diagonal element ($a_{00} = 1$) zero out all elements above and below, i.e., subtract line 0 from line 1 with factor $= a_{01}/a_{00}$ ($= -1.0$), or add line 0 to line 1 with factor $= -a_{01}/a_{00}$ ($= 1.0$):

$$\begin{pmatrix} 1.0 & 0.5 & -1.5 & -0.5 \\ 0.0 & 3.5 & 0.5 & 11.5 \\ 3.0 & 1.0 & -3.0 & 0.0 \end{pmatrix},$$

and line 0 from line 2 (subtract with factor $= a_{02}/a_{00} = 3.0$)

$$\begin{pmatrix} 1.0 & 0.5 & -1.5 & -0.5 \\ 0.0 & 3.5 & 0.5 & 11.5 \\ 0.0 & -0.5 & 1.5 & 1.5 \end{pmatrix}.$$

Repeat for all n lines. Goto step 1 and do the same with a_{11} , divide line by a_{11} ($= 3.5$) and zero out all elements above and below a_{11} . Repeat for all n diagonal elements (2 in this case). The final matrix is then

$$\begin{pmatrix} 1.0 & 0.0 & 0.0 & 1.0 \\ 0.0 & 1.0 & 0.0 & 3.0 \\ 0.0 & 0.0 & 1.0 & 2.0 \end{pmatrix}.$$

The solution is now the last column of the matrix, which we print. The MIPS program below implements this algorithm. We make functions for all the important passes. All these functions receive the address of the matrix in **\$a0** and its dimension n in **\$a1**. Other arguments are passed in **\$a2** and **\$a3**, and in one case a float is directly passed through **\$f12**:

- `element_ij` places the value of a_{ij} in `$v0`, and its address in `$v1` (handy, as we will see). C equivalent:
`float element(float *a, int n, int i, int j)`
- `printmatrix` prints the $(n+1) \times n$ matrix. `printsolution` prints the last column (n) of this matrix. C equivalent:
`void printmatrix(float *a, int n)`
`void printsolution(float *a, int n)`
- `diagonal_i` creates a 1 on diagonal a_{ii} . C equivalent:
`void diagonal(float *a, int n, int i)`
- `subtractline_i1i2f` subtracts line i_1 from line i_2 with a factor f (passed in `$f12`). C equivalent:
`void subtractline(float *a, int n, int i1, int i2, float f)`
- `sweeppdown_x` and `sweepup_x` clear the matrix below and above diagonal element a_{xx} . C equivalent:
`void sweeppdown(float *a, int n, int x)`
`void sweepup(float *a, int n, int x)`

To highlight the use of registers and stack, we will use s-registers for the line index and t-registers for the column index. Very much care has to be taken to save the registers (of the correct type) at the correct place. Remember, all t-registers that will still be used, have to be saved on the stack by the calling routine just before calling the function (and restored immediately after returning from the function) while s-registers, those that will be used by the function, have to be saved on the stack by the called function right at the start, and recovered from the stack just before leaving the function.

```
#####
#   MIPS Assembly macros   #
#   macros.asm             #
#####

.macro push %reg
    addiu $sp, $sp, -4
    sw %reg, 0($sp)
.end_macro

.macro pop %reg
    lw %reg, 0($sp)
    addiu $sp, $sp, 4
.end_macro

.macro inc %reg
    addi %reg, %reg, 1
.end_macro
```

```
.macro dec %reg
    addi %reg, %reg, -1
.end_macro

.macro done
    li $a0, 0
    li $v0, 17
    syscall
.end_macro
```

```
#####
#   MIPS Assembly program using Gauss' method to solve   #
#   a system of n equations with n unknowns             #
#####
.include "macros.asm"

.eqv n 3 # number of independent variables

.data
spaces: .asciiz "   "
newline: .asciiz "\n"
textstart: .asciiz "Start array:\n"
textend: .asciiz "End array:\n"
textendresult: .asciiz "End result:\n"
array: .float
2.0, 1.0, -3.0, -1.0,
-1.0, 3.0, 2.0, 12.0,
3.0, 1.0, -3.0, 0.0

.text
main:
    # print start array:
    la $a0, array
    li $a1, n
    la $a2, textstart
    jal printarray

    li $t0, 0    # i
    li $t1, n
startdiagonalloop:
    beq $t0, $t1, enddiagonalloop
    la $a0, array
    li $a1, n
    move $a2, $t0
    push $t0
    push $t1
    jal diagonal_i
    pop $t1
    pop $t0
    # now we have a 1 at the diagonal a[i,i]
```

```

    la $a0, array
    li $a1, n
    move $a2, $t0 # x
    push $t0
    push $t1
    jal sweepdown_x
    pop $t1
    pop $t0
    # column below a[i,i] is all zeros!
    la $a0, array
    li $a1, n
    move $a2, $t0 # x
    push $t0
    push $t1
    jal sweepup_x
    pop $t1
    pop $t0
    # column above a[i,i] is all zeros!
    inc $t0 # next diagonal
    j startdiagonalloop
enddiagonalloop:

    la $a0, array
    li $a1, n
    la $a2, textend
    jal printarray

    la $a0, array
    li $a1, n
    la $a2, textendresult
    jal printsolution

    done # terminate

#----- FUNCTIONS -----#

#####
sweepdown_x:
#####
# Zeros all elements in a column x starting from line x+1 #
# with help from element[x,x]=1.0 #
# arguments: #
# a0 = *array #
# a1 = n #
# a2 = x #
# return: #
# void #
#####
    push $ra
    push $s0

    move $s0, $a2 # save original x
    li $t1, n

```



```

    move $t0, $s0
startloop_sd:
    inc $t0    # i++
    beq $t0, $t1, endloop_sd
    move $a2, $t0    # i
    move $a3, $s0    # j
    push $t0
    push $t1
    jal element_ij   # v0 now contains a[i,j]
    pop $t1
    pop $t0
    mtc1 $v0, $f12   # f12 now contains a[i,j]
    li $a1, n
    move $a2, $s0    # i1
    move $a3, $t0    # i2
    push $t0
    push $t1
    jal subtractline_i1i2f
    pop $t1
    pop $t0
    j startloop_sd
endloop_sd:
    pop $s0
    pop $ra
    jr $ra

#####
    sweepup_x:
#####
#   Zeros all elements in a column x starting from line x-1 #
#   with help from element[x,x]=1.0                          #
#   arguments:                                              #
#       a0 = *array                                         #
#       a1 = n                                              #
#       a2 = x                                              #
#   return:                                                 #
#       void                                                #
#####
    push $ra
    push $s0

    move $s0, $a2 # save original x
    move $t0, $s0
startloop_su:
    dec $t0    # i--
    bltz $t0, endloop_su
    move $a2, $t0    # i
    move $a3, $s0    # j
    push $t0
    jal element_ij   # v0 now contains a[i,j]
    pop $t0
    mtc1 $v0, $f12   # f12 now contains a[i,j]
    li $a1, n

```

```

move $a2, $s0    # i1
move $a3, $t0    # i2
push $t0
jal subtractline_i1i2f
pop $t0
j startloop_su
endloop_su:
pop $s0
pop $ra
jr $ra

#####
subtractline_i1i2f:
#####
# Subtract line i1 from line i2 by factor f #
# arguments:                                #
#   a0 = *array                             #
#   a1 = n                                   #
#   a2 = i1                                 #
#   a3 = i2                                 #
#   f12 = f                                 #
# return:                                    #
#   void                                    #
#####
# save all $s regs that we'll use and $ra:
push $ra
push $s0
push $s1
push $s5
push $s6
move $s0, $0    # j
li $s1, n
inc $s1    # n+1
move $s5, $a2    # save i1
move $s6, $a3    # save i2
startloop_sl:
beq $s0, $s1, endloop_sl
move $a2, $s5    # i1
move $a3, $s0    # j
jal element_ij    # $v0 now contains a[i1,j]
mtc1 $v0, $f10    # f10 now contains a[i1,j]
mul.s $f10, $f10, $f12 # f10 now contains f*a[i1,j]
mtc1 $0, $f0
sub.s $f10, $f0, $f10 # f10 now contains -f*a[i1,j]
move $a2, $s6    # i2
move $a3, $s0    # j
jal element_ij    # $v0 now contains a[i2,j],
                  # $v1 contains addr of a[i2,j]
mtc1 $v0, $f11    # f11 now contains a[i2,j]
add.s $f11, $f11, $f10 # f11 now contains a[i2,j]-f*a[i1,j]
swc1 $f11, 0($v1)
inc $s0
j startloop_sl

```

```

endloop_sl:
    # recover all saved regs:
    pop $s6
    pop $s5
    pop $s1
    pop $s0
    pop $ra
    jr $ra

#####
    diagonal_i:
#####
    # Divides all elements on a line i by element a[i,i]  #
    # arguments:                                           #
    #     a0 = *array                                     #
    #     a1 = n                                           #
    #     a2 = i                                           #
    # return:                                              #
    #     void                                             #
#####
    push $ra
    push $s0
    push $s1
    move $a3, $t0    # element_ii
    push $t0
    jal element_ij   #$v0 = array[i,i]
    pop $t0
    mtc1 $v0, $f8    # f8 = divisor element; all elements
                    # on line should be divided by $s3
    # divide line loop j ($s0)
    li $s0, 0      # j
    li $s1, n
    inc $s1        # n+1
startloopj_d1:
    beq $s0, $s1, endloopj_d1
    la $a0, array
    li $a1, n
    move $a2, $t0
    move $a3, $s0    # element_ij
    push $t0
    push $t1
    jal element_ij   #$v0 = array[i,j], $v1 = addr of array[i,j]
    pop $t1
    pop $t0
    mtc1 $v0, $f12
    div.s $f12, $f12, $f8
    swc1 $f12, 0($v1)
    inc $s0
    j startloopj_d1
endloopj_d1:
    pop $s1
    pop $s0
    pop $ra

```

```

jr $ra

#####
printarray:
#####
# Prints an array (n+1)x(n)      #
# arguments:                    #
#     a0 = *array               #
#     a1 = n                    #
#     a2 = *titlestring         #
# return:                       #
#     void                      #
#####
push $ra
push $s0
push $s1
li $v0, 4
move $a3, $a0
move $a0, $a2
syscall
move $a0, $a3
li $t1, n
li $s1, n
addi $s1, $s1, 1 # n+1
move $t0, $0 # i
startloopi_pa:
beq $t0, $t1, endloopi_pa
move $s0, $0 # j
startloopj_pa:
beq $s0, $s1, endloopj_pa
la $a0, array
li $a1, n
move $a2, $t0
move $a3, $s0
push $t0
push $t1
jal element_ij #$v0 = array[i,j]
pop $t1
pop $t0
mtc1 $v0, $f12
li $v0, 2
syscall
la $a0, spaces
li $v0, 4
syscall
inc $s0
j startloopj_pa
endloopj_pa:
la $a0, newline
li $v0, 4
syscall
inc $t0
j startloopi_pa

```

```

endloopi_pa:
    la $a0, newline
    li $v0, 4
    syscall
    pop $s1
    pop $s0
    pop $ra
    jr $ra

#####
    printsolution:
#####
# Prints last column of array (n+1)x(n)      #
# arguments:                                #
#     a0 = *array                            #
#     a1 = n                                  #
#     a2 = *titlestring                      #
# return:                                    #
#     void                                    #
#####
    push $ra
    push $s0
    push $s1
    li $v0, 4
    move $a3, $a0
    move $a0, $a2
    syscall
    move $a0, $a3
    li $s1, n
    move $s0, $0    # i
startloopi_pr:
    beq $s0, $s1, endloopi_pr
    la $a0, array
    li $a1, n
    move $a2, $s0
    li $a3, n
    jal element_ij    #$v0 = array[i,j]
    mtc1 $v0, $f12
    li $v0, 2
    syscall
    la $a0, newline
    li $v0, 4
    syscall
    inc $s0
    j startloopi_pr
endloopi_pr:
    la $a0, newline
    li $v0, 4
    syscall
    pop $s1
    pop $s0
    pop $ra
    jr $ra

```

```
#####
element_ij:
#####
# Returns element array[i,j]                                #
# arguments:                                                  #
#     a0 = *array                                             #
#     a1 = n                                                  #
#     a2 = i                                                  #
#     a3 = j                                                  #
# return                                                      #
#     $v0 = array[i,j]                                        #
#     $v1 = address of element array[i,j]                    #
#####
    move $t0, $a1
    addi $t0, $t0, 1
    mult $t0, $a2
    mflo $t0
    add $t0, $t0, $a3
    li $t1, 4 #sizeof(float)
    mult $t0, $t1
    mflo $t0
    add $v1, $a0, $t0
    lw $v0, 0($v1)
    jr $ra
```

Output:

```
Start array:
2.0   1.0   -3.0   -1.0
-1.0   3.0   2.0   12.0
3.0   1.0   -3.0   0.0

End array:
1.0   0.0   0.0   1.0
0.0   1.0   0.0   3.0
0.0   0.0   1.0   2.0

End result:
1.0
3.0
2.0
```

10.11 Calculating blockchain

Time to look at an extensive example to finalize this chapter. 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 assembly 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.

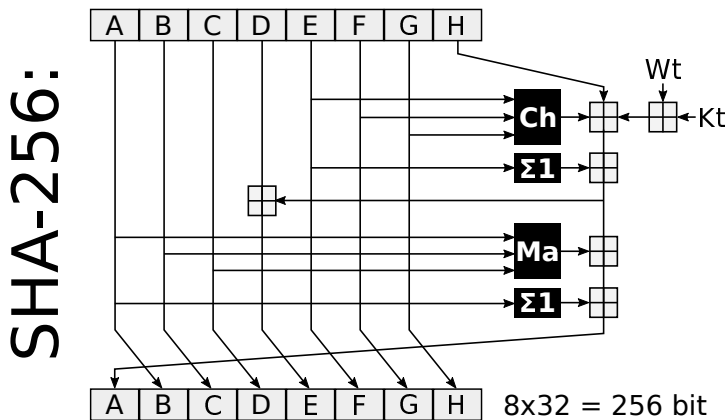


Figure 97: Example of the SHA-256 algorithm of encrypting data (A-H).
Figure adopted from Wikipedia.

The most famous is the secure hash SHA-256 algorithm that is shown in Figure 97 (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 $\Sigma 0$ 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 G. In this way, the bits of F and G are shuffled together based on the value of E.
- The next box, $\Sigma 1$, rotates and sums the bits of E, similar to $\Sigma 0$ except the shifts are 6, 11, and 25 bits.
- The plus boxes, \oplus , 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: Blockchain

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.

10.12 RARS: Evolution of MARS

Like with any technology, over the years improvements were made. This led to a new version of the architecture of MIPS, named RISC-V. Without going into a lot of detail, we highlight here the major differences at the level of Assembly. A good MIPS RISC-V Assembler is RARS (RISC-V Assembler and Runtime Simulator) that can be found at TheThirdOne at [github.com](https://github.com/TheThirdOne/rars) (<https://github.com/TheThirdOne/rars>). The environment is very similar to MARS to which you are by now very familiar. Or should be. A full manual on RISC-V can be found at <https://riscv.org/technical/specifications/>. The main differences between RARS en MARS are:

- RISC-V is extensible to 64 bit.

Table XXXIV: RISC-V (RARS) registers

Integer registers:							
x0	zero	x8	s0	x16	a6	x24	s8
x1	ra	x9	s1	x17	a7	x25	s9
x2	sp	x10	a0	x18	s2	x26	s10
x3	gp	x11	a1	x19	s3	x27	s11
x4	tp	x12	a2	x20	s4	x28	t3
x5	t0	x13	a3	x21	s5	x29	t4
x6	t1	x14	a4	x22	s6	x30	t5
x7	t2	x15	a5	x23	s7	x31	t6

Floating-point registers:							
\$f0	\$ft0	\$f8	\$fs0	\$f16	\$fa6	\$f24	\$fs8
\$f1	\$ft1	\$f9	\$fs1	\$f17	\$fa7	\$f25	\$fs9
\$f2	\$ft2	\$f10	\$fa0	\$f18	\$fs2	\$f26	\$fs10
\$f3	\$ft3	\$f11	\$fa1	\$f19	\$fs3	\$f27	\$fs11
\$f4	\$ft4	\$f12	\$fa2	\$f20	\$fs4	\$f28	\$ft8
\$f5	\$ft5	\$f13	\$fa3	\$f21	\$fs5	\$f29	\$ft9
\$f6	\$ft6	\$f14	\$fa4	\$f22	\$fs6	\$f30	\$ft10
\$f7	\$ft7	\$f15	\$fa5	\$f23	\$fs7	\$f31	\$ft11

- No commas needed in RARS code, although can be used if wanted.
- No \$ to specify a register, the name of the register suffices. Of course no labels with names of registers can be used.
- Numbered registers prefixed with x. Example: x6 is equal to t1.
- More 'argument' registers: a0 to a7. See Table XXXIV for a list of registers.
- No 'return-value' registers v0 and v1. Return values can be placed in the a-registers
- The special multiplication and division registers \$lo and \$hi do no longer exist. Multiplication and division instructions directly produce the result in a destination register. For example


```
mul t2 t1 t0
```

 (set t2 to the lowest 32 bits of t1×t0)
- To make code easily moved in space, addresses are described relative to the program counter. Loaded by the instruction `auipc` (add upper

immediate to pc). As an example, if a label points to an object in the data (text) segment at 0x1001000 and the following instruction is at the start of the code segment 0x00400000

```
0x00400000 auipc x5 0x0fc10
```

it loads the address 0x1001000 into x5 (t0). It will left-shift the immediate pattern by 12 places (adding 3 hexadecimal 0s) and add it to the program counter (0x00400000).

- Jump instruction `j` does not exist. It is implemented with a jump-and-link `jal` instruction, with the current address (not) saved in register `zero`

```
j label
```

becomes

```
jal zero label
```

With the attempt to write the pc in register `zero` ignored (or at least not successful). A regular `jal label` used for function calls is still `jal ra label`

- Likewise, the jump-(to-address-in)-register `jr` has a different syntax, it can now do relative addressing

```
jr reg
```

becomes

```
jalr reg 0
```

where an offset can be added to the address in register `reg`. (In this case the offset is 0)

- Small differences in syntax: `ecall` instead of `syscall` (with the system call number in `a7` instead of `$v0` and different registers used for the arguments and return values, see Appendix L). `asci` instead of `ascii`

- Macros:

`call label`: Call a function (jump and link), implemented with

```
auipc x6 0
jalr x1 x6 relative-address
```

The first instruction is 'add upper immediate to pc' and store in `t1` (x6). The second instruction adds the relative address calculated by the compiler to this stored pc and jumps to it, while storing the current address in `x1` (ra). Note that a value stored in `t1` (x6) will be lost by such a function call without our clear knowledge (since we only used the word `call`). But, as we already knew that the calling code (the 'caller') has the responsibility to save *all* the relevant 'temporary'

registers on the stack (see, Section 10.8) this adds no extra restriction.
ret: return from function (implemented with `jalr x0 x1 0`, see above)
j label: jump to label-address (implemented with
 `jalr x0 relative-address`,
 see above)

- A compiler directive **ebreak**. Which is rather an emulator-environment system feature and not a compiler directive, but can come in handy for debugging purposes
- All 32 floating point registers are 64 bit. (See Table XXXIV)
- Multi-tasking instructions **fence** and **fence.i** to synchronize execution.
- 4096 control status registers (CSR) and a set of special instructions dedicated to reading and writing in these control status registers. An example is the floating-point control-and-status register (FPCSR) at 0x003, see Fig. 98. It contains the rounding mode, and exception flags (NV, DZ, OF, UF and NX). An example is given below where the DZ flag is read that occurs when division by zero is performed.

```
#####
# Program in RARS RISC-V to read floating point #
# status register (FPCSR) #
#####

.data
crss: .asciz "FP status register (0x003): "
nvs: .asciz "\nNV: "
dzs: .asciz "\nDZ: "
ofs: .asciz "\nOF: "
ufs: .asciz "\nUF: "
nxs: .asciz "\nNX: "
errors: .asciz "\nDivision by zero!"

.eqv fpcsr 0x003

.text
li a7 4
la a0 crss
ecall

li t0 0
li t1 2
fcvt.d.w f0 t0
fcvt.d.w f1 t1
fddiv.d f2 f1 f0 # f2 = 2.0/0.0
```

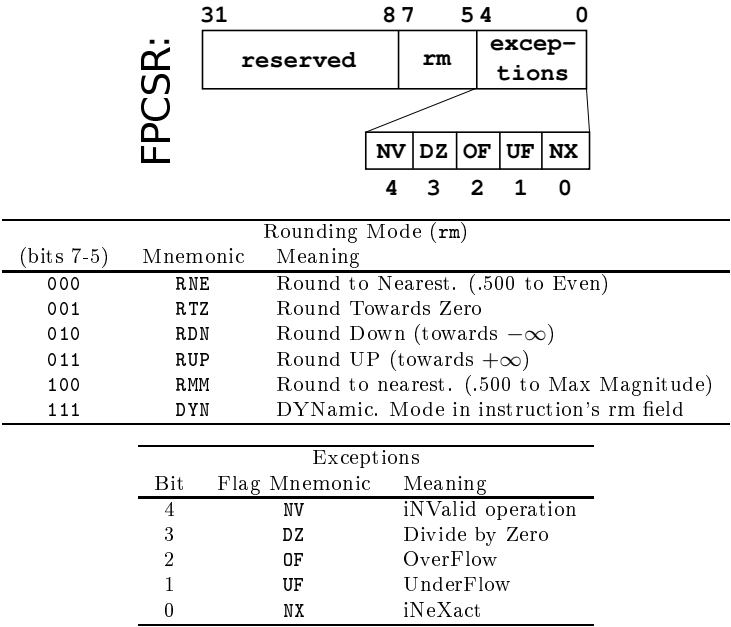


Figure 98: The RISC-V Floating-Point Control and Status Register (FPCSR).

```
csrr a0 fpcsr      # load FP status register in a0
andi a1 a0 8       # mask bit 3 (division by zero error)
srai a1 a1 3       # right-shift 3 digits
li a7 1
ecall              # print fpcsr

li a7 4
la a0 dzs
ecall              # print "DZ: "
mv a0 a1
li a7 1
ecall              # print DZ bit of FP status register

beqz a1 terminate
error:
la a0 errors
li a7 4
ecall

terminate:
li a7 10
ecall
```

Output:

```
FP status register (0x003): 8
DZ: 1
Division by zero!
-- program is finished running (0) --
```

- 32 64-bit counters in CSR registers, the lower 32-bits are in 0xc00-0xc1f, the associated higher 32 bits in registers 0xc80-0xc9f. "The first three of these (CYCLE, TIME, and INSTRET) have dedicated functions (cycle count, real-time clock, and instructions-retired respectively), while the remaining counters, if implemented, provide programmable event counting." Pseudo-instructions exist in RARS to have access to these counters through the `csrrs` RISC-V instruction.:

```
rdcycle rd
rdcycleh rd
rdtime rd
rdtimeh rd
rdinstret rd
rdinstreth rd
```

```
#####
# Program in RARS RISC-V to read a timer      #
#####

.data
prompt: .asciz "Give an integer:"
finaltext: .asciz "Cycles it took to reply: "

.text

repeatbefore:
    rdtimeh t2
    rdtime t1
    rdtimeh t3
    bne t3 t2 repeatbefore # make sure high did not
    # change while reading the times
    # start time in (t2, t1)

printsometext:
    li a7 4
    la a0 prompt
    ecall
    li a7 5
    ecall

repeatafter:
    rdtimeh t4
    rdtime t3
```

```

rdtimeh t5
bne t5 t4 repeatafter # make sure high did not
    # change while reading the times
# end time in (t3, t4)

li a7 4
la a0 finaltext
ecall

sub t2 t4 t2
sub t1 t3 t1
beqz t2 continue # high clock value changed?
li t5 0x80000000 # (+2^31)
add t1 t1 t5
add t1 t1 t5      # then t1 <-- 2^32+t1
continue:
li a7 1
mv a0 t1
ecall

terminate:
li a7 10
ecall

```

Output:

```

Give an integer:5
Cycles it took to reply: 3206
-- program is finished running (0) --

```

- Special attention deserves the calculation of the immediate value. The construction of the immediate value used in the operation depends on the type of instruction. As an example,

```
lw x6 100(x5),
```

(load into x6 the word found in address stored in x5 plus the immediate value 100). This is an instruction of type-I (see Appendix K) and the immediate value thus coded in the 12 most-significant bits (31-20) of the instruction. With the help of Appendix K we find that the opcode is 3 and the fun3-code is 2, with the type-I format given, the machine language instruction is

```

imm(12) tgt(5) fun(3) dst(5) opcode(7)
000001100100 00101 010 00110 0000011
0000 0110 0100 0010 1010 0011 0000 0011
0x 0 6 4 2 a 3 0 3

```

The immediate value is a sign-extended value, generated by adding 20 copies of the sign bit on the left of the 12-bit imm-field of the instruction above. In this case the sign bit is 0, so the final immediate value used in the operation is

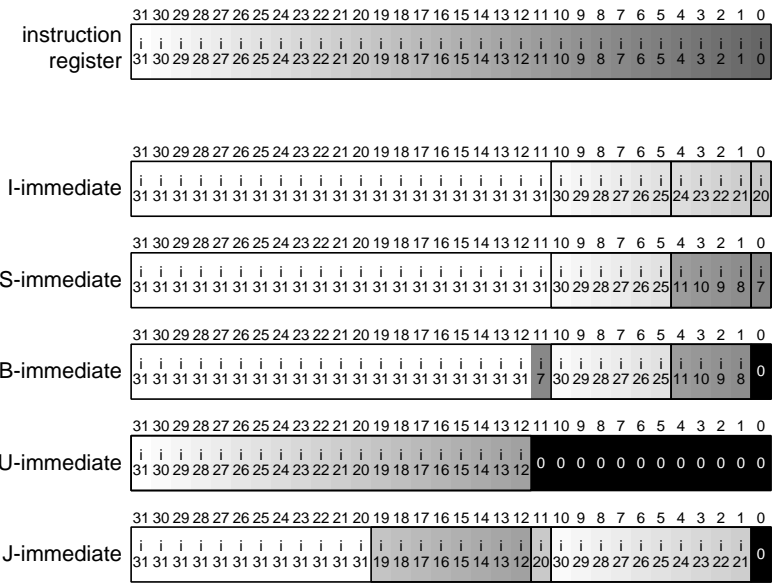


Figure 99: Construction of the immediate value of the operation based on the imm-field of the instruction for the different types of instructions. The numbers above the box are the bit numbers of the immediate value used in the operation and the numbers inside the box are the bit numbers of the instruction's imm-field (except when written '0', in which case the bit value is zero).

0000 0000 0000 0000 0000 0000 0110 0100

See Figure 99. We see that bits 31-11 of the final immediate value are copied from bit 31 of the imm-field, and bits 10-0 copied from bits 30-20 of the imm-field. The figure shows how the immediate value is constructed for other instruction types as well.

An example of a RARS program, an evaluator of a two-operand expression, is given here. (For this simple example we omit the part of saving registers on stack when calling functions):

```
#####
# Program in RARS RISC-V to calculate arithmetic #
# expressions                                     #
#####
```

```

.data

inputbuffer: .space 100
prompt: .asciz "Expression: "
errormessage: .asciz "Error in expression"

.text

main:

getexpressionstring:
    la a0 prompt
    li a7 4
    ecall
    la a0 inputbuffer
    li a1 100
    li a7 8
    ecall # read operation string

getfirstoperand:
    call strtoint # extract first operand
    mv a2 a1
    call skipspaces
getoperator:
    lb a3 (a0) # operator: + - * / %
    addi a0 a0 1
    call skipspaces
getsecondoperand:
    call strtoint # extract second operand

# now we have
#   a1: first operand int
#   a2: second operand int
#   a3: operation char

    call performoperation
    beqz a0 printresult
printerror:
    li a7 4 # print string
    la a0 errormessage
    ecall
    j endprog
printresult:
    li a7 1 # print int
    mv a0 a1
    ecall

endprog:
    li a7, 10
    ecall

#----- FUNCTIONS -----#

```



```

performoperation:
#####
# PERFORM OPERATION                                     #
#   input:                                              #
#       a1, a2: operands                               #
#       a3: operator                                   #
#   output:                                             #
#       a1: result                                      #
#       a0: 0 if no error                              #
#####
    li t1 0 # default: no error
    li t0 '+'
    bne a3 t0 notplus
    add a1 a1 a2
    j exitoperation
notplus:
    li t0 '-'
    bne a3 t0 notminus
    sub a1 a1 a2
    j exitoperation
notminus:
    li t0 '*'
    bne a3 t0 nottimes
    mul a1 a1 a2
    j exitoperation
nottimes:
    li t0 '/'
    bne a3 t0 notdiv
    div a1 a2 a1
    j exitoperation
notdiv:
    li t0 '%'
    bne a3 t0 nothing
    rem a1 a2 a1
    j exitoperation
nothing:
    li t1 -1 # error
exitoperation:
    mv a0 t1
ret

strtoint:
#####
# CONVERT STRING TO INT                                 #
#   input:                                              #
#       a0: string address                             #
#   output:                                             #
#       a1: int                                         #
#       a0: address of first non-digit char           #
#####
    mv a1 zero # result
    li t1 10
    li t2 1 # sign

```

```

    lb t0 (a0)          # digit
    li t3 '-'
    bne t0 t3 startconvert
    addi a0 a0 1
    li t2 -1
startconvert:
    lb t0 (a0)
    addi t0 t0 -48      # subtract '0' from char. '0' --> 0, etc,
    bltz t0 endconvert # if < 0
    bge t0 t1 endconvert # if >= 10
    mul a1 a1 t1        # a1 *= 10
    add a1 a1 t0        # a1 += t0
    addi a0 a0 1        # a0++
    j startconvert
endconvert:
    mul a1 a1 t2        # multiply by sign
    ret

skipspaces:
#####
# SKIP SPACES                                     #
#   input:                                       #
#       a0: string address                       #
#   output:                                       #
#       a0: address of first non-space char      #
#####
    li t1 ' '        # space
skiploadchar:
    lb t0 (a0)
    bne t0 t1 exitskipspaces
    addi a0 a0 1
    j skiploadchar
exitskipspaces:
    ret

```

Output:

```

Expression: -28 * 4
-112
-- program is finished running (0) --

```

11 | Examples of architectures

We can now take a look at some particular architectures. How they fit in the chronology of computer engineering. What makes them different from the others? This is limited to architectures based on electronics, with the exception of the first one. Generally speaking an architecture is described by the following aspects:

- **Underlying technology.** For instance, the Difference Engine of Babbage is based on mechanical columns to store decimal numbers. The Intel 4004 is based on MOS-FET (CMOS) random logic based on BCD calculations.
- **Operand/register size (ALU).** E.g. adding 8-bit data.
- **Operator size/instruction size.** E.g. 16-bit instruction register.
- **Data bus size.** E.g. 16-bit data bus.
- **Address bus size.** (E.g. 32 bit address $\rightarrow 2^{32}$ addresses). The addressable memory space depends on this address bus size and the data distance in memory (for instance 1 byte, then giving 2^{32} byte = 4 GB memory space).
- **Clock frequency** (both internal as well as the bus).
- **Number of possible instructions.** Viz. RISC vs. CISC.
- **Number of pins** of the CPU.
- **Memory organization.** Viz. Von Neumann vs. Harvard. Stacks and heaps, etc.
- **Number of registers.**

- **CPU/ALU implementation**, microcode or random logic.
- **Advanced features**.
- **Year** of development.
- **Number of transistors**.
- **Transistor size**. Also called minimum feature size (MFS).

Let’s take a look at some famous and important architectures.

11.1 Difference Engine of Charles Babbage

The Difference Engine of Charles Babbage was invented – or developed – at the end of the 18th century and built by Babbage in 1819-1822 and can really be called revolutionary. It was a mechanical computer made of vertical cylinders (see Fig. 16) and was designed to calculate polynomials (and thus any function, as we have seen with Taylor-expansions). Each cylinder containing/storing a single decimal digit . Execution consisted of adding the value of column $n + 1$ to column n . The name comes from calculating differences. This is best explained with an example. Imagine we want to calculate $p(x) = 2x^2 - 3x + 2$ for integer values, for instance for $x = 4$. We reset it with the seed values at $x = 0$. We store:

- Cylinder 0: the value (2) at $x = 0$
- Cylinder 1: the difference (−1) between the cylinder-0 value at $x = 0$ and $x = 1$
- Cylinder 2: the difference (4) between cylinder-1 values at $x = 0$ and $x = 1$
- Cylinder 3: the difference (0) between cylinder-2 values at $x = 0$ and $x = 1$

All other cylinders start with 0. Generally speaking: we store the *differences* (hence Difference Engine) and differences of differences between cylinder values at $x = 0$ and $x = 1$. The starting state is thus:

	Cyl0	Cyl1	Cyl2	Cyl3
x	$p(x)$	$\Delta p(x)$	$\Delta^2 p(x)$	$\Delta^3 p(x)$
0	2	−1	4	0

Now we let it ‘run’. Cyl1 will be added to Cyl0 resulting in Cyl0 being 1. Then Cyl2 is added to Cyl1, it becoming 3, and Cyl3 is added to Cyl2, staying 4. We thus have these steps, with the active cylinders indicated at each step:

x	Cyl0 $p(x)$	Cyl1 $\Delta p(x)$	Cyl2 $\Delta^2 p(x)$	Cyl3 $\Delta^3 p(x)$
0	2	-1	4	0
0	1	-1	4	0
0	1	3	4	0
1	1	3	4	0

The last cylinder (first zero) was activated and thus we now have a value for $x = 1$, namely 1. Let’s let it run some more iterations:

x	Cyl0 $p(x)$	Cyl1 $\Delta p(x)$	Cyl2 $\Delta^2 p(x)$	Cyl3 $\Delta^3 p(x)$
1	4	3	4	0
1	4	7	4	0
2	4	7	4	0
2	11	7	4	0
2	11	11	4	0
3	11	11	4	0
3	22	11	4	0
3	22	15	4	0
4	22	15	4	0

Thus we know that $p(4) = 22$. And that done on a mechanical computer! So, the Difference Engine can do polynomial calculations. Many functions can be converted into polynomials. And we can use a convention that the integers in the cylinders are scaled, for instance by a factor 100, so that 34 in fact represents 0.34. Any function can be calculated to any precision when given enough time. What used to be a tedious job of calculating manually tables of functions – entire halls were filled with (mostly) women doing pen-and-paper calculations all day all their lives – could now be done mechanically.

11.2 Intel 4004

The Intel 4004 was developed in 1971 by Intel for Busicom and can be considered the first CPU with all features on board a single chip. (Don’t forget, electronic calculators already existed before). It was based on MOS-FET CMOS technology and it implemented the operations by random logic instead of using micro-code.

The single chip was very small compared to modern CPUs. It had only 16 pins (DIL, dual in-line package, see Figure 100) compared to modern CPUs that have about 500 pins. This was, however, the starting point of the entire Intel architecture that most computers still use today.



Figure 100: Left: DIL chip of an Intel 4004. (Wikipedia) Right: pin-out of chip.

Some specifications:

- Random logic.
- 4-pin external data bus to memory chips.
- 12 bit address, together with 8-bit address distance gives $2^{12} = 4096$ bytes memory space.
- Clock: 740 kHz.
- 2300 transistors with $10\ \mu\text{m}$ MFS. An area of *one* of the Intel 4004 transistors can, with modern AMD Ryzen technology, contain about 1 million modern transistors, or, in other words, about 4 thousand entire Intel 4004 processors!
- Data: binary-coded decimal (BCD).
- Von Neumann memory architecture, and program separated from data in memory.
- Instruction set: 46 instructions, with 8 bit or 16 bit size. We thus have variable-length instruction set
- Registers:
 - 16×4 bit data
 - Instruction register: 8 bit
 - Program counter: 12 bit
 - Stack registers: 12 bit. Up to 3 levels only
 - Accumulator: 4 bit. To store temporary results
 - Flags register

- Temporary register.
- 4-bit internal data bus.

For more details, see the datasheet "4004 single chip 4-bit p-channel microprocessor" from Intel. The 4004 chip worked together with a 4001 ROM, a 4002 RAM and a 4003 shift register. Together they were the MCS-4 'chip set' (a word we still use today to describe the basic chips on a motherboard).

The 4004 instruction set is variable-instruction-size. Most instructions (see Appendix A) are 8-bit (1 byte) but some are 16-bit (2 byte) long. An example of an 8-bit instruction is fetch-indirect, **FIN** (0011) with the next nibble containing the 3 bits specifying the register pair that contains the address of the data and a 0. So the full instruction is:

Assembly: **FIN** {R,R+1}
Binary: 0011 RRR0

with **RRR** specifying the register pair. For instance, using registers 2 and 3 would be **FIN** 0010 and thus 0011 0010.

Another 8-bit example: Adding the contents of register **R** to the accumulator:

Assembly: **ADD** R
Binary: 1000 RRRR

with **RRRR** specifying the register. For example, adding: **ALU** + register 4 → **ALU** is **ADD** 0100 and thus 1000 0100.

A 16-bit instruction is fetch immediate (**FIM**, 0010) that copies the 8-bit value **D** to register pair {**R**,**R+1**}

Assembly: **FIM** {R,R+1} D
Binary: 0010 RRR0 DDDD DDDD

Note that there are no multiplication or division instructions. But, as we have seen in this book, divisions and multiplications are done by shift-mask-and subtract/add. In the 4004 instruction set there is also no masking instruction, nor any shift. However, the rotate-right (**RAR**, see Figure 102) and rotate-left (**RAL**) put the bit that was shifted out of the register into the carry bit. This carry bit can then be read and the adding of the operand performed if necessary (see the Russian-peasant algorithm).

Successors to the 4004 were (chronologically): Intel 4040 → 8008 → 8080 → 8085 → 8088 → 8086 → x86 family.

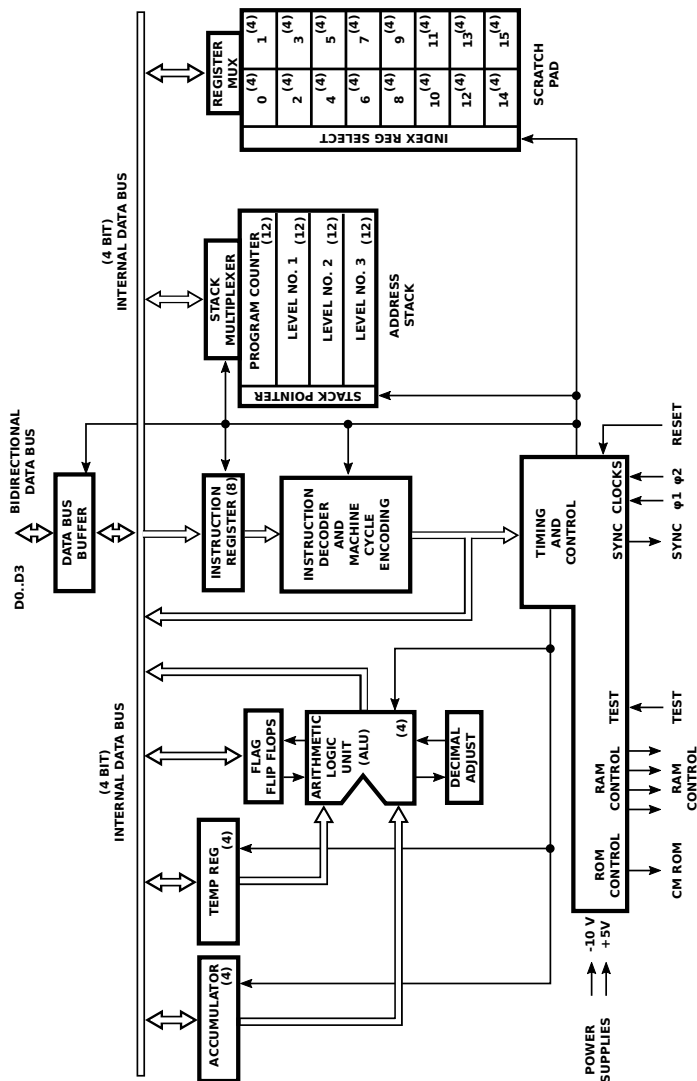


Figure 101: The Intel 4004 architecture (redrawn from the Intel 4004 datasheet). The thick arrows are the data bus. The thin arrows are the control lines. The register sizes are indicated in parentheses.

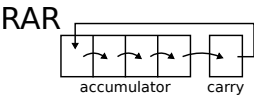


Figure 102: The Intel 4004 instruction 'rotate-right through carry' (RAR) rotates the 4 bits of the accumulator and the carry.



Figure 103: MOS 6502 40-pin-DIL chip. (Wikipedia).

11.3 MOS 65xx

The MOS 6502 was designed by the team of Chuck Peddle. The idea was to make an ultra-cheap CPU. Compare the price of \$20 to the competitor 6800 of Motorola: \$700 (1975). It became a breakthrough in the personal computer market: Apple, Apple II, Commodore C=64, Nintendo, Atari, BBC Micro, etc. (And Terminator T800; 6502 code appears on the head-up display of the Terminator robot from the famous movies).

Some specifications:

- 3510 transistors on 5×5 mm².
- 40 pin DIL.
- 4 MHz clock frequency.
- Few registers:
 - Program counter, PC (16 bit)
 - Accumulator, A (8 bit)
 - X, Y registers (8 bit)
 - Status register (8 bit): N=negative, V=overflow, 1, B=break, D=decimal mode, I=IRQ disable, Z=zero, C=carry
 - Instruction register
 - Stack pointer (8 bit).
- 16-bit address bus, address distance 8 bit: $2^{16} \times 8 = 64$ kB address space.
- 8-bit data bus and registers (A, X, Y).
- No tri-state bus technology.
- Rudimentary pipelining: The last clock cycle of an instruction already fetching a new instruction.

The 6502 has few registers (only X and Y), but used a hardware trick. At the time memory was as fast as CPU registers. The 6502 used so-called zero-page addressing. Most 65xx operations have such variants. When this addressing mode is used, the next byte (8 bit) in the program code is the

address in the first page ('zero page') of memory containing the operand or destination. With 8 bits we can address $2^8 = 256$ addresses. So, effectively there are 256 registers. Of course, the code is slower, because another byte has to be fetched from memory. Note that it thus also means the processor uses variable-length instructions.

In total there are 16 addressing modes, see Appendix B. Some examples:

- **implied**: for instance INX (increase X register by one).
- **immediate**: the operand is simply the next byte in the code segment at PC+1.
- **absolute**: the operand is at the address given by the next two bytes in the code segment, least significant byte first.
- **accumulator**: the operand is in the accumulator.
- **stack**: the operand is at the address pointed at by the stack pointer.
- **zero page**: the operand is at address given by the next byte of code.
- **absolute indexed**: the operand is at the address given by the next two bytes of code (LSB first) plus the value of the the register (X or Y).
- **absolute indirect**: the operand is at the address at the address given by the next two bytes of code (LSB first).
- **absolute indexed indirect**: the operand is at the address at the address given by the next two bytes of code (LSB first) plus the value of the the register (X or Y).

As can be seen from the addressing modes, instructions have different lengths: 1, 2 or 3 bytes. There is therefore no classic instruction register! In total there are 70 instructions, see Appendix B. Each with several addressing modes. Nearly all of the ($2^8 = 256$) possible opcodes were used. It is still RISC technology.

As an example, a jump to address 0xFF00, absolute addressing, would be (\$ means hexadecimal in 65xx Assembly)

assembly: JMP a, \$FF, \$00

16-bit address words are Little-Endian, lo(w)-byte first, followed by the hi(gh)-byte. (An assembler will use a human readable, Big-Endian notation as in HHLL). We find in a the instruction-set table that JMP a is 0x4C, so the hexadecimal machine language code becomes

4C 00 FF

or in binary

```
0100 1100 0000 0000 1111 1111
```

The 6510 was an evolution of the 6502 in that it had a built-in I/O port. It was used in the famous Commodore 64, until today still the most-sold computer in the world.

If we look at the instruction set (Appendix B) we can write a 6502 Assembly program writing "Hello world!". Here is a program for the 6502 Macroassembler & Simulator of Michal Kowalski available on-line

```
; -----;
; Writes "Hello, World!" in MOS 6502      ;
; Runs in the Coldfire Windows 95 6502 Emulator and ;
; Simulator of Michal Kowalski, available at      ;
; http://exifpro.com/utills.html      ;
; -----;

      *= $0600      ; start address of code segment

; assembler definitions:
IOputchar      = $E001
IOsetcursorX   = $E005
IOsetcursorY   = $E006
ASCII_CR       = $0D
ASCII_LF       = $0A

; code segment:
      ; Set cursor at (0,0):
      LDA #0
      STA IOsetcursorX
      LDA #0
      STA IOsetcursorY

      LDX #0
printchar:
      LDA hellow,x ; load char hellow[x] into A
      BEQ done     ; if char=0 then goto done
      STA IOputchar ; print char
      INX          ; increase X
      JMP printchar ; goto printchar
done:
      BRK          ; terminate program

; data segment:
hellow:
.BYTE "Hello world!", ASCII_CR, ASCII_LF, $00
```

The assembler creates the following label look-up table:

```

IOputchar = $E001
IOsetcursorX = $E005
IOsetcursorY = $E006
printchar = $060C
done = $0618
hellow = $061A

```

So that the final program would be assembled into

```

<code segment>:
LDA #0      : A9 00
STA IOsetcursorX : 8D 05 E0
LDA #0      : A9 00
STA IOsetcursorY : 8D 06 E0
LDX #0      : A2 00
LDA hellow,x : BD 1A 06
BEQ done     : F0 07
STA IOputchar : 8D 01 E0
INX          : E8
JMP printchar : 4C 0C 06
BRK          : 00 00
<data segment>:
"hello "     : 48 65 6C 6C 6F 20
"world!"     : 77 6F 72 6C 64 21
CR LF 0      : 0D 0A 00

```

Thus

```

$0600: A9 00 8D 05 E0 A9 00 8D
$0608: 06 A2 00 BD 1A 06 F0 07
$0610: 8D 01 E0 E8 4C 0C 06 00
$0618: 00 48 65 6C 6C 6F 20 77
$0620: 6F 72 6C 64 21 0D 0A 00

```

Output:

```

Hello world!
139

```

Let's do something a little more complicated, in which we can also see the power of zero-page addressing mode. The program below again writes "Hello world!" to the screen, but now in a function call. Because MOS 65xx is an 8-bit architecture (registers are 8-bit) the address to the function cannot be stored in a register. Instead, it is passed to the function in the zero-page dual address {\$01,\$00}. In the loop zero page indexed addressing is used (for some reason the assembler does not accept X as index, so Y is used instead). Note also the convention used for labels. Those available for users calling

the function start with a normal letter. Those that are 'internal' and not available to – or should not be used by – the users start with an underscore `_`. The program also has a routine for printing an integer. Analyze the code and try to understand it.

```
; -----;
; MOS 6502 Assembly. Prints a string and an integer ;
; using functions ;
; Runs in the Coldfire Windows 95 6502 Emulator and ;
; Simulator of Michal Kowalski, available at ;
; http://exifpro.com/utills.html ;
; -----;

        *= $0600          ; start address of code segment

; assembler definitions:
ASCII_CR    = $0D
ASCII_LF    = $0A

; code segment:
        LDA #<hellow      ; get lo byte of address
        STA $00           ; store in zero page
        LDA #>hellow      ; get hi byte of address
        STA $01           ; store in zero page
        JSR printstr      ; jump to subroutine

        LDA #139
        JSR printint

        BRK              ; terminate program

; data segment:
hellow:
        .BYTE "Hello world!", ASCII_CR, ASCII_LF, $00

;----- FUNCTIONS : -----

printstr:
        ;-----
        ; ($01,$00): address of null-terminated string
        ;-----
; assembler definitions:
IOputchar   = $E001

        LDY #0
_loopstr:
        LDA ($00),Y      ; load char address[y] into A
        BEQ _donestr     ; if char=0 then goto done
        STA IOputchar    ; print char
        INY              ; increase Y
        JMP _loopstr     ; goto _loopstr
_donestr:
        RTS              ; exit subroutine
```

```

printint:
;-----
; A: value to print
;-----
LDX #$FF
SEC
_loop100:
INX
SBC #100
BCS _loop100
ADC #100
JSR _printdigit
LDX #$FF
SEC
_loop10:
INX
SBC #10
BCS _loop10
ADC #10
JSR _printdigit
T%   ORA #48; ASCII '0'
STA IOputchar
PLA
RTS
;----- end of functions : -----

```

Output:

```

Hello world!
139

```

Finally, instead of emulating a processor and writing assembly code for it, we can also emulate an entire machine, including the processor, BIOS (kernel) routines, interfaces and peripheral equipment and run a normal program designed for that computer in that emulator. For the MOS 6510 there exists a very good emulator, called Vice. This emulates a Commodore 64 (or similar computers from the same Commodore 65xx family). In Linux it can be installed from the Software Manager. The Vice emulator comes without the ROM/kernel routines (note the kernel for the Commodore was for some reason called 'kernel'). That is because these routines are still copyrighted and we have to get them separately. Fortunately they are readily available online (at the moment of writing this, the archive is called `vice-1.5-roms.tar.gz`) and contain all code needed: ROM/kernel routines to operate the cassette, disk and I/O. It should be copied to directory `/usr/lib/vice`, after which Vice should start smoothly.

Software disks (files with extension `.d64`) can then be found online at

<https://commodore.software>

There exist many assemblers for the C=64, but to be able to use it, one also needs a manual with the instructions how to use it, or it would be stabbing in the dark. One assembler that has this requisition is Merlin 64 for which a book by Glen Bredon can be found too (Roger Wagner Publishing Inc.).

It is important to note that, although the processor for which the assemblers are written might be the same, assemblers can have different syntax. The assembly language can be different and in the worst case even have different mnemonics. Yet, the machine code it generates is equal. For instance, comment can in some languages be written by a semicolon, others by %, others by a (double) slash, others by asterisk. It depends on the preferences of the people that wrote the assembler. With that in mind, we can take a look at a Merlin 64 Assembly program for the Commodore 64 and see the syntax is different from the one above. See the example in Figure 104. We enter the editor/assembler by typing 'E' and then 'A' for writing code. We can start writing our source code here, starting with the comment *HELLO WORLD. Some observations:

- Lines are numbered (these numbers are not part of the code, but only shown for convenience). A code line starting with an '*' is considered comment. Text on a line after any ';' is ignored too.
- The code has three columns: label, mnemonic, operand.
- A mnemonic **ORG** tells the assembler what is the starting address for the assembled machine code program.
- The mnemonic **EQU** is an assembler directive like 'define'. When the label defined like this is encountered in the source code, it is substituted by the Merlin assembler by the value (in the operand column).
- A mnemonic **TXT** means the operand is ASCII text (for example, 'A' will be translated into hexadecimal 41), **HEX** means it is a hexadecimal value (without the need to specify \$).
- # means immediate, \$ means a hexadecimal value, % means a binary value.

```
*HELLO WORLD IN MERLIN 6510 ASSEMBLY COMMODORE 64
      ORG      $8000      ; start of code segment
CHROUT EQU     $FFD2      ; assembler directives
SETCUR EQU     $FFF0      ; equivalence
BRDCOL EQU     $D020      ;
      LDA      #1         ; load-imm 1 into reg A
      STA      BRDCOL     ; store A into addr
      LDY      #0         ; load-imm 0 into reg Y
      LDY      #0         ; load-imm 0 into reg Y
      CLC          ; clear carry
```

Table XXXV: Steps for the C=64 Merlin Assembly example of Fig. 104. See Table XXXVI for Merlin commands.

Load VICE
Install ROMS (in directory /usr/lib/vice/)
Attach disk merlin64_assembler.d64 to drive 8 (drive type 1540 II)
LOAD "*",8
RUN
(starts gray Merlin screen after a while)
%E
(enters editor)
:A
(add code)
Write program source code of Figure 104
On empty line type RETURN, exits editing
:ASM
answer "n" to question if you want to update source
:MON
(enters monitor)
\$8000l
(last char is 'el', not 'one'. lists program in memory at \$8000)
\$8000g
(runs the program starting at \$8000)
Admire the beautiful text Hello world! on the top left of your screen
\$q
(quits editor/monitor)

	JSR	SETCUR	; call subr. SETCUR
	LDX	#0	; reset reg X
LOOP	LDA	DATA,X	; A <-- M[DATA+X]
	BEQ	DONE	; if 0 then goto DONE
	JSR	CHROUT	; call subr. CHROUT
	INX		; x++
	JMP	LOOP	; goto LOOP
DONE	RTS		; return to system
DATA	TXT	'Hello'	; data segment ...
	HEX	20	
	TXT	'world!'	
	HEX	00	

The steps to write and compile C=64 Merlin Assembly are given in Table XXXV. The program does three things: First it sets the border color to white by copying 1 to kernal address \$D020. Then it sets the cursor to the


```

1  *HELLO WORLD
2  ORG $8000
3  CHROUT EQU $FFD2
4  SETCUR EQU $FFF0
5  BRDCOL EQU $0020
6
7  STA H1
8  LDV BRDCOL
9  LDV #0
10 CLC
11 JSR SETCUR
12 LDV #0
13 LOOP LDA DATA,X
14 BEQ DONE
15 JSR CHROUT
16 IMX
17 JMP LOOP
18
19 DONE RTS
20 DATA TXT 'Hello'
21 HEX 20
22 TXT 'world!'
23 HEX 00

```

```

: MON
$80001

0000-A9 01 LDA W$001
0001-8D 20 STA $0020
0002-A2 00 LDV $800
0003-AD 00 LDV $800
0004-18 CLC
0005-20 F0 JSR $FFF0
0006-A2 00 JSR W$00
0007-BD 1C LDA $801C,X
0008-7F 07 BEQ $8018
0009-20 D2 JMP $FFD2
0010-E8 IMX
0011-4C 0F JMP $800F
0012-68 RTS
0013-68 PLA
0014-43 4C FOR $4C
0015-47 4F JMP $204F
0016-57
0017-3F
0018-52
0019-4C JMP
0020-44 21 $2144
0021-4C
$

```

```

Hello world!
$ 4 SETCUR
5 BRDCOL
6
7
8
9
10
11
12
13 LOOP LDA DATA,X
14 BEQ DONE
15 JSR CHROUT
16 IMX
17 JMP LOOP
18
19 DONE RTS
20 DATA TXT 'Hello'
21 HEX 20
22 TXT 'world!'
23 HEX 00

```

```

: MON
$8000g

```

Figure 104: A 6510 Assembly program writing "Hello world!" on the Vice Commodore 64 emulator running the Merlin 64 assembler environment. Second panel: program in memory. Third panel: state of the screen after running the program.

top left of the screen by a call to kernal subroutine at \$FFF0 with registers X and Y equal to 0 and the carry flag cleared by CLC (carry flag set would read the cursor position instead of setting it). And then it prints, in a loop,

Table XXXVI: C=64 Merlin Assembler commands

Environment	Prompt	Command	Meaning
Disk	%	L	Load source file
		S	Save source file
		E	Editor/assembler
		X	Disk command
Editor	:	A	Add code
		L	List
		D#	Delete line
		I#	Insert before line
		E#	Edit line
		ASM	Assemble
Monitor	\$	<addr>l	List memory
		<addr>g	Run at address

character by character a text "Hello world!" (each printed through a kernal subroutine at \$FFD2 until the string-terminator null is found). After the program finishes (DONE RTS at line 18) control is returned to the system, the Merlin assembler monitor in this case, and the \$ prompt and cursor ■ reappear. (Third panel of Fig. 104).

11.4 Atmel AVR

AVR is a typical example of a micro-controller unit (MCU). They are characterized by a simple and small design, low specifications, but therefore very cheap, and specialized in signal acquisition and processing and rudimentary processing. These are the typical controllers in consumer electronics (washing machines, etc.). The AVR Atmel is used in the famous Arduino series, which is the reason why it is presented here. The chip comes in various packages ranging from dual-inline (DIL) to surface-mounted device (SMD), various memory sizes and clock speeds.

An example is the ATmega32 (32 kB memory) shown in Figure 105. As you can see, the μ processor is a full machine in that it has everything that is needed on-board. A description can be done on basis of the pins:

- Internal memory only, so there are no data pins or address pin, as there is no external data bus or address bus.
- Digital input/output pins (24): PB0...PB7, PC0...PC7 and PD0...PD7 to which digital signals (0/1) can be connected. These can either be

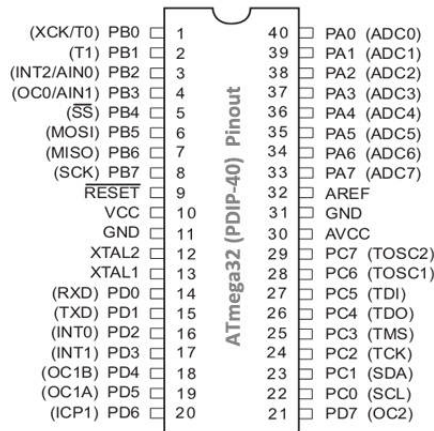


Figure 105: ATmega32 pinout. (Wikipedia).

sensors or actuators because each can independently be configured for input or output.

- Some of the digital pins have special purpose too. For example PD0 and PD1, they serve as a (pseudo)RS232 serial communications port*: receive (RXD) and transmit (TXD). These are also used to upload the code to the internal memory.
- The digital pins are accessible through overlay techniques, as part of register memory (see below).
- Analog pins: These are input only (10-bit ADC, analog-to-digital-converter). They convert an input voltage into a number in the range from 0 (0 volt) to 1023 (VCC or AREF).
- - VCC and GND: power supply
 - XTAL1 and XTAL2: connections for an external quartz crystal to define the clock frequency
 - $\overline{\text{RESET}}$: (not) reset
 - AREF: reference voltage for ADCs (defining the 1023 value)
 - AVCC: power supply for ADCs.

The architecture is 8 bit (registers and internal data bus). The instruction set is RISC with the AVR Assembly having about 100 different instructions. And it is running on a relatively low frequency, of the order of 1 MHz.

*Pseudo-RS232 because the logic levels are 0 and +5 V instead of the RS232 levels of +12 V and -12 V respectively

The unconventional novelty in the Atmel architecture is that it has several types of memory that moreover use overlay techniques (see the section on overlays in Chapter 7). The main memory is logically made up of

- 32 8-bit registers for general purpose, which resemble a little the zero-paging technique of the MOS 65xx processor.
- 64 I/O lines, including the PB0...PB7, PC0...PC7 and PD0...PD7. Writing to and reading from these memory addresses is equal to getting access to the outside world by overlay techniques.
- 2 kB of SRAM (like normal RAM, but static and not dynamic).
- Registers, I/O space and SRAM form a continuous memory space. Addresses from 0x0000 to 0x085F.
- Flash memory: This is like a conventional (but built in) hard disk (of typically 32 kB). In this memory segment the addressing distance is 16 bit (2 bytes); A 32 kB disk has 16 kW (kiloword) size. The program resides in this part of the memory. It is not erased on power cycles.
- On top of it there is EEPROM memory (typically some kB), accessible through 4 memory addresses in I/O space, two for the EEPROM address, one for the EEPROM data, and one for a read or write strobe that synchronizes I/O memory data with EEPROM memory data. This memory is also not erased by power-off and typically stores the 'settings' of the equipment. For example whether we want centigrade (°C) or fahrenheit (°F) on our thermometer.
- The program counter points to an address in flash memory. Flash stores only one program. The addresses are from 0 to 16 k, so 14 bits address from 0x0000 to 0x3FFF.
- A status register:

I	T	H	S	V	N	Z	C
---	---	---	---	---	---	---	---

C:	carry	S:	sign
Z:	zero	H:	half-bit carry
N:	negative	T:	test
V:	overflow	I:	interrupt.

- 13 different addressing modes, addressing to 8-bit data memory or 16-bit program memory.
- Although still RISC, the instruction set is large and can be found in Appendix C. Noteworthy are stack instructions (POP and PUSH), I/O instructions (IN and OUT), and the absence of floating-point instructions. The reader is advised to download the AVR Instruction Set Manual from Atmel, which is readily available on the internet.

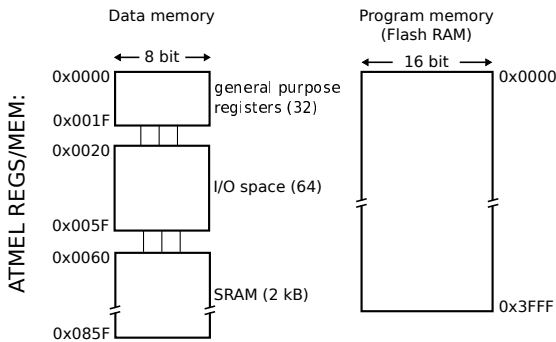


Figure 106: Atmel memory organization.

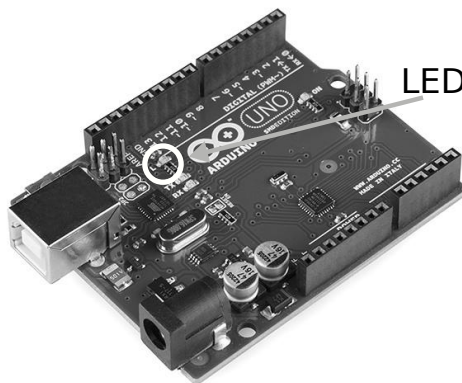


Figure 107: Arduino Uno using an AVR processor; the LED used in the blink example indicated. (Adapted from an image at Wikipedia).

Writing programs in AVR Assembly is a little more tricky, because the AVR devices do not have a display to output data. However, we can show here how it works, by reproducing the famous 'blink' example of an Arduino. Arduino is an excellent developers and teaching platform based on the AVR micro-controllers, see Figure 107. Moreover, it is an open-source engineering project and as such fully inline with the spirit of this book. If you are not yet familiar with Arduino, try to get a hold of one.

Arduino has its own IDE where programs can be written and tested in C++. The famous test-if-it-works, the equivalent of 'hello world' in other environments, is the blink example. It does nothing more than blinking an onboard LED, connected to digital port 13 (In Figure 107 the LED is indicated by 'L', close to pin 13, pin 13 is directly wired to port B5 of the Atmel processor). We are here going to write this same example in AVR

Assembly and upload it to the Arduino, without the help of the Arduino IDE.

We first write our Assembly source code. It is shown here below:

```

;-----;
; blink.asm ;
; blinks an LED which is connected to PB5 (Arduino pin 13) ;
;-----;
; avra blink.asm
; avrdude -p atmega328p -c arduino -b 115200 -P /dev/ttyUSB0
; -U flash:w:blink.hex:i

; .include "m328Pdef.inc"
; or directly define:
.device ATmega328P
.equ PORTB = 0x05
.equ DDRB = 0x04
.equ PORTB5 = 5

setup:
    ldi r16,0b00100000
    out DDRB,r16 ; pin 5 of port B is output

loop:
    sbi PORTB,PORTB5 ; set bit 5 of I/O port B
    call delaycounter ; software delay
    cbi PORTB,PORTB5 ; clear bit 5 of I/O port B
    call delaycounter ; software delay
    rjmp loop ; eternal loop

delaycounter:
    ldi r29,0x20
    ldi r30,0xFF
    ldi r31,0xFF
delayloop:
    dec r31
    brne delayloop
    dec r30
    brne delayloop
    dec r29
    brne delayloop
    ret

```

It works as follows: The first thing to do is to include the specifications of the processor we are going to use. In this case an ATmega 328P, so we include the definitions file for it, `m328Pdef.inc`. Constants, such as `DDRB` used later, are now defined. Alternatively (as shown) we can directly define the relevant constants we are going to use by a couple of `.equ` assembler directives.

The first real AVR Assembly instruction is `ldi r16,0b00100000`, which means load-immediate the 8-bit binary value `00100000` into register `r16`. The next instruction, `out DDRB,r16`, sets the direction – 0 is input, 1 is

output – of the digital ports B (pins 8 to 15 on the Arduino) by writing this bit pattern in `r16` to the overlay address of Port B. With only the third bit set (sixth from the right), only B5 is output, the rest is input. The next instruction, `sbi PORTB,PORTB5`, sets bit 5 of port B (to 1, meaning 5 volt). Thus, after this instruction, pin 13 of the Arduino is high (5 volt), and through a shunt resistor current is supplied to the LED which thus lights up.

We now have to wait for some time before switching it off again. In this simple example we just call a routine (`delaycounter`) that implements a software down counter and calculate how many cycles it needs to have the desired delay before continuing. In this case, we used a down-counter from `0x20ffff`, stored in three registers, `r29`, `r30` and `r31`. After the counter reaches zero, the function is exited (`ret`) and pin 5 of port B cleared which switches off the LED. Another (equal) delay and an unconditional jump is made to the beginning of the eternal loop. Each cycle of the software counter takes two instructions (decrement register 31 and a branching instruction), so on a 16 MHz processor the software delay takes in total $2 \times 0x20ffff / (16 \text{ MHz}) = 0.26 \text{ s}$. The blinking rate is thus about 2 Hz.

We now need to assemble our program and upload it to our AVR processor. For the assembler we can use the AVRA assembler of Ro5bert available at the Software Manager of Linux, or on GitHub. After 'making' the AVRA project (`make install`) we can compile our AVR Assembly program with

```
avra blink.asm
```

If things go well it will respond with something like (an irrelevant warning omitted here)

```
AVRA: advanced AVR macro assembler (version 1.4.2)
Pass 1...
Pass 2...
done
```

```
Assembly complete with no errors.
```

```
Segment usage:
```

```
Code      :      19 words (38 bytes)
Data      :          0 bytes
EEPROM    :          0 bytes
```

Now we have an Intel Hexadecimal machine code file, `blink.hex`:

```
:020000020000FC
:10000000000E204B92D9A0E9409002D980E9409006F
:10001000F9CFFFEFEFEFD0E2FA95F1F7EA95E1F7CC
:06002000DA95D1F7089506
:00000001FF
```

This we have to upload to the AVR processor. A nice program is `avrdude` (AVR Downloader/UploaDEr) of Brian S. Dean, available at nongnu.org.

Note that it is also part of the Arduino environment, so you may already have it installed on your computer. If not, run the `configure` file, and check if all dependencies are met (install those missing, esp. the `-dev` packages at the Synaptic Packet manager).

The easiest way of finding out the exact command of uploading our machine-code compiled program, however, is by using the Arduino environment and making sure we have enabled the "verbose output during upload" in the preferences. In that case we can just take the blink example (or any other program) and upload it to an Arduino. The Arduino environment will inform us of the command-line `avrdude` instruction. We can adapt it then to our own hex-file and write something like

```
avrdude -p atmega328p -c arduino -b 115200 -P /dev/ttyUSB0
-U flash:w:blink.hex:i
```

It specifies the processor (`atmega328p`) and the bitrate of transfer (115200 baud), as well as the programmer type (`arduino`). If everything went well, the LED is blinking at about 2 Hz. The frequency can be adjusted by the value loaded in register `r29` of the code. In case you do not have the Arduino environment, but do have `avrdude`, other commands might work, such as `avrdude -p m328P -c stk500v1 -b 57600 -P /dev/ttyUSB0 -U flash:w:blink.hex`. It is a little more stabbing in the dark, though.

11.5 Intel x86

The x86 family of processors were all based on the 4004 series and is named on basis of the 8086 processor and successors. It is part of a series: 4004 → 4040 → 8008 → 8080 → 8085 → **8086** → 80286 → 80386 → 80486 → Pentium. (They decided to start giving non-numerical names like Pentium to the chips to be able to register it as a trademark; you cannot register a number). They are made most famous by the IBM 'personal computer' (PC) series that also includes the AT and XT models. In principle the processors in this series tried to keep backward compatibility – from the 8086 onwards – meaning that the same compiled program would flawlessly run on a more-recent processor. That is, of course, both its strength and its weakness. Tricks have been used to achieve it.

It was designed by Intel, but other companies made compatible versions. AMD, Cyrix, etc. Sold first in 1978 with a 5-MHz clock and consisting of 29,000 transistors in a 40-pin-DIL package. It is still the most used architecture for personal computing, as in most modern desktop and laptop computers.

Because this family of processors spans decades, the features evolved a lot. But starting at:

- 16-bit registers.

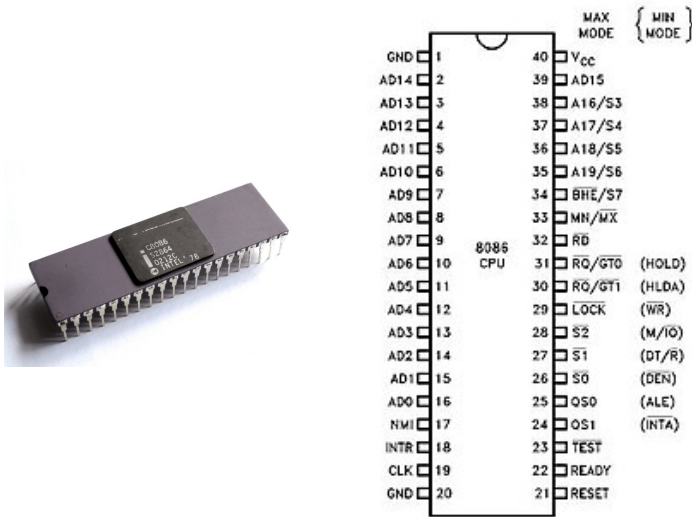


Figure 108: Intel 8086 40-pin-DIL chip and pin-out. (Wikipedia). GND: Ground, AD0-AD15: Multiplexed data/address lines. AD16-AD19: MSb address. INTR: (hardware) interrupt request. NMI: Non-maskable interrupt. CLK: Clock. RESET: Reset computer. RD: (Not) read. READY: Acknowledgment signal from I/O devices (input). TEST: If 8087 co-processor is busy, this input signal is high. VCC: Power supply. MN/MX: Mode. INTA: (Not) interrupt acknowledge (output). ALE: (Valid) address available on address bus (output). M/IO: Hardware overlay (see page 176) or memory access (output). WR: Write (output). HOLD: DMA needs to use the address bus (input). HLDA: Hold acknowledge signal to DMA (output). DEN: Data enable.

- 16-bit data bus. Pins 2-16 and 39 of the chip shown in Figure 108. These are multiplexed also as address lines. During the first clock cycle the lines act as address and in the remaining cycles they carry data.
- 16-bit address bus, but later versions used memory segmentation to add 4 bits to the address bus (pins 35-38). (Today we still can talk of a segmentation fault, when the computer completely messes up and we get a so-called 'blue screen of death', BSOD). An address (offset) was used in the code is 16 bit, but relative to a segment that was left-shifted 4 bits. The final address was then the sum of the two. An example:

```
0110 1000 1000 0111      segment
0011 0100 1010 1001 offset
```

```
----- +
0110 1011 1101 0001 1001 address
```

That leaves 2^{20} address space (of 8 bits each) = 1 MB.

- 14 16-bit registers
 - general purpose: AX, BX, CX, DX (also addressable half/byte: AH, AL, etc.)
 - SP: stack pointer, BP: base stack pointer
 - SI, DI: address pointers
 - IP: instruction pointer (program counter)
 - CS: code segment. DS: data segment. ES: extra segment. SS: stack segment
 - Status register (carry, parity, auxiliary, zero, sign, trap, interrupt, direction, overflow).

Later versions:

- The latest version has a 48 bit address bus ($2^{48} = 256$ TB), 8 billion transistors at 11 nm MFS, 4 GHz clock and more than 1000 pins.
- In 32-bit x86 architectures, the register names add an 'E' in front, so EAX, etc.
- In 64-bit x86 architectures, the register names add an 'R' in front, so RAX, etc.
- There existed 8087 co-processors (obsolete after 80486) implementing IEEE 754 with 8 80-bit registers (extended format).
- Later: Protected mode (vs. real mode). Good for multiprocessing. Prevents programs from corrupting one another (or even crash the system; blue-screen of death cannot happen).
- MMX technology. It uses the x87 co-processor for integer processing for graphics calculations. These are 64-bit integers in 80-bit registers (if MSB 16 bit are set to 1 then it is NaN). This was made obsolete by SSE technology. Streaming SIMD Extensions (SSE) is a single instruction, multiple data (SIMD) instruction set extension.
- The 80586 was named 'Pentium' because you cannot copyright-protect a number.

x86 Assembly is a CISC architecture with a lot of instructions (about 1290) with a lot of addressing modes and variable instruction length (ranging from 1 to 15 bytes). They are too numerous to mention all here, but some deserve to be highlighted:

- Special stack instructions: `push`, `pop`, `call`, `ret`, `enter`, `leave`. Compared to MIPS: `call` is like `jal`, `ret` is equivalent to `jr $ra`, while `enter` and `leave` take care of pushing and popping registers to the stack by creating and destroying a stack frame.
- ALU instructions:
 - arithmetic: `add`, `sub`, `mul`, `idiv`
 - logic: `and`, `or`, `xor`, `neg`
 - shift and rotate: `sal`, `sar`, `shl`, `shr`, `rol`, `ror`
 - floating point is relative to stack: `fadd st(0)`, `st(1)` is pop two items from stack, add them and put result on stack
also: `fsin`, `fcos`, `ftan`, `fatan`, `fexp`, `fyl2x` (`log2`).
- data flow: `mov`.
- jump: unconditional: `jump`. conditional: `jz`, `jnz`, `jl`, `jg`, `ja`.
- flag: `cmp`.
- software interrupts: `int`.

Because the same architecture was used for many decades, and it evolved from 16 bit to 64 bit, the register names are a little confusing. In 64-bit there are 16 registers and they are called

```
R0  R1  R2  R3  R4  R5  R6  R7  R8  R9  ... R15
RAX RCX RDX RBX RSP RBP RSI RDI
```

(The first 8 of the registers have alternate names written below). The lowest 32 bits of each register, or the same registers in older 32-bit technology, have names like this:

```
R0D R1D R2D R3D R4D R5D R6D R7D R8D R9D ... R15D
EAX ECX EDX EBX ESP EBP ESI EDI
```

The lowest 16 bits of each register, or the same registers in older 16-bit technology, have names like this:

```
R0W R1W R2W R3W R4W R5W R6W R7W R8W R9W ... R15W
AX  CX  DX  BX  SP  BP  SI  DI
```

The lowest 8 bits of each register have names like this:

```
R0B R1B R2B R3B R4B R5B R6B R7B R8B R9B ... R15B
AL  CL  DL  BL  SPL BPL SIL DIL
```

The 'second byte' (bits 8 to 15) of R0 to R3 are accessed through

AH CH DH BH

Addressing is done by square parenthesis. So the operand in the addressing

`[rax-rdi*4]`

is the contents of the address pointed to by register **RAX** with an offset of -4 times the contents of register **RDI**.

Numbers can be written as

300	decimal
300d	decimal, d suffix
0d300	decimal, 0d prefix
06ch	hexadecimal, h suffix
0x6c	hexadecimal, 0x prefix
100q	octal, q suffix
0q100	octal, 0q prefix
11000001b	binary, b suffix
0b1100001	binary, 0b prefix

Let's write some Assembly code for it. It can be done in many ways. Assuming that your work computer is one based on the x86 architecture, you do not need any emulator. Neither of the processor (as our MIPS emulator in MARS), nor of the entire computer (as was done above for the MOS 6510 system the Commodore 64). You can directly run your program on the computer, without emulation. Write a program in an assembly language of choice, implemented by a specific assembler, and compile (assemble) the program and run it. The instruction set of x86 is too big to show here; the reader is advised to get a hold of the Intel document "Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 2 (2A, 2B, 2C & 2D): Instruction Set Reference, A-Z".

Here is an example program of writing "Hello world!" in early Microsoft Macro Assembler (MASM) x86 language. In this, 'h' is hexadecimal, ';' starts comment, 'db' is data size byte (8-bit), 0x21 is the MS-DOS interrupt for I/O (See Appendix F), among other things:

```

; -----;
; Writes "Hello world!" to the console using ;
; MS-DOS interrupt 0x21. Runs on MS-DOS only. ;
; -----;
section .data
msg db 'Hello world!$'; $-terminated string

section .text
start:
    lea dx, msg ; load effective address

```

```

mov ah, 09h ; 0x09: write string to stdout
int 21h     ; interrupt 0x21; DOS interrupt
mov ah, 4Ch ; 0x4C: terminate
int 21h
end start

```

Since it is making use of an MS-DOS interrupt (0x21, see Appendix F), MS-DOS must be loaded in memory. It will not run on other operating systems running on this x86 hardware. For instance, on Linux, using `int 0x21` will cause

Segmentation fault (core dumped)

This book will not delve much into the Microsoft operating system, for two reasons. First because it is a closed platform and thus against the spirit of stallinga.org. Moreover, by the time you read this, Microsoft Windows will be history and converted into merely yet-another-Linux-distro where all open-source and public-domain software will work, including the following examples. A Linux version for interrupt-generated text output "Hello world!", using specific Linux system calls (software interrupt 0x80; kernel system call, see p. 320) is the following.

In Linux we can use NASM (Netwide Assembler) to assemble the program and use 32-bit registers and interrupt 0x80. The following program will *only* run in Linux x86 and will not run in Windows or MS-DOS even if it has underlying x86 hardware!

```

; -----;
; file: hello.asm ;
; Writes "Hello, World!" to the console using ;
; only interrupt 0x80. Runs on 64-bit Linux only. ;
; To assemble and run: ;
; nasm -felf64 hello.asm ;
; ld hello.o ;
; ./a.out ;
; -----;

global _start

section .data
message: db "Hello, World!", 10 ; note the LF at the end

section .text
_start:
mov eax, 4 ; 4 = write string
mov ebx, 1 ; 1 = stdout
mov ecx, message ; address of string start
mov edx, 0xE ; length of string
int 0x80

mov eax, 1 ; 1 = exit

```

```

mov ebx, 0          ; exit code 0
int 0x80

```

Output:

```
Hello, World!
```

We could also use a more user-friendly syscall method, similar to what we are used to in MIPS. See the example below. Note that the `int 0x80` with `eax` equal to 4 in the example above becomes `syscall` with `rax` equal to 1; not only did we go from 32 to 64 bit architecture, but also the parameter to place in `ax` changed from 4 to 1. Note also the different registers used in passing the arguments of string address, string length, and output file:

```

; -----;
; file: hello.asm                                ;
; Writes "Hello, World!" to the console using    ;
; only system calls. Runs on 64-bit Linux only.  ;
; To assemble and run:                          ;
;   nasm -felf64 hello.asm                      ;
;   ld hello.o                                  ;
;   ./a.out                                     ;
; source: https://cs.lmu.edu/~ray/notes/nasmtutorial/ ;
; -----;

        global  _start

_start:  section .text
        mov     rsi, message ; address of string to output
        mov     rax, 1       ; system call for write
        mov     rdi, 1       ; file handle 1 is stdout
        mov     rdx, 0xD     ; number of bytes
        syscall              ; invoke operating system to
                             ; do the write
        mov     rax, 0x3C    ; system call for exit
        xor     rdi, rdi     ; exit code 0
        syscall              ; invoke operating system to exit

        section .data
message: db      "Hello World!", 10 ; note the LF at the end

```

Output:

```
Hello World!
```

Even more confusing, we could also use a different assembler with a different syntax to write the same code. Here is an `as` (the portable GNU

assembler) version of the int-0x80 program (code by [Ciro Santilli](#) on [stack-overflow](#)):

```

/*****\
* file: hello.asm                                *
* Writes "Hello, World!" to the console using *
* interrupt 0x80. To assemble and run:         *
*   as -o hello.o hello.asm                    *
*   ld hello.o                                  *
*   ./a.out                                     *
*****/

.data
    s:
        .ascii "hello world!\n"
        len = . - s
.text
.global _start
_start:

    movl $4, %eax /* write system call number */
    movl $1, %ebx /* stdout */
    movl $s, %ecx /* the data to print */
    movl $len, %edx /* length of the buffer */
    int $0x80

    movl $1, %eax /* exit system call number */
    movl $0, %ebx /* exit status */
    int $0x80

```

Output:

```
hello world!
```

Note the different syntax for comment, the use of the \$ sign, the 32-bit `movl` instruction, the reverse order of source and destination operands, the % symbol, the declaration of a string constant, and the determination of the length of the string.

Another way of having an Assembly program generate output is by borrowing functions from the C libraries. An example is the use of `printf`, which has a further advantage that we can also print other quantities like integers (`syscall` can only print strings). This is shown in the next example:

```

; -----;
; file: print.asm                                ;
; Writes "Hello, World!" and "123" to the console ;
; using external C-function printf. Runs on 64-bit ;
; Linux only.                                    ;
; To assemble and run:                          ;

```

```

;      nasm -felf64 print.asm                                ;
;      gcc -no-pie print.o                                  ;
;      ./a.out                                              ;
; -----;

        global      main          ; entry point for c-compiler
        extern      printf        ; will be borrowed from c-library
                                   ; printf(char* format, <data>)
                                   ; printf(rdi, rsi)

main:
        section     .text

        lea         rdi, [rel message] ; "Hello, World!"
                                   ; no need for rsi
                                   ; or (alternative):
        lea         rdi, [rel fmts]    ; "%s"
        lea         rsi, [rel message] ; "Hello World!"

        mov         al, 0             ; no SSE regs used
        call        printf

        lea         rdi, [rel fmti]   ; "%d\n"
        mov         rsi, 123           ; int to print
        mov         al, 0             ; no SSE regs used
        call        printf

        mov         rax, 0x3C         ; system call for exit
        xor         rdi, rdi          ; exit code 0
        syscall                    ; invoke operating system to exit

        section     .data
message: db          "Hello, World!",10,0; don't forget to terminate
fmti:   db          "%d",10,0         ; format for printing int
fmts:   db          "%s",0           ; format for printing string

```

Output:

```

Hello, World!
123

```

Note how now the gcc linker/compiler is used to generate the executable. We also have to supply the C entry point `main`. We know how `printf()` takes (at least) two arguments: a pointer to a string specifying the format, and then the data to print. In Assembly they go to `RDI` and `RSI` respectively. The actual printing is done by a `call printf`, which we borrow from the C library and should thus be declared external by `extern`. Since for printing "Hello World!" we need only the format string, we can do without `RSI`, or we can specify string (`%s`) in the format `RDI` and supply the text in `RSI`.

We can also do the opposite, use Assembly inside our C-code. (Note that the C code is then of course no longer independent on architecture

since it will use architecture specific instructions). The program below gives an example.

- We have to put every line of Assembly code inside an `'asm("...");'` construction.
- Global variables are on the heap, and accessible through a `[.]` reference ('contents of address'), while local variables and function arguments are on the stack and accessible through the stack pointer `rsp`. As an example, in the function `foo`, the integer `i` is the last item placed on the stack and has size 4, so it is at address `rsp-4` and its value is `[rsp-4]`. Curiously, the integer `x` is at `rsp-20` (one would have expected it to be at `rsp-8`, the difference is likely due to memory alignment).
- Return values, if any, have to be placed in `rax`.

```

/*****
 * file: asm.c
 * How to use in-line x86 Assembly in C
 * gcc -no-pie -masm=intel asm.c
 * ./a.out
 *****/

#include <stdio.h>

int a, b; // global vars are on the heap

int foo(int x){ // arguments are on the stack
    int i = 4; // local vars too
    asm("mov rax, [rsp-20]"); // int x;
    asm("mov rcx, [rsp-4]"); // int i;
    asm("mul rcx"); // rax = i*x;
    asm("mov rcx, [a]"); // global int a;
    asm("mul rcx"); // rax = a*i*x;
    asm("mov rcx, [b]"); // global int b;
    asm("mul rcx"); // rax = b*a*i*x;
    // return value is in rax
    // stack adjusted by gcc and 'ret' is added by gcc
}

void printstring(char *q){
    asm("mov rsi, [rsp-8]"); // q*
    asm("mov rax, 1");
    asm("mov rdi, 1");
    asm("mov rdx, 0xD");
    asm("syscall");
    // stack adjusted by gcc and 'ret' is added by gcc
}

int main(int argc, char **argv){

```

```

char p[] = "Hello world!\n";
int y;

a = 10;    b = 2;
y = foo(3);
printstring(p);
printf("foo return value: y = %d\n", y);

return 0;
}

```

Output:

```

Hello world!
foo return value: y = 240

```

As a curiosity, we can also do the opposite: instead of translating assembly language to machine language, we can also translate machine language back to Assembly. To give an example, the `nasm` program of page 287, translates and links into an executable file (`a.out` in Linux) that starts with something like this in hexadecimal format:

```

7F 45 4C 46 02 01 01 00 00 00 00 00 00 00 00 02 00
3E 00 01 00 00 00 00 10 40 00 00 00 00 40 00 00 00

```

or in ASCII format it should start with `.ELF...`, indicating it is an Executable and Linkable Format file, so that Linux operating system knows what to do with it. It will load the code segment and data segment into memory and set the program counter to the start of code segment. Disassembling can now be done with the command

```
objdump -d a.out
```

which results in

```

a.out:      file format elf64-x86-64

Disassembly of section .text:

0000000000401000 <_start>:
 401000: 48 be 00 20 40 00 00  movabs $0x402000,%rsi
 401007: 00 00 00
 40100a: b8 01 00 00 00      mov     $0x1,%eax
 40100f: bf 01 00 00 00      mov     $0x1,%edi

```

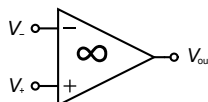
401014:	ba 0e 00 00 00	mov	\$0xe,%edx
401019:	0f 05	syscall	
40101b:	b8 3c 00 00 00	mov	\$0x3c,%eax
401020:	48 31 ff	xor	%rdi,%rdi
401023:	0f 05	syscall	

which is quite close.

The reader is suggested to read books such as Sivarama P. Dandamudi, Guide to Assembly Language Programming in Linux (Springer, ISBN 978-0387-25897-3), and Microsoft Macro Assembler Reference Manual Programmers Guide by Microsoft Corporation. There are not ready-made answers to x86 assembly programming, since it is a very dynamic architecture and things change rapidly. What is written today, tomorrow may no longer be valid.

11.6 Analog computing

We know computers as basically digital computers. Meaning that they crunch *integer* numbers. The world has turned digital, and we can nearly no longer imagine another world. However, not so long time ago, the world was basically analog. An example is radio and television. Television signals were primarily analog. Would it be possible to build an analog computer as well? The answer is yes. As an example may serve the ones based on operational amplifiers (opamps for short).



For our purpose we can imagine them as ideal, meaning that no currents into the amplifier and that they will amplify infinitely whatever difference voltage is supplied at the input:

$$\text{out} = \infty \times (\text{in}_+ - \text{in}_-).$$

This seems rather fruitless if not for the fact that we can use feedback (connecting the output somehow to the input) in which case (when the feedback is to the in- terminal) the difference will be zero, because:

$$(\text{in}_+ - \text{in}_-) = \text{out} / \infty = 0.$$

The amplifier will do the internal 'calculation' to make the difference between the inputs nil. In other words, for negative feedback,

$$\text{in}_+ = \text{in}_-.$$

We can then read the result of the calculation at interesting points of the circuit. We build such analog computers by connecting calculating elements to it. These all implement functions. Don't forget that (analog) electronic components all have a function between current and voltage. An example is a resistance R that has a current linear proportional to the applied voltage:

$$I_R(V_R) = f(V_R) = V_R/R,$$

or the inverse function,

$$V_R(I_R) = f(I_R) = RI_R.$$

Another component is a diode that has an exponential behavior

$$I_D(V) = f(V_D) = I_0 \exp\left(\frac{V_D}{(26 \text{ mV})}\right),$$

or the inverse function,

$$V_D(I) = f(I_D) = (26 \text{ mV}) \times \ln\left(\frac{I_D}{I_0}\right).$$

Let's use these two components to calculate something. A circuit is given in Figure 109. The $+$ -input of the opamp is connected to ground and is thus at $V = 0$. assuming we have an ideal amplifier and with the negative feedback, we have that also the minus-terminal must be at $V = 0$. We call this virtual ground. The diode has one leg at V_{in} and at the other leg at $V = 0$. We can calculate the current according to the above equation. This current does not enter the opamp and must flow through the resistance that thus has a voltage drop equal to the above equation. With one leg at (virtual) ground, the otehr leg must be at

$$V_{\text{out}} = 0 - RI_0 \times \exp\left(\frac{V_{\text{in}}}{(26 \text{ mV})}\right).$$

Our analog computer can thus calculate the exponential function. With other components we can do more advanced things. Even solve differential equations. The reader is invited to read teh author's book about Electronic Instrumentation.

11.7 Advanced architectures: Quantum computing and asynchronous (clockless) computing

For the architectures above, specifically the electronic ones, the memory and states are binary. That is, each memory element has two possible

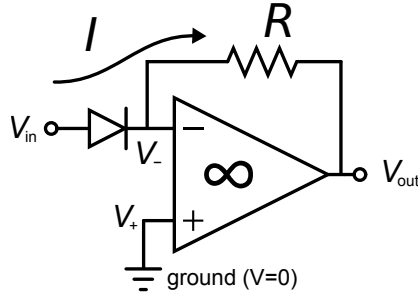


Figure 109: An analog computer based on an operational amplifier. The diode and the resistance have well defined functional relations between voltage and current and this defines the overall relation between V_{in} and V_{out} .

states, which we can label '0' and '1' for convenience. For instance, in TTL (transistor-transistor logic) a '0' is 0 volt and a '1' is 5 volt. In CMOS the voltages may be different (3.3 volt), but invariable there are two states to every bit of memory and the logic hardware processes these binary bits of information.

Not so for quantum computing bits. The idea of quantum mechanics is that a state is only defined once it is being measured, or 'observed' as it is called in jargon. It is not so that the state exists, but only is *unknown* to us until we determine it. No, the state is not existing until observed. Or better to say, it is in a state of superposition of all possible states until measured. At which moment the state 'collapses' into the one remaining possible state. Don't worry if you think it is weird, or that you don't understand it. Even Feynman famously said "If you think you understand quantum mechanics, you don't understand quantum mechanics". The reader is highly recommended to watch his Character of Physical Nature lectures.

The basic ingredient of quantum computing is that a bit, when not observed, is in a superposition of states, we call this a 'qubit'. Now the trick in quantum computing is to process the bits *before* they collapse into the final state. In this way, in contrast to a binary bit that can store only two states, a qubit can store more than two states. So, imagine we have a quantum AND-gate with two qubits. At the entrance of the AND-gate we have four possible states, but they exist all at the same time, since they have not collapsed yet.

We would *simultaneously* do calculations on all possible states! Imagine a bit is related to the spin state of an electron in a magnetic field, '0' is spin-up (\uparrow), and '1' is spin-down (\downarrow). One such bit would have two possible states, we call these wave functions as a superposition of states, written in bra-ket notation as

$$a = \alpha |\uparrow\rangle + \beta |\downarrow\rangle, \quad (1)$$

with the parameters α and β values between 0 and 1 representing the superposition state, and with a sum $\alpha^2 + \beta^2 = 1$. (Collapsing the wavefunction would imply one of the parameters to become 1 and the other 0). Applying two of these wavefunctions, a_1 and a_2 to the qubit AND-gate results in calculating simultaneously all results with all four possible entry 'values' (superposition waves). It is obvious that the computing power rapidly becomes astronomical with the number of entry waves. With n bits we can do 2^n calculations simultaneously. With a simple 100-qubit computer we can do 2^{100} calculations simultaneously, which is about 10^{30} . If we can do one computing step per second, effectively we can do 10^{30} steps every second. That is in terms of processing power 10^{20} 10-GHz cores in parallel. This is a tremendous potential.

The question still remains if it is feasible or not. With the advantages of such a technique in nature directly obvious – imagine being able to calculate the state of the universe for the next ten minutes ahead, including the position and speed of the predator/prey – it is remarkable that the technique is not anywhere implemented in the animal kingdom. Note that other quantum mechanics phenomena do exist in nature. No animal uses quantum computing, even if it would make the species directly masters of the universe. One might come to the conclusion that quantum computing is a nice idea, and even possible to do in some way or another in a laboratory, but not feasible and reliable in reality, or not feasible altogether.



Another odd way of doing computing is with a clockless computer, in so-called asynchronous computing. Remember that we introduced the clock in our gated latch (see Figure 35 on page 93), and never went back to clockless components. From that moment on, all our circuitry was based on synchronized behavior of the components, with a clock as central synchronization unit.

A human brain differs from modern computers in that it does not have a clock to synchronize the logic circuits. Or in other words, the definition of clockless computing: "A digital logic architecture that does not use a central timing clock to synchronize all the circuits in a chip. Called 'asynchronous logic', such an architecture eliminates approximately 15% of the chip's circuits and 20% of its power requirement." On top of that, computation can be faster. Imagine the case of a ripple-carry adder adding two 32-bit numbers. In the worst case, it would take 32 clock cycles. And thus we have to *always* wait 32 cycles for the output, even if the result is ready much earlier, maybe after only two clock cycles. A clockless computer would not need to wait the full 32 cycles, but would simply signal when ready. On average, this can be much faster.

It is obvious why a human brain would do without a clock. Moreover, it solves the two fundamental questions

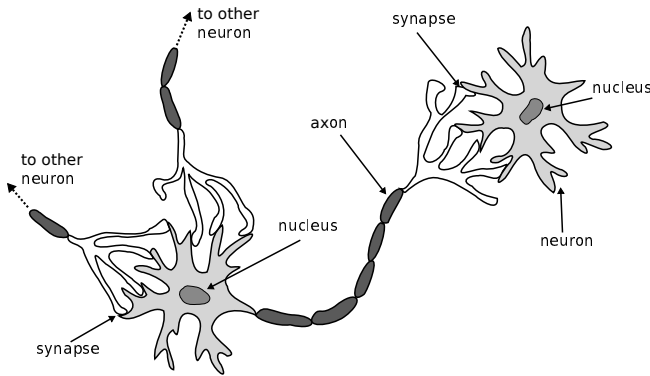


Figure 110: Neurons communicating in a brain. All neurons/axons are both memory and processing, thus overcoming the Von Neumann bottleneck of communication.

- How did a brain evolve? If a clock is part of the brain, it must have been there from day one. It cannot have been that it evolved without a clock and then one day a clock was added to the architecture. A clockless brain, on the other hand, is a natural smooth evolution from mono-cellular entities.
- If a central clock is present, it leaves the brain and body vulnerable to glitches in this clock. Imagine a slight brain injury that damaged the clock, it would immediately kill the human.

It is thus obvious that a clockless architecture is the only real solution in nature. Having said that, the human brain seems to have a 'natural frequency of operation', which is about 11 Hz. (Yes, eleven hertz). Epileptic people can go into a fit when they are exposed to such frequencies, but even fully healthy people are vulnerable to them.

Once a clock has been eliminated from the design, there is no need for a control logic as well. Remember a Von Neumann architecture that is used by all modern computers. A central processing unit, housing an ALU and control logic, with the program and data stored *externally* on memory suffers from a Von Neumann bottleneck, where the CPU has to communicate the information over a bus to the memory, see Figure 60 on p. 130 for a classic Von Neumann computer architecture. A brain does not suffer from this bottleneck, since each neuron and synapse with axons (see Fig. 110) is both the processor as well as the memory. All is memory and all is processing. Although the 'clock' is slow, it is effectively a multi-core processor, with billions of computers working simultaneously, communicating asynchronously with each-other. Calling it a multi-core is an understatement.

Table XXXVII: Number of neurons in various species (source: verywellmind)

Species	Number of neurons
Homo Sapiens	86 billion
Fruit fly	100 thousand
Mouse	75 million
Cat	250 million
Chimpanzee	7 billion
Elephant	257 billion

Table XXXVIII: Specifications of the Summit supercomputer



Item	Spec
Architecture	9,216 POWER9 22-core CPUs 27,648 NVIDIA Tesla V100 GPUs
Operating System	Red Hat Linux
Speed	200 petaFLOPS*
Storage	250 PB
Power	13 MW
Size	873 m ²
Cables	219 km
Location	Lawrence Livermore National Laboratory, California
Price	325 M\$

FLOPS: floating-point operations per second
image source: Wikipedia

A human brain uses only about 20 watt of energy. It has about 100 billion neurons (see Table XXXVII), but still it can perform most tasks much better than the most advanced computer in the world, which at the moment of this writing is the Summit supercomputer used for scientific research with specifications as in Table XXXVIII, that uses 13 megawatt and has 'only' about 40 thousand cores.

In computer science, clockless components are sometimes used and these are called asynchronous circuits. Instead of a clock determining when data

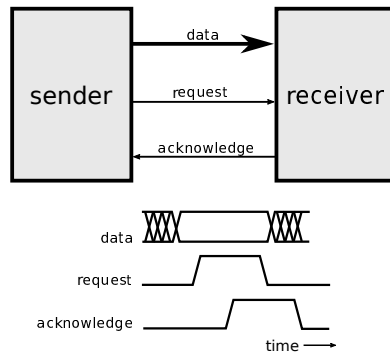


Figure 111: Clockless computing. Components signal each-other when data is ready and when it is received and the receiver is ready to receive more data so that processing can continue.

are guaranteed stable at the output, the circuit uses signals that indicate completion of instructions and operations, specified by simple data transfer protocols. We also call these signals 'handshake' and are often used in long distance communication ('telecom'), where having the same clock signal on both sides of the communication line – i.e., have the two sides synchronized – is technically more complicated. Instead, the sending end asserts a handshake signal 'data ready' or 'request to send' (see Figure 111), and the receiving end replies with a handshake signal 'acknowledge' when the data is received. This acknowledge signal can then be used to trigger the next generation of data by the sender.



In recent years computers have become 'smarter', and products are made that are labeled Artificial Intelligence (AI). Do computers indeed acquire 'intelligence' once they get complicated enough, or do they continue to be mere if-then-else-machines? Advanced, but still not having the 'spark'. Or, reasoning the other way around, we have the essential philosophical question: To what extent are humans simply complicated computers? Alan Turing having thought about that came up with a method to distinguish computers (robots) from humans. This is his so-called Turing test, or imitation game. As Wikipedia writes:

"Turing proposed that a human evaluator would judge natural language conversations between a human and a machine designed to generate human-like responses. The evaluator would be aware that one of the two partners in conversation is a machine, and all participants would be separated from one another. The conversation would be limited to a text-only channel such as a computer keyboard and screen so the result would not depend on the machine's

ability to render words as speech. If the evaluator cannot reliably tell the machine from the human, the machine is said to have passed the test. The test results do not depend on the machine's ability to give correct answers to questions, only how closely its answers resemble those a human would give".

It is very similar to the observation of Descartes:

"If there were machines which bore a resemblance to our bodies and imitated our actions as closely as possible for all practical purposes, we should still have two very certain means of recognizing that they were not real men. The first is that they could never use words, or put together signs, as we do in order to declare our thoughts to others. For we can certainly conceive of a machine so constructed that it utters words, and even utters words that correspond to bodily actions causing a change in its organs. ... But it is not conceivable that such a machine should produce different arrangements of words so as to give an appropriately meaningful answer to whatever is said in its presence, as the dullest of men can do. Secondly, even though some machines might do some things as well as we do them, or perhaps even better, they would inevitably fail in others, which would reveal that they are acting not from understanding, but only from the disposition of their organs. For whereas reason is a universal instrument, which can be used in all kinds of situations, these organs need some particular action; hence it is for all practical purposes impossible for a machine to have enough different organs to make it act in all the contingencies of life in the way in which our reason makes us act". (Translation by Robert Stoothoff).

Many attempts have been made to create human-like robots. A nice example is ELIZA. This 1960s artificial intelligence program basically answered every statement or question of the human with a question of itself, often repeating the statement of the human and simply adding a question mark*:

```
> Hello, I am Eliza.
* hello
> How are you today.. What would you like to discuss?
* I feel happy
> Do you enjoy feeling happy?
* Sure, why not?
> What is it that you really want to know?
* I want to know who you are
> What if you never got to know who you are?
```

*<http://psych.fullerton.edu/mbirnbaum/psych101/Eliza.htm>

This, of course, is not real 'intelligence', but rather a preprogrammed set of rules. Remember that we have seen that in Assembly there is nothing visible of the layers above (imperative programming like C, or object-oriented programming like C++, etc.); the only thing implemented in Assembly is if-then-goto. As such, all 'artificial intelligence' boils down to a (huge) set of if-statements. And these are based on Boolean logic, starting with the Huntington Postulates. Nothing more.

"John Searle's 1980 paper *Minds, Brains, and Programs* proposed the 'Chinese room' thought experiment and argued that the Turing test could not be used to determine if a machine can think. Searle noted that software (such as ELIZA) could pass the Turing test simply by manipulating symbols of which they had no understanding. Without understanding, they could not be described as 'thinking' in the same sense people are. Therefore, Searle concludes, the Turing test cannot prove that a machine can think." (Wikipedia).

Can there one day be a robot that is indistinguishable from humans? Are such units already living among us?

A | Intel 4004 instruction set

Instruction	Mne- monic	1st byte	2nd byte
No Operation	NOP	0000	0000
Jump Conditional	JCN	0001	<i>CCCC AAAA AAAA</i>
Fetch Immediate	FIM	0010	<i>RRR0 DDDD DDDD</i>
Send Register Control	SRC	0010	<i>RRR1</i>
Fetch Indirect	FIN	0011	<i>RRR0</i>
Jump Indirect	JIN	0011	<i>RRR1</i>
Jump Unconditional	JUN	0100	<i>AAAA AAAA AAAA</i>
Jump to Subroutine	JMS	0101	<i>AAAA AAAA AAAA</i>
Increment	INC	0110	<i>RRRR</i>
Increment and Skip	ISZ	0111	<i>RRRR AAAA AAAA</i>
Add	ADD	1000	<i>RRRR</i>
Subtract	SUB	1001	<i>RRRR</i>
Load	LD	1010	<i>RRRR</i>
Exchange	XCH	1011	<i>RRRR</i>
Branch Back and Load	BBL	1100	<i>DDDD</i>
Load Immediate	LDM	1101	<i>DDDD</i>
Write Main Memory	WRM	1110	0000
Write RAM Port	WMP	1110	0001
Write ROM Port	WRR	1110	0010
Write Status Char 0	WR0	1110	0100
Write Status Char 1	WR1	1110	0101
Write Status Char 2	WR2	1110	0110
Write Status Char 3	WR3	1110	0111
Subtract Main Memory	SBM	1110	1000

Bits: *A*: address, *R*: register number, *C*: condition, *D*: data

Instruction	Mne- monic	1st byte	2nd byte
Read Main Memory	RDM	1110	1001
Read ROM Port	RDR	1110	1010
Add Main Memory	ADM	1110	1011
Read Status Char 0	RD0	1110	1100
Read Status Char 1	RD1	1110	1101
Read Status Char 2	RD2	1110	1110
Read Status Char 3	RD3	1110	1111
Clear Both	CLB	1111	0000
Clear Carry	CLC	1111	0001
Increment Accumulator	IAC	1111	0010
Complement Carry	CMC	1111	0011
Complement	CMA	1111	0100
Rotate Left	RAL	1111	0101
Rotate Right	RAR	1111	0110
Transfer Carry and Clear	TCC	1111	0111
Decrement Accumulator	DAC	1111	1000
Transfer Carry Subtract	TCS	1111	1001
Set Carry	STC	1111	1010
Decimal Adjust Accumulator	DAA	1111	1011
Keyboard Process	KBP	1111	1100
Designate Command Line	DCL	1111	1101

Bits: *A*: address, *R*: register number, *C*: condition, *D*: data

B | MOS 65xx instruction set

65xx addressing modes:

mode	description	assembly	operand
a	Absolute	opc, A, B	(BA)
(a, x)	Absolute Indexed Indirect	opc, A, B	((BA+X))
a, x	Absolute Indexed with X	opc, A, B	(BA+X)
a, y	Absolute Indexed with Y	opc, A, B	(BA+Y)
(a)	Absolute Indirect	opc, A, B	((BA))
A	Accumulator	opc	ACCU
#	Immediate	opc, A	A
i	Implied	opc	opc
r	Program Counter Relative	opc, A	A+PC
s	Stack	opc	(SP)
zp	Zero Page	opc, A	(0A)
(zp, x)	Zero Page Indexed Indirect	opc, A	((0A+X))
zp, x	Zero Page Indexed with X	opc, A	(0A+X)
zp, y	Zero Page Indexed with Y	opc, A	(0A+Y)
(zp), x	Zero Page Indirect Indexed with X	opc, A	((0A)+X)
(zp), y	Zero Page Indirect Indexed with Y	opc, A	((0A)+Y)

opc: opcode byte. A, B: next bytes in code segment

BA: 16-bit address. 0A: zero-page address. (...): contents of address

65xx mnemonics:

ADC	add M to A with carry	LSR	logic shift right M or A
AND	logic AND M with A	NOP	no operation
ASL	arithm. shift left	ORA	OR with M or A
BBR	branch on bit reset	PHA	push A to stack
BBS	branch on bit set	PHP	push status on stack
BCC	branch on carry clear	PHX	push X on stack
BCS	branch on carry set	PHY	push Y on stack
BEQ	branch on equal	PLA	pop A from stack
BIT	bit test	PLP	pop status from stack
BMI	branch on minus	PLX	pop X from stack
BNE	branch on not equal	PLY	pop Y from stack
BPL	branch on plus	RMB	reset memory bit
BRA	branch always	ROL	rotate left M or A
BRK	break	ROR	rotate right M or A
BVC	branch on overflow clear	RTI	return from interrupt
BVS	branch on overflow set	RTS	return from subroutine
CLC	clear carry	SBC	subtract from A with borrow
CLD	clear decimal mode	SEC	set carry
CLI	clear IRQ disable bit	SED	set decimal mode
CLV	clear overflow flag	SEI	set IRQ disable
CMP	compare M to A	SMB	set memory bit
CPX	compare M to X	STA	store A in M
CPY	compare M to Y	STP	stop mode
DEC	decrement M or A	STX	store X in M
DEX	decrement X	STY	store Y in M
DEY	decrement Y	STZ	store zero in M
EOR	Exclusive OR M with A	TAX	copy A to X
INC	increment M or A	TAY	copy A to Y
INX	increment X	TRB	test and reset memory bit
INY	increment Y	TSB	test and set memory bit
JMP	jump to	TSX	copy SP to X
JSR	jump to subroutine	TXA	copy X to A
LDA	load A with M	TXS	copy X to SP
LDX	load X with M	TYA	copy Y to A
LDY	load Y with M	WAI	wait for interrupt

A: accumulator. M: memory. X: X register. Y: Y register. SP: stack pointer. IRQ: interrupt request

65xx instruction opcodes:

MSN ↓	LSN															
0	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	BRK s	ORA (zp,x)			TSB zp	ORA zp	ASL zp	RMB0 zp	PHP s	ORA #	ASL A		TSB a	ORA a	ASL a	BBR0 r
1	BPL r	ORA (zp),y	ORA (zp)		TRB zp	ORA zp,x	ASL zp,x	RMB1 zp	CLC i	ORA a,y	INC A		TRB a	ORA a,x	ASL a,x	BBR1 r
2	JSR a	AND (zp,x)			BIT zp	AND zp	ROL zp		RMB2 s	PLP #	AND A	ROL	BIT a	AND a	ROL a	BBR2 r
3	BMI r	AND (zp),y	AND (zp)		BIT zp,x	AND zp,x	ROL zp,x		RMB3 i	AND a,y	DEC A		BIT a,x	AND a,x	ROL a,x	BBR3 r
4	RTI s	EOR (zp,x)				EOR zp	LSR zp	RMB4 zp	PHA s	EOR #	LSR A		JMP a	EOR a	LSR a	BBR4 r
5	BVC r	EOR (zp),y	EOR (zp)			EOR zp,x	LSR zp,x	RMB5 zp	CLI i	EOR a,y	PHY s			EOR a,x	LSR a,x	BBR5 r
6	RTS s	ADC (zp,x)			STZ zp	ADC zp	ROR zp	RMB6 zp	PLA s	ADC #	ROR A		JMP (a)	ADC a	ROR a	BBR6 r
7	BVS r	ADC (zp),y	ADC (zp)		STZ zp,x	ADC zp,x	ROR0 zp,x	RMB7 zp	SEI i	ADC a,y	PLY s		JMP (a,x)	ADC a,x	ROR a,x	BBR7 r
8	BRA r	STA (zp,x)			STY zp	STA zp	STX zp	SMB0 zp	DEY i	BIT #	TXA i		STY a	STA a	STX a	BBS0 r
9	BCC r	STA (zp),y	STA (zp)		STY zp,x	STA zp,x	STX zp,y	SMB1 zp	TYA i	STA a,y	TXS i		STZ a	STA a,x	STZ a,x	BBS1 r
A	LDY #	LDA (zp,x)	LDX #		LDY zp	LDA zp	LDX zp	SMB2 zp	TAY i	LDA #	TAX i		LDY A	LDA a	LDX a	BBS2 r
B	BCS r	LDA (zp),y	LDA (zp)		LDY zp,x	LDA zp,x	LDX zp,y	SMB3 zp	CLV i	LDA a,y	TSX i		LDY a,x	LDA a,x	LDX a,y	BBS3 r
C	CPY #	CMP (zp,x)			CPY zp	CMP zp	DEC zp	SMB4 zp	INY i	CMP #	DEX i	WAI i	CPY a	CMP a	DEC a	BBS4 r
D	BNE r	CMP (zp),y	CMP (zp)			CMP zp,x	DEC zp,x	SMB5 zp	CLD i	CMP a,y	PHX s	STP i		CMP a,x	DEC a,x	BBS5 r
E	CPX #	SBC (zp,x)			CPX zp	SBC zp	INC zp	SMB6 zp	INX i	SBC #	NOP i		CPX a	SBC a	INC a	BBS6 r
F	BEQ r	SBC (zp),y	SBC (zp)			SBC zp,x	INC zp,x	SMB7 zp	SED i	SBC a,y	PLX s			SBC a,x	INC a,x	BBS7 r

MSN: most-significant nibble, LSN: least-significant nibble

C| set

AVR Atmel instruction

AVR arithmetic and logic instructions

Mnemonic	Operands	Description	Operation	Flags	Clk
ADD	Rd, Rr	Add without Carry	$Rd \leftarrow Rd + Rr$	ZCNVH	1
ADC	Rd, Rr	Add with Carry	$Rd \leftarrow Rd + Rr + C$	ZCNVH	1
ADIW	Rd, K	Add Immediate to Word	$Rd+1:Rd \leftarrow Rd+1:Rd + K$	ZCNV	2
SUB	Rd, Rr	Subtract without Carry	$Rd \leftarrow Rd - Rr$	ZCNVH	1
SUBI	Rd, K	Subtract Immediate	$Rd \leftarrow Rd - K$	ZCNVH	1
SBC	Rd, Rr	Subtract with Carry	$Rd \leftarrow Rd - Rr - C$	ZCNVH	1
SBCI	Rd, K	Subtract Immediate with Carry	$Rd \leftarrow Rd - K - C$	ZCNVH	1
SBIW	Rd, K	Subtract Immediate from Word	$Rd+1:Rd \leftarrow Rd+1:Rd - K$	ZCNV	2
AND	Rd, Rr	Logical AND	$Rd \leftarrow Rd \text{ AND } Rr$	ZNV	1
ANDI	Rd, K	Logical AND with Immediate	$Rd \leftarrow Rd \text{ AND } K$	ZNV	1
OR	Rd, Rr	Logical OR	$Rd \leftarrow Rd \text{ OR } Rr$	ZNV	1
ORI	Rd, K	Logical OR with Immediate	$Rd \leftarrow Rd \text{ OR } K$	ZNV	1
EOR	Rd, Rr	Exclusive OR	$Rd \leftarrow Rd \text{ XOR } Rr$	ZNV	1
COM	Rd	Ones'-Complement	$Rd \leftarrow \$FF - Rd$	ZCNV	1
NEG	Rd	Two's-Complement	$Rd \leftarrow \$00 - Rd$	ZCNVH	1
SBR	Rd,K	Set Bit(s) in Register	$Rd \leftarrow Rd \text{ OR } K$	ZNV	1
CBR	Rd,K	Clear Bit(s) in Register	$Rd \leftarrow Rd \text{ AND } (\$FFh - K)$	ZNV	1
INC	Rd	Increment	$Rd \leftarrow Rd + 1$	ZNV	1
DEC	Rd	Decrement	$Rd \leftarrow Rd - 1$	ZNV	1
TST	Rd	Test for Zero or Minus	$Rd \leftarrow Rd \text{ AND } Rd$	ZNV	1
CLR	Rd	Clear Register	$Rd \leftarrow Rd \text{ XOR } Rd$	ZNV	1
SER	Rd	Set Register	$Rd \leftarrow \$FF$	-	1
MUL	Rd,Rr	Multiply Unsigned	$R1, R0 \leftarrow Rd \times Rr$	C	2

AVR branch instructions

Mnemonic	Operands	Description	Operation	Flags	Clock
RJMP	k	Relative Jump	$PC \leftarrow PC + k + 1$	-	2
LJMP		Indirect jump to (Z)	$PC \leftarrow Z$	-	2
JMP	k	Jump	$PC \leftarrow k$	-	3
RCALL	k	Relative Call Subroutine	$PC \leftarrow PC + k + 1$	-	3
ICALL		Indirect Call to (Z)	$PC \leftarrow Z$	-	3
CALL	k	Call Subroutine	$PC \leftarrow k$	-	4
RET		Subroutine Return	$PC \leftarrow STACK$	-	4
RETI		Interrupt Return	$PC \leftarrow STACK$	I	4
CPSE	Rd,Rr	Compare, Skip if Equal	if (Rd = Rr) $PC \leftarrow PC + 2$ or 3	-	1/2/3
CP	Rd,Rr	Compare	$Rd - Rr$	ZCINVH	1
CPC	Rd,Rr	Compare with Carry	$Rd - Rr - C$	ZCINVH	1
CPI	Rd,K	Compare with Immediate	$Rd - K$	ZCINVH	1
SBRC	Rr,b	Skip if Bit in Register Cleared	if (Rr(b)=0) $PC \leftarrow PC + 2$ or 3	-	1/2/3
SBRS	Rr,b	Skip if Bit in Register Set	if (Rr(b)=1) $PC \leftarrow PC + 2$ or 3	-	1/2/3
SBIC	P,b	Skip if Bit in I/O Register Cleared	if (I/O(P,b)=0) $PC \leftarrow PC + 2$ or 3	-	1/2/3
SBIS	P,b	Skip if Bit in I/O Register Set	if (I/O(P,b)=1) $PC \leftarrow PC + 2$ or 3	-	1/2/3
BRBS	s,k	Branch if Status Flag Set	if (SREG(s)=1) then $PC \leftarrow PC + k + 1$	-	1/2
BRBC	s,k	Branch if Status Flag Cleared	if (SREG(s)=0) then $PC \leftarrow PC + k + 1$	-	1/2
BREQ	k	Branch if Equal	if (Z=1) then $PC \leftarrow PC + k + 1$	-	1/2
BRNE	k	Branch if Not Equal	if (Z=0) then $PC \leftarrow PC + k + 1$	-	1/2
BRCS	k	Branch if Carry Set	if (C=1) then $PC \leftarrow PC + k + 1$	-	1/2
BRCC	k	Branch if Carry Cleared	if (C=0) then $PC \leftarrow PC + k + 1$	-	1/2
BRSH	k	Branch if Same or Higher	if (C=0) then $PC \leftarrow PC + k + 1$	-	1/2
BRL0	k	Branch if Lower	if (C=1) then $PC \leftarrow PC + k + 1$	-	1/2
BRMI	k	Branch if Minus	if (N=1) then $PC \leftarrow PC + k + 1$	-	1/2
BRPL	k	Branch if Plus	if (N=0) then $PC \leftarrow PC + k + 1$	-	1/2
BRGE	k	Branch if Greater or Equal, Signed	if (N XOR V)=0 then $PC \leftarrow PC + k + 1$	-	1/2
BRLT	k	Branch if Less Than, Signed	if (N XOR V)=1 then $PC \leftarrow PC + k + 1$	-	1/2
BRHS	k	Branch if Half Carry Flag Set	if (H=1) then $PC \leftarrow PC + k + 1$	-	1/2
BRHC	k	Branch if Half Carry Flag Cleared	if (H=0) then $PC \leftarrow PC + k + 1$	-	1/2
BRTS	k	Branch if T Flag Set	if (T=1) then $PC \leftarrow PC + k + 1$	-	1/2
BRTC	k	Branch if T Flag Cleared	if (T=0) then $PC \leftarrow PC + k + 1$	-	1/2
BRVS	k	Branch if Overflow Flag is Set	if (V=1) then $PC \leftarrow PC + k + 1$	-	1/2
BRVC	k	Branch if Overflow Flag is Cleared	if (V=0) then $PC \leftarrow PC + k + 1$	-	1/2
BRIE	k	Branch if Interrupt Enabled	if (I=1) then $PC \leftarrow PC + k + 1$	-	1/2
BRID	k	Branch if Interrupt Disabled	if (I=0) then $PC \leftarrow PC + k + 1$	-	1/2

AVR data transfer instructions

Mnem	Operands	Description	Operation	Flags	Clk
onlc					
MOV	Rd,Rr	Copy Register	$Rd \leftarrow Rr$	-	1
LDI	Rd,K	Load Immediate	$Rd \leftarrow K$	-	1
LDS	Rd,k	Load Direct from SRAM	$Rd \leftarrow (k)$	-	3
LD	Rd,X	Load Indirect	$Rd \leftarrow (X)$	-	2
LD	Rd,X+	Load Indirect and Post-Increment	$Rd \leftarrow (X), X \leftarrow X + 1$	-	2
LD	Rd,-X	Load Indirect and Pre-Decrement	$X \leftarrow X - 1, Rd \leftarrow (X)$	-	2
LD	Rd,Y	Load Indirect	$Rd \leftarrow (Y)$	-	2
LD	Rd,Y+	Load Indirect and Post-Increment	$Rd \leftarrow (Y), Y \leftarrow Y + 1$	-	2
LD	Rd,-Y	Load Indirect and Pre-Decrement	$Y \leftarrow Y - 1, Rd \leftarrow (Y)$	-	2
LDD	Rd,Y+q	Load Indirect with Displacement	$Rd \leftarrow (Y + q)$	-	2
LD	Rd,Z	Load Indirect	$Rd \leftarrow (Z)$	-	2
LD	Rd,Z+	Load Indirect and Post-Increment	$Rd \leftarrow (Z), Z \leftarrow Z + 1$	-	2
LD	Rd,-Z	Load Indirect and Pre-Decrement	$Z \leftarrow Z - 1, Rd \leftarrow (Z)$	-	2
LDD	Rd,Z+q	Load Indirect with Displacement	$Rd \leftarrow (Z + q)$	-	2
STS	k,Rr	Store Direct to SRAM	$(k) \leftarrow Rr$	-	3
ST	X,Rr	Store Indirect	$(X) \leftarrow Rr$	-	2
ST	X+,Rr	Store Indirect and Post-Increment	$(X) \leftarrow Rr, X \leftarrow X + 1$	-	2
ST	-X,Rr	Store Indirect and Pre-Decrement	$X \leftarrow X - 1, (X) \leftarrow Rr$	-	2
ST	Y,Rr	Store Indirect	$(Y) \leftarrow Rr$	-	2
ST	Y+,Rr	Store Indirect and Post-Increment	$(Y) \leftarrow Rr, Y \leftarrow Y + 1$	-	2
ST	-Y,Rr	Store Indirect and Pre-Decrement	$Y \leftarrow Y - 1, (Y) \leftarrow Rr$	-	2
STD	Y+q,Rr	Store Indirect with Displacement	$(Y + q) \leftarrow Rr$	-	2
ST	Z,Rr	Store Indirect	$(Z) \leftarrow Rr$	-	2
ST	Z+,Rr	Store Indirect and Post-Increment	$(Z) \leftarrow Rr, Z \leftarrow Z + 1$	-	2
ST	-Z,Rr	Store Indirect and Pre-Decrement	$Z \leftarrow Z - 1, (Z) \leftarrow Rr$	-	2
STD	Z+q,Rr	Store Indirect with Displacement	$(Z + q) \leftarrow Rr$	-	2
LPM		Load Program Memory	$R0 \leftarrow (Z)$	-	3
IN	Rd,P	In Port	$Rd \leftarrow P$	-	1
OUT	P,Rr	Out Port	$P \leftarrow Rr$	-	1
PUSH	Rr	Push Register on Stack	$STACK \leftarrow Rr$	-	2
POP	Rd	Pop Register from Stack	$Rd \leftarrow STACK$	-	2

AVR bit and bit-test instructions

Mnem onic	Operands	Description	Operation	Flags	Clk
LSL	Rd	Logical Shift Left	$Rd(n+1) \leftarrow Rd(n), Rd(0) \leftarrow 0, C \leftarrow Rd(7)$	ZC NVH	1
LSR	Rd	Logical Shift Right	$Rd(n) \leftarrow Rd(n+1), Rd(7) \leftarrow 0, C \leftarrow Rd(0)$	ZC NV	1
ROL	Rd	Rotate Left Through Carry	$Rd(0) \leftarrow C, Rd(n+1) \leftarrow Rd(n), C \leftarrow Rd(7)$	ZC NVH	1
ROR	Rd	Rotate Right Through Carry	$Rd(7) \leftarrow C, Rd(n) \leftarrow Rd(n+1), C \leftarrow Rd(0)$	ZC NV	1
ASR	Rd	Arithmetic Shift Right	$Rd(n) \leftarrow Rd(n+1), n=0..6$	ZC NV	1
SWAP	Rd	Swap Nibbles	$Rd(3..0) \leftrightarrow Rd(7..4)$	-	1
BSET	s	Flag Set	$SREG(s) \leftarrow 1$	SREG(s)	1
BCLR	s	Flag Clear	$SREG(s) \leftarrow 0$	SREG(s)	1
SBI	P, b	Set Bit in I/O Register	$I/O(P, b) \leftarrow 1$	-	2
CBI	P, b	Clear Bit in I/O Register	$I/O(P, b) \leftarrow 0$	-	2
BST	Rr, b	Bit Store from Register to T	$T \leftarrow Rr(b)$	T	1
BLD	Rd, b	Bit load from T to Register	$Rd(b) \leftarrow T$	-	1
SEC		Set Carry	$C \leftarrow 1$	C	1
CLC		Clear Carry	$C \leftarrow 0$	C	1
SEN		Set Negative Flag	$N \leftarrow 1$	N	1
CLN		Clear Negative Flag	$N \leftarrow 0$	N	1
SEZ		Set Zero Flag	$Z \leftarrow 1$	Z	1
CLZ		Clear Zero Flag	$Z \leftarrow 0$	Z	1
SEI		Global Interrupt Enable	$I \leftarrow 1$	I	1
CLI		Global Interrupt Disable	$I \leftarrow 0$	I	1
SES		Set Signed Test Flag	$S \leftarrow 1$	S	1
CLS		Clear Signed Test Flag	$S \leftarrow 0$	S	1
SEV		Set Two's-Complement Overflow	$V \leftarrow 1$	V	1
CLV		Clear Two's-Complement Overflow	$V \leftarrow 0$	V	1
SET		Set T in SREG	$T \leftarrow 1$	T	1
CLT		Clear T in SREG	$T \leftarrow 0$	T	1
SEH		Set Half Carry Flag in SREG	$H \leftarrow 1$	H	1
CLH		Clear Half Carry Flag in SREG	$H \leftarrow 0$	H	1
NOP		No Operation		-	1
SLEEP		Sleep		-	1
WDR		Watchdog Reset		-	1

Source: AVR Assembler User Guide (doc1022.pdf)

D | x86 instruction set

Instruction	Meaning	Opcode
AAA	ASCII adjust AL after addition	0x37
AAD	ASCII adjust AX before division	0xD5
AAM	ASCII adjust AX after multiplication	0xD4
AAS	ASCII adjust AL after subtraction	0x3F
ADC	Add with carry	0x10...0x15, 0x80/2...0x83/2
ADD	Add	0x00...0x05, 0x80/0...0x83/0
AND	Logical AND	0x20...0x25, 0x80/4...0x83/4
CALL	Call procedure	0x9A, 0xE8, 0xFF/2, 0xFF/3
CBW	Convert byte to word	0x98
CLC	Clear carry flag	0xF8
CLD	Clear direction flag	0xFC
CLI	Clear interrupt flag	0xFA
CMC	Complement carry flag	0xF5
CMP	Compare operands	0x38...0x3D, 0x80/7...0x83/7
CMPSB	Compare bytes in memory	0xA6
CMPSW	Compare words	0xA7
CWD	Convert word to doubleword	0x99
DAA	Decimal adjust AL after addition	0x27
DAS	Decimal adjust AL after subtraction	0x2F
DEC	Decrement by 1	0x48...0x4F, 0xFE/1, 0xFF/1
DIV	Unsigned divide	0xF6/6, 0xF7/6
ESC	Used with floating-point unit	0xD8...0xDF
HLT	Enter halt state	0xF4
IDIV	Signed divide	0xF6/7, 0xF7/7
IMUL	Signed multiply	0xF6/5, 0xF7/5
IN	Input from port	0xE4, 0xE5, 0xEC, 0xED
INC	Increment by 1	0x40...0x47, 0xFE/0, 0xFF/0
INT	Call to interrupt	0xCC, 0xCD
INTO	Call to interrupt if overflow	0xCE
IRET	Return from interrupt	0xCF
JCC	Jump if condition	0x70...0x7F
JCXZ	Jump if CX is zero	0xE3
JMP	Jump	0xE9...0xEB, 0xFF/4, 0xFF/5

Instruction	Meaning	Opcode
LAHF	Load FLAGS into AH register	0x9F
LDS	Load pointer using DS	0xC5
LEA	Load Effective Address	0x8D
LES	Load ES with pointer	0xC4
LOCK	Assert BUS LOCK# signal	0xF0
LODSB	Load string byte	0xAC
LODSW	Load string word	0xAD
LOOP/LOOPX	Loop control	0xE0...0xE2
MOV	Move	0xA0...0xA3
MOVSB	Move byte from string to string	0xA4
MOVSW	Move word from string to string	0xA5
MUL	Unsigned multiply	0xF6/4...0xF7/4
NEG	Two's-complement negation	0xF6/3...0xF7/3
NOP	No operation	0x90
NOT	Negate the operand, logical NOT	0xF6/2...0xF7/2
OR	Logical OR	0x08...0x0D, 0x80...0x83/1
OUT	Output to port	0xE6, 0xE7, 0xEE, 0xEF
POP	Pop data from stack	0x07, 0x0F, 0x17, 0x1F, 0x58...0x5F, 0x8F/0
POPF	Pop FLAGS register from stack	0x9D
PUSH	Push data onto stack	0x06, 0x0E, 0x16, 0x1E, 0x50...0x57, 0xFF/6
PUSHF	Push FLAGS onto stack	0x9C
RCL	Rotate left (with carry)	0xD0...0xD3/2
RCR	Rotate right (with carry)	0xD0...0xD3/3
REPXX	Repeat MOVs/STOS/CMPS/LODS/SCAS	0xF2, 0xF3
RET	Return from procedure	
RETN	Return from near procedure	0xC2, 0xC3
RETF	Return from far procedure	0xCA, 0xCB
ROL	Rotate left	0xD0...0xD3/0
ROR	Rotate right	0xD0...0xD3/1
SAHF	Store AH into FLAGS	0x9E
SAL	Shift Arithmetically left (signed shift left)	0xD0...0xD3/4
SAR	Shift Arithmetically right (signed shift right)	0xD0...0xD3/7
SBB	Subtraction with borrow	0x18...0x1D, 0x80...0x83/3
SCASB	Compare byte string	0xAE
SCASW	Compare word string	0xAF
SHL	Shift left (unsigned shift left)	0xD0...0xD3/4
SHR	Shift right (unsigned shift right)	0xD0...0xD3/5
STC	Set carry flag	0xF9
STD	Set direction flag	0xFD
STI	Set interrupt flag	0xFB
STOSB	Store byte in string	0xAA
STOSW	Store word in string	0xAB
SUB	Subtraction	0x28...0x2D, 0x80...0x83/5
TEST	Logical compare (AND)	0x84, 0x84, 0xA8, 0xA9, 0xF6/0, 0xF7/0
WAIT	Wait until not busy	0x9B
XCHG	Exchange data	0x86, 0x87, 0x91...0x97
XLAT	Table look-up translation	0xD7
XOR	Exclusive OR	0x30...0x35, 0x80...0x83/6

Floating Point x86

Instruction	Meaning
F2XM1	$2^x - 1$
FABS	Absolute value
FADD	Add
FADDP	Add and pop
FBLD	Load BCD
FBSTP	Store BCD and pop
FCHS	Change sign
FCLEX	Clear exceptions
FCOM	Compare
FCOMP	Compare and pop
FCOMPP	Compare and pop twice
FDECSTP	Decrement floating point stack pointer
FDISI	Disable interrupts
FDIV	Divide
FDIVP	Divide and pop
FDIVR	Divide reversed
FDIVRP	Divide reversed and pop
FENI	Enable interrupts
FFREE	Free register
FIADD	Integer add
FICOM	Integer compare
FICOMP	Integer compare and pop
FIDIV	Integer divide
FIDIVR	Integer divide reversed
FILD	Load integer
FIMUL	Integer multiply
FINCSTP	Increment floating point stack pointer
FINIT	Initialize floating point processor
FIST	Store integer
FISTP	Store integer and pop
FISUB	Integer subtract
FISUBR	Integer subtract reversed
FLD	Floating point load
FLD1	Load 1.0 onto stack
FLDCW	Load control word
FLDENV	Load environment state
FLDENVW	Load environment state, 16-bit
FLDL2E	Load $\log_2(e)$ onto stack
FLDL2T	Load $\log_2(10)$ onto stack
FLDLG2	Load $\log_{10}(2)$ onto stack
FLDLN2	Load $\ln(2)$ onto stack

Floating Point x86	
Instruction	Meaning
FLDPI	Load π onto stack
FLDZ	Load 0.0 onto stack
FMUL	Multiply
FMULP	Multiply and pop
FNCLEX	Clear exceptions, no wait
FNDISI	Disable interrupts, no wait
FNENI	Enable interrupts, no wait
FNINIT	Initialize floating point processor, no wait
FNOP	No operation
FNSAVE	Save FPU state, no wait, 8-bit
FNSAVEW	Save FPU state, no wait, 16-bit
FNSTCW	Store control word, no wait
FNSTENV	Store FPU environment, no wait
FNSTENVW	Store FPU environment, no wait, 16-bit
FNSTSW	Store status word, no wait
FPATAN	Partial arctangent
FPREM	Partial remainder
FPTAN	Partial tangent
FRNDINT	Round to integer
FRSTOR	Restore saved state
FRSTORW	Restore saved state
FSAVE	Save FPU state
FSAVEW	Save FPU state, 16-bit
FSCALE	Scale by factor of 2
FSQRT	Square root
FST	Floating point store
FSTCW	Store control word
FSTENV	Store FPU environment
FSTENVW	Store FPU environment, 16-bit
FSTP	Store and pop
FSTSW	Store status word
FSUB	Subtract
FSUBP	Subtract and pop
FSUBR	Reverse subtract
FSUBRP	Reverse subtract and pop
FTST	Test for zero
FWAIT	Wait while FPU is executing
FXAM	Examine condition flags
FXCH	Exchange registers
FXTRACT	Extract exponent and significand
FYL2X	$y \cdot \log_2 x$
FYL2XP1	$y \cdot \log_2(x + 1)$

For a full list of instructions and their descriptions please consult the AMD reference AMD64 Architecture Programmer's Manual Volume 1: Application Programming (Publication No. 24592 of Advanced Micro Devices).

E | x86 processor exceptions and hardware interrupts

(source: https://en.wikipedia.org/wiki/Interrupt_descriptor_table)

x86 processor exceptions:

Int	Description
0x00	Division by zero
0x01	Single-step interrupt
0x02	Non-maskable interrupt
0x03	Breakpoint
0x04	Overflow
0x05	Bound range exceeded
0x06	Invalid opcode
0x07	Co-processor not available
0x08	Double fault
0x09	Co-processor overrun
0x0A	Invalid task state segment
0x0B	Segment not present
0x0C	Stack segment fault
0x0D	General protection fault
0x0E	Page fault
0x0F	Reserved
0x10	x87 floating point exception
0x11	Alignment check
0x12	Machine check
0x13	SIMD floating-point exception
0x14	Virtualization exception
0x15	Control protection exception

x86 hardware interrupts:

Int	Description
0x20-0x27	IRQ 0-7
0x70-0x77	IRQ 8-15

IRQ 0	System timer	IRQ 8	Clock
IRQ 1	Keyboard controller	IRQ 9	ACPI (advanced)
IRQ 2	Cascaded from IRQ 8-15	IRQ 10	Peripherals (SCSI..)
IRQ 3	Serial port 2	IRQ 11	Peripherals (SCSI..)
IRQ 4	Serial port 1	IRQ 12	Mouse
IRQ 5	Parallel port or sound card	IRQ 13	CPU float
IRQ 6	Floppy disk	IRQ 14	ATA (HD/CD)
IRQ 7	Parallel port	IRQ 15	Secondary ATA

F | x86 BIOS, MS-DOS and API and Linux interrupts

x86 BIOS software interrupts

(source: https://en.wikipedia.org/wiki/BIOS_interrupt_call)

Int	Description
05h	Print-screen button press
10h	Video services
12h	Return memory size
13h	Low level disk services
14h	Serial port services
15h	Misc. system services
16h	Keyboard services
17h	Printer services
19h	Load OS
1Ah	Real-time clock services
1Bh	Ctrl-Break handler
1Ch	Timer tick handler
20h	Program terminate
22h	Program terminate address

x86 OS software interrupts

Int	Description
21h	MS-DOS API interrupts (see p. 320)
80h	Linux system call (see p. 320)

MS-DOS API (application programming interface) interrupt 21h

Note: MS-DOS must be loaded in memory

(source: https://en.wikipedia.org/wiki/DOS_API)

AH	description	AH	description
00h*	Program terminate	30h	Get DOS version
01h	Read character from <code>stdin</code>	31h	Terminate and stay resident
02h	Write character to <code>stdout</code>	35h	Get Interrupt vector
03h	Auxiliary input	36h	Get free disk space
04h	Auxiliary output	39h	Create subdirectory
05h	Write character to printer	3Ah	Remove subdirectory
06h	Console Input/Output	3Bh	Set working directory
07h	Direct char read <code>stdin</code> , no echo	3Ch	Create file
08h	Char read from <code>stdin</code> , no echo	3Dh	Open file
09h	Write string to <code>stdout</code>	3Eh	Close file
0Ah	Buffered input	3Fh	Read file
0Bh	Get <code>stdin</code> status	40h	Write file
0Ch	Flush buffer for <code>stdin</code>	41h	Delete file
0Dh	Disk reset	42h	Seek file
0Eh	Select default drive	43h	Get/set file attributes
19h	Get current default drive	47h	Get current directory
25h	Set interrupt vector	4Ch	Exit program
2Ah	Get system date	4Dh	Get return code
2Bh	Set system date	54h	Get verify flag
2Ch	Get system time	56h	Rename file
2Dh	Set system time	57h	Get/set file date
2Eh	Set verify flag		

*: Also BIOS interrupt 20h

Linux x86 interrupt 0x80

Note: Linux must be loaded in memory

(source: http://faculty.nps.edu/cseagle/assembly/sys_call.html)

		arguments			return
eax	description	ebx	ecx	edx	eax
1	exit	int error_code			
3	read	u-int fd	char *buf		len
4	write	u-int fd	const char *buf	size	
5	open	const char *filename	int flags	int mode	fd
6	close	u-int fd			
13	time	time_t *tloc			time_t

Linux x86 `syscall`

(source: http://blog.rchapman.org/posts/Linux_System_Call_Table_for_x86_64/)

		arguments			return
rax	description	rdi	rsi	rdx	rax
0x00	read	u-int fd	char *buf	size	len
0x01	write	u-int fd	const char *buf	size	
0x02	open	const char *filename	int flags	int mode	fd
0x03	close	u-int fd			
0x3c	exit	int error_code			
0xc9	time	time_t *tloc			time_t

G | Linux (Debian) system calls

(source: Debian man page syscall)

Syntax for system calls for various architectures in Linux Debian:

Arch/ABI	Instruction	System call	Ret val1	Ret val2	Error
alpha	callsys	v0	v0	a4	a3
arc	trap0	r8	r0	-	-
arm/OABI	swi NR	-	a1	-	-
arm/EABI	swi 0x0	r7	r0	r1	-
arm64	svc #0	x8	x0	x1	-
blackfin	except 0x0	P0	R0	-	-
i386	int \$0x80	eax	eax	edx	-
ia64	break 0x100000	r15	r8	r9	r10
m68k	trap #0	d0	d0	-	-
microblaze	brki r14,8	r12	r3	-	-
mips	syscall	v0	v0	v1	a3
nios2	trap	r2	r2	-	r7
parisc	ble 0x100(%sr2,%r0)	r20	r28	-	-
powerpc	sc	r0	r3	-	r0
powerpc64	sc	r0	r3	-	cr0.S0
riscv	ecall	a7	a0	a1	-
s390	svc 0	r1	r2	r3	-
s390x	svc 0	r1	r2	r3	-
superh	trap #0x17	r3	r0	r1	-
sparc/32	t 0x10	g1	o0	o1	psr/csr
sparc/64	t 0x6d	g1	o0	o1	psr/csr
tile	swint1	R10	R00	-	R01
x86-64	syscall	rax	rax	rdx	-
x32	syscall	rax	rax	rdx	-
xtensa	syscall	a2	a2	-	-

Registers used to pass the system call arguments:

Arch/ABI	arg1	arg2	arg3	arg4	arg5	arg6	arg7

alpha	a0	a1	a2	a3	a4	a5	-
arc	r0	r1	r2	r3	r4	r5	-
arm/OABI	a1	a2	a3	a4	v1	v2	v3
arm/EABI	r0	r1	r2	r3	r4	r5	r6
arm64	x0	x1	x2	x3	x4	x5	-
blackfin	R0	R1	R2	R3	R4	R5	-
i386	ebx	ecx	edx	esi	edi	ebp	-
ia64	out0	out1	out2	out3	out4	out5	-
m68k	d1	d2	d3	d4	d5	a0	-
microblaze	r5	r6	r7	r8	r9	r10	-
mips/o32	a0	a1	a2	a3	-	-	-
mips/n32,64	a0	a1	a2	a3	a4	a5	-
nios2	r4	r5	r6	r7	r8	r9	-
parisc	r26	r25	r24	r23	r22	r21	-
powerpc	r3	r4	r5	r6	r7	r8	r9
powerpc64	r3	r4	r5	r6	r7	r8	-
riscv	a0	a1	a2	a3	a4	a5	-
s390	r2	r3	r4	r5	r6	r7	-
s390x	r2	r3	r4	r5	r6	r7	-
superh	r4	r5	r6	r7	r0	r1	r2
sparc/32	o0	o1	o2	o3	o4	o5	-
sparc/64	o0	o1	o2	o3	o4	o5	-
tile	R00	R01	R02	R03	R04	R05	-
x86-64	rdi	rsi	rdx	r10	r8	r9	-
x32	rdi	rsi	rdx	r10	r8	r9	-
xtensa	a6	a3	a4	a5	a8	a9	-

H | MIPS instruction set

	31	26	25	21	20	16	15	11	10	6	5	0
R :	opcode		\$A		\$B		\$C		imm		func	
I :	opcode		\$A		\$B		imm					
J :	opcode		imm									
fR :	opcode		type		\$A		\$B		\$C		func	
fI :	opcode		type		\$A		imm					
	6		5		5		5		5		6	

Values in the tables on the next pages are hexadecimal, unless otherwise specified

MIPS: Logic and arithmetic instructions

MIPS instruction		Meaning	
Mne- monic	Operands	Description	Operation
sll	\$rd, \$rt, num	Shift left logical	$\$rd \leftarrow \$rt \ll \text{num}$
srl	\$rd, \$rt, num	Shift right logical	$\$rd \leftarrow \$rt \gg \text{num}$
sra	\$rd, \$rt, num	Shift right arithmetic	$\$rd \leftarrow \$rt \gg \text{num} + \text{msb}$
sllv	\$rd, \$rt, \$rs	Shift left logic. var.	$\$rd \leftarrow \$rt \ll \$rs$
srlv	\$rd, \$rt, \$rs	Shift right logic. var.	$\$rd \leftarrow \$rt \gg \$rs$
srav	\$rd, \$rt, \$rs	Shift right arithm. var.	$\$rd \leftarrow \$rt \gg \$rs + \text{msb}$
num: (decimal) 0...31			
add	\$rd, \$rt, \$rs	Add	$\$rd \leftarrow \$rt + \$rs$
addu	\$rd, \$rt, \$rs	Add unsigned	$\$rd \leftarrow \$rt + \$rs$
sub	\$rd, \$rt, \$rs	Subtract	$\$rd \leftarrow \$rt - \$rs$
subu	\$rd, \$rt, \$rs	Subtract unsigned	$\$rd \leftarrow \$rt - \$rs$
addi	\$rd, \$rt, imm	Add immediate	$\$rd \leftarrow \$rt + \text{imm}$
addiu	\$rd, \$rt, imm	Add imm. unsign.	$\$rd \leftarrow \$rt + \text{imm}$
mul	\$rd, \$rt, \$rs	mult and move	$\$rd \leftarrow \$rt * \$rs$
mult	\$rt, \$rs	Multiply	$\$hi, \$lo \leftarrow \$rt * \rs
multu	\$rt, \$rs	Multiply unsigned	$\$hi, \$lo \leftarrow \$rt * \rs
div	\$rt, \$rs	Divide	$\$lo \leftarrow \$rt / \$rs$ $\$hi \leftarrow \$rt \% \$rs$
divu	\$rt, \$rs	Divide unsigned	$\$lo \leftarrow \$rt / \$rs$ $\$hi \leftarrow \$rt \% \$rs$
imm: halfword			
and	\$rd, \$rt, \$rs	AND	$\$rd \leftarrow \$rs \& \$rt$
or	\$rd, \$rt, \$rs	OR	$\$rd \leftarrow \$rt \$rs$
nor	\$rd, \$rt, \$rs	NOR	$\$rd \leftarrow !(\$rt \$rs)$
xor	\$rd, \$rt, \$rs	XOR	$\$rd \leftarrow \$rt \wedge \$rs$
andi	\$rd, \$rt, imm	AND immediate	$\$rd \leftarrow \$rt \& \text{imm}$
ori	\$rd, \$rt, imm	OR immediate	$\$rd \leftarrow \$rt \text{imm}$
xori	\$rd, \$rt, imm	XOR immediate	$\$rd \leftarrow \$rt \wedge \text{imm}$
imm: halfword			
slt	\$rd, \$rt, \$rs	Set (1) if less than*	$\$rd \leftarrow (\$rt < \$rs) ? 1 : 0$
sltu	\$rd, \$rt, \$rs	slt unsigned	$\$rd \leftarrow (\$rt < \$rs) ? 1 : 0$
slti	\$rd, \$rt, imm	slt immediate	$\$rd \leftarrow (\$rt < \text{imm}) ? 1 : 0$
sltiu	\$rd, \$rt, imm	slt imm. unsigned	$\$rd \leftarrow (\$rt < \text{imm}) ? 1 : 0$

imm: halfword. *: Unset (0) otherwise

MIPS: Jump, branch and memory instructions

MIPS instruction		Meaning	
Mne- monic	Operands	Description	Operation
j	addr	Jump	pc←addr
jal	addr	Jump and link	pc←addr,\$ra←pc+4
jr	\$rs	Jump register	pc←\$rs
jalr	[\$rd,] \$rs	Jump and link register	pc←\$rs,{ \$ra/\$rd }←pc+4
beq	\$rt, \$rs, addr	Branch if equal	\$rs==\$rt?pc←addr
bne	\$rt, \$rs, addr	Branch if not equal	\$rs!=\$rt?pc←addr
bltz	\$rt, addr	Branch if < 0	\$rt<0?pc←addr
bgtz	\$rt, addr	Branch if > 0	\$rt>0?pc←addr
blez	\$rt, addr	Branch if ≤ 0	\$rt≤0?pc←addr
bgez	\$rt, addr	Branch if ≥ 0	\$rt≥0?pc←addr
if condition is false: pc←pc+4. See p. 330 for more branching instructions			
mfhi	\$rd	Move from HI	\$rd←hi
mthi	\$rs	Move to HI	\$rs→hi
mflo	\$rd	Move from LO	\$rd←lo
mtlo	\$rs	Move to LO	\$rs→lo
lb	\$rd, offset(\$rt)	Load byte	\$rd←M[\$rt+offset]
lbu	\$rd, offset(\$rt)	Load byte unsigned	\$rd←M[\$rt+offset]
lh	\$rd, offset(\$rt)	Load halfword	\$rd←M[\$rt+offset]
lhu	\$rd, offset(\$rt)	Load halfword unsigned	\$rd←M[\$rt+offset]
lui	\$rd, imm	Load upper immediate	\$rd←imm≪16
lw	\$rd, offset(\$rt)	Load word	\$rd←M[\$rt+offset]
sb	\$rs, offset(\$rt)	Store byte	\$rs→M[\$rt+offset]
sh	\$rs, offset(\$rt)	Store halfword	\$rs→M[\$rt+offset]
sw	\$rs, offset(\$rt)	Store word	\$rs→M[\$rt+offset]

appropriate bits only, rest unchanged. M: memory. offset: halfword

MIPS: Jump, branch and memory instructions

MIPS instruction		Machine code					
Mne- monic	Operands	for- mat	opcode/ func	\$A	\$B	\$C	imm
j	addr	J	02/-	-	-	-	mask
jal	addr	J	03/-	-	-	-	mask
jr	\$rs	R	00/08	\$rs	0	0	0
jalr	[\$rd,] \$rs	R	00/09	\$rs	0	\$rd	0
imm: 26-bit unsigned value. addr=page+4*imm, page=pc&0xF0000000 mask=(addr&0xFFFFFC)/4							
beq	\$rt, \$rs, addr	I	04/-	\$rt	\$rs	-	relative
bne	\$rt, \$rs, addr	I	05/-	\$rt	\$rs	-	relative
bltz	\$rt, addr	I	01/-	\$rt	0	-	relative
bgtz	\$rt, addr	I	07/-	\$rt	0	-	relative
blez	\$rt, addr	I	06/-	\$rt	0	-	relative
bgez	\$rt, addr	I	01/-	\$rt	1	-	relative
imm: 16-bit two's-complement signed value. relative=(addr-pc-4)/4							
mghi	\$rd	R	00/10	0	0	\$rd	0
mthi	\$rs	R	00/11	\$rs	0	0	0
mflo	\$rd	R	00/12	0	0	\$rd	0
mtlo	\$rs	R	00/13	\$rs	0	0	0
lb	\$rd, offset(\$rt)	I	20/-	\$rt	\$rd	-	offset
lbu	\$rd, offset(\$rt)	I	24/-	\$rt	\$rd	-	offset
lh	\$rd, offset(\$rt)	I	21/-	\$rt	\$rd	-	offset
lhu	\$rd, offset(\$rt)	I	25/-	\$rt	\$rd	-	offset
lui	\$rd, imm	I	0f/-	0	\$rd	-	imm
lw	\$rd, offset(\$rt)	I	23/-	\$rt	\$rd	-	offset
sb	\$rs, offset(\$rt)	I	28/-	\$rt	\$rs	-	offset
sh	\$rs, offset(\$rt)	I	29/-	\$rt	\$rs	-	offset
sw	\$rs, offset(\$rt)	I	2b/-	\$rt	\$rs	-	offset

appropriate bits only, rest unchanged. offset: halfword

imm,offset: 16-bit two's-complement signed value.

MIPS: Floating-point instructions

MIPS instruction		Meaning	
Mnemonic	Operands	Description	Operation
lwc1	\$fd, offs(\$rt)	Load single	\$fd←M[\$rt+offs]
ldc1	\$fd, offs(\$rt)	Load double	\$fd←M[\$rt+offs]
swc1	\$fs, offs(\$rt)	Store single	\$fs→M[\$rt+offs]
sdc1	\$fs, offs(\$rt)	Store double	\$fs→M[\$rt+offs]

M: memory

Mnemonic	Operands	Description	Operation
c.eq.SIZE	\$fs, \$ft	FPs equal	cflag←(\$fs==\$ft)?1:0
c.lt.SIZE	\$fs, \$ft	FPs less than	cflag←(\$fs<\$ft)?1:0
c.le.SIZE	\$fs, \$ft	FPs less or equal	cflag←(\$fs<=\$ft)?1:0

cvt.TO.FROM	\$fd, \$fs	Convert	\$fd = convert(\$fs)
-------------	------------	---------	----------------------

add.SIZE	\$fd, \$ft, \$fs	FP add	\$fd←\$ft+\$fs
sub.SIZE	\$fd, \$ft, \$fs	FP subtract	\$fd←\$ft-\$fs
mul.SIZE	\$fd, \$ft, \$fs	FP multiply	\$fd←\$ft*\$fs
div.SIZE	\$fd, \$ft, \$fs	FP divide	\$fd←\$ft/\$fs

mov.SIZE	\$fd, \$fs	Copy	\$fd←\$fs
mfc1	\$rd, \$fs	Copy from co-proc.	\$rd←\$fs
mtc1	\$rs, \$fd	Copy to co-proc.	\$rs→\$fd

SIZE = {s,d}. {TO,FROM} = {s,d,w}

Mnemonic	Operands	Description	Operation
bc1t	addr	Branch if cflag true	cflag? pc←addr
bc1f	addr	Branch if cflag false	cflag? pc←addr

Otherwise: pc←pc+4

Program flow:

Mnemonic	Description	
syscall		System call
break	num	Exit with exception
nop		No operation

*: Specify type of syscall by \$v0. See appendix J

MIPS: Floating-point instructions

MIPS instruction		Machine code				
Mnemonic	Operands	form	opcode	\$A	\$B	imm
lwc1	\$fd, offs(\$rt)	I	31	\$rt	\$fd	offs
ldc1	\$fd, offs(\$rt)	I	35	\$rt	\$fd	offs
swc1	\$fs, offs(\$rt)	I	39	\$rt	\$fs	offs
sdcl	\$fs, offs(\$rt)	I	3d	\$rt	\$fs	offs

offs: 16-bit two's-complement signed value.

Mnemonic	Operands	form	opc/typ	\$A	\$B	\$C	func
c.eq.SIZE	\$fs, \$ft	fR	11/tp	\$ft	\$fs	0	32
c.lt.SIZE	\$fs, \$ft	fR	11/tp	\$ft	\$fs	0	3c
c.le.SIZE	\$fs, \$ft	fR	11/tp	\$ft	\$fs	0	3e

cvt.TO.FROM	\$fd, \$fs	fR	11/from	0	\$fs	\$fd	to
-------------	------------	----	---------	---	------	------	----

add.SIZE	\$fd, \$ft, \$fs	fR	11/tp	\$fs	\$ft	\$fd	00
sub.SIZE	\$fd, \$ft, \$fs	fR	11/tp	\$fs	\$ft	\$fd	01
mul.SIZE	\$fd, \$ft, \$fs	fR	11/tp	\$fs	\$ft	\$fd	02
div.SIZE	\$fd, \$ft, \$fs	fR	11/tp	\$fs	\$ft	\$fd	03

mov.SIZE	\$fd, \$fs	fR	11/tp	\$fd	\$fs	0	06
mfc1	\$rd, \$fs	fR	11/00	\$rd	\$fs	0	00
mtc1	\$rs, \$fd	fR	11/04	\$rs	\$fd	0	00

SIZE, TO, FROM: {tp, from} = 10 (s), 11 (d), 14 (w). to = 20 (s), 21 (d), 24 (w)

Mnemonic	Operands	form	opc/typ	\$A	imm		
bc1t	addr	fI	21/08	1	relative		
bc1f	addr	fI	21/08	0	relative		

imm: 16-bit two's-complement signed value. relative=(addr-pc-4)/4

Program flow:

Mnemonic		form	opc/func	\$A	\$B	\$C	imm
syscall		R	00/0c	0	0	0	0
break	num	R	00/0d	0	0	0	num
nop		R	00/00	0	0	0	0

MIPS: (Some) pseudo-instructions

rol	\$rd, \$rt, num	Rotate left	\$rd←rol(\$rt,num)
	srl \$at, \$rt, 32-num		
	sll \$rd, \$rt, num		
	or \$rd, \$rd, \$at		
ror	\$rd, \$rt, num	Rotate right	\$rd←ror(\$rt,num)
	sll \$at, \$rt, 32-num		
	srl \$rd, \$rt, num		
	or \$rd, \$rd, \$at		
blt	\$rt, \$rs, addr	Branch if less than	\$rt<\$rs?pc←addr
	slt \$at, \$rt, \$rs		
	bne \$at, \$zero, addr		
bgt	\$rt, \$rs, addr	Branch if greater than	\$rt>\$rs?pc←addr
	slt \$at, \$rs, \$rt		
	bne \$at, \$zero, addr		
ble	\$rt, \$rs, addr	Branch if less/equal	\$rt<=\$rs?pc←addr
	slt \$at, \$rs, \$rt		
	beq \$at, \$zero, addr		
bge	\$rt, \$rs, addr	Branch if greater/equal	\$rt<=\$rs?pc←addr
	slt \$at, \$rt, \$rs		
	beq \$at, \$zero, addr		
beqz	\$rt, addr	Branch if equal zero	\$rt==0?pc←addr
	beq \$rt, \$zero, addr		
bnez	\$rt, addr	Branch if not equal 0	\$rt!=0?pc←addr
	bne \$rt, \$zero, addr		
move	\$rd, \$rs	Copy	\$rd←\$rs
	ori \$rd, \$zero, \$rs		
jalr	\$rs	Jump and link register	pc←\$rs,\$ra←pc+4
	jalr \$rs, \$ra		

MIPS: (Some) pseudo-instructions (cont.)

lw	\$rd, addr	Load word from address	\$rd ← M[addr]
	lui \$at, addr ≫ 16 lw \$rd, [addr AND 0x0000FFFF](\$at),		
li	\$rd, word	Load immediate	\$rd ← word
	lui \$at, word ≫ 16 ori \$rd, \$at, word AND 0x0000FFFF		
lh	\$rd, halfword	Load immediate	\$rd ← halfword
	ori \$rd, \$zero, halfword		
la	\$rd, addr	Load address	\$rd ← addr
	lui \$at, addr ≫ 16 ori \$rd, \$at, addr AND 0x0000FFFF Note: equal to instruction li \$rd, addr		

to be implemented by macros (see Section 10.9):

inc	\$rt	Increment	\$rt ← \$rt + 1
	addi \$rt, \$rt, 1		
dec	\$rt	Decrement	\$rt ← \$rt - 1
	subi \$rt, \$rt, 1		

push	\$rs	Push onto stack	M[--\$sp] ← \$rs
	addiu \$sp, \$sp, -4 sw \$rs, 0(\$sp)		
pop	\$rd	Pop from stack	\$rd ← M[\$sp++]
	lw \$rd, 0(\$sp) addiu \$sp, \$sp, 4		

return		Return from subroutine	pc ← \$ra
	jr \$ra		
done		Terminate	
	li \$v0, 10 syscall		

I | MARS (MIPS) Assembler directives

Directive	Example	Meaning
<code>.data</code>		Start of data segment
<code>.text</code>		Start of code segment
<code>.globl</code>	<code>.globl main</code>	Entry point for external reference (linker)
<code>.space n</code>	<code>myarray: .space 12</code>	Reserve n bytes of space on the heap
<code>.ascii "string"</code>	<code>mystring: .ascii "Ajax"</code>	Store string in heap memory
<code>.asciiz "string"</code>	<code>mytext: .asciiz "Benfica"</code>	Store string+0x00 in heap memory
<code>.byte b1, b2,...bn</code>		Store byte(s) in heap memory
<code>.half h1, h2,...hn</code>		Store half-word(s) in memory
<code>.word w1, w2,...wn</code>	<code>myvector: .word 1, 2, 4</code>	Store word(s) in heap memory
<code>.float f1, f2,...fn</code>	<code>myvector: .float 1.0, 2.1, 3.6</code>	Store float(s) in heap memory
<code>.double d1, d2,...dn</code>	<code>pi: .double 3.1415926E03</code>	Store double-precision float(s) in heap memory
<code>.eqv text text</code>	<code>.eqv myvalue 64</code>	Define a substitution
<code>.macro ...</code>	<code>.end_macro</code> <code>.macro endprog</code> <code>li a7, 10</code> <code>ecall</code> <code>.end_macro</code>	Define a macro* (not a function)

*: (does not store it in memory)

J | (MARS) MIPS 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,\$f1 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
open file	13	\$a0 = address of filename \$a1 = flags (0=read, 1=overwrite,9=append) \$a2 = mode (0)	\$v0 file descriptor
read from file	14	\$a0 = file descriptor \$a1 = addr. input buffer \$a2 = max length	\$v0 number of chars read (0:end-of-file, <0:error)
write to file	15	\$a0 = file descriptor \$a1 = addr. output buffer \$a2 = number of chars	\$v0 number of chars written (<0: error)
close file	16	\$a0 = file descriptor	
exit (terminate with value)	17	\$a0 = termination result	

function	\$v0	argument(s)	return value(s)
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 = integer	\$a0: next random int
random int in range	42	\$a0 = integer \$a1 = limit	\$a0: next random int in range 0...\$a1-1
random float	43	\$a0 = integer	\$f0: 0.0...0.999...
random double	44	\$a0 = integer	\$f0, \$f1: 0.0...0.999...

K | (RARS) RISC-V base integer instruction set (RV32I)



type	31	25	24	20	19	15	14	12	11	7	6	0				
R :	fun7				src				tgt		fun3		dst		opcode	
I :	imm								tgt		fun3		dst		opcode	
S/B* :	imm (msb)				src				tgt		fun3		imm (lsb)		opcode	
U/J* :	imm										dst		opcode			
F :	fun5		fmt		src				tgt		rm		dst		opcode	
	7		5		5				3		5		7			

*: Difference between S and B, and U and J, is how the imm fields are constructed on basis of the immediate value used in the operation

Values in the tables on the next pages are hexadecimal, unless otherwise specified

RARS RISC-V: Logic and arithmetic instructions

RISC-V instruction		Meaning	
Mne- monic	Operands	Description	Operation
sll	rd rt rs	Shift left logical	$rd \leftarrow rt \ll rs$
slli	rd rt num	sll immediate	$rd \leftarrow rt \ll num$
srl	rd rt rs	Shift right logical	$rd \leftarrow rt \gg rs$
srli	rd rt num	srl immediate	$rd \leftarrow rt \gg num$
sra	rd rt rs	Shift right arithmetic	$rd \leftarrow rt \gg rs + msb$
srai	rd rt num	sra immediate	$rd \leftarrow rt \gg num + msb$
num: (decimal) 0...31			
add	rd rt rs	Add	$rd \leftarrow rt + rs$
addi	rd rt imm	add immediate	$rd \leftarrow rt + imm$
sub	rd rt rs	Subtract	$rd \leftarrow rt - rs$
mul	rd rt rs	Multiply (lo)	$rd \leftarrow rt * rs [31:0]$
mulh	rd rt rs	Multiply (hi)	$rd \leftarrow rt * rs [63:32]$
mulhu	rd rt rs	mulh unsigned	$rd \leftarrow rt * rs [63:32]$
mulhsu	rd rt rs	mulhu rt signed	$rd \leftarrow rt * rs [63:32]$
div	rd rt rs	Divide	$rd \leftarrow rt / rs$
divu	rd rt rs	div unsigned	$rd \leftarrow rt / rs$
rem	rd rt rs	Remainder	$rd \leftarrow rt \% rs$
remu	rd rt rs	rem unsigned	$rd \leftarrow rt \% rs$
imm: 32-bit signed			
and	rd rt rs	AND	$rd \leftarrow rs \& rt$
andi	rd rt imm	and immediate	$rd \leftarrow rt \& imm$
or	rd rt rs	OR	$rd \leftarrow rt rs$
ori	rd rt imm	or immediate	$rd \leftarrow rt imm$
xor	rd rt rs	XOR	$rd \leftarrow rt \wedge rs$
xori	rd rt imm	xor immediate	$rd \leftarrow rt \wedge imm$
imm: 32-bit signed			
slt	rd rt rs	Set (1) if less than*	$rd \leftarrow (rt < rs) ? 1 : 0$
slti	rd rt imm	slt immediate	$rd \leftarrow (rt < imm) ? 1 : 0$
sltu	rd rt rs	slt unsigned	$rd \leftarrow (rt < rs) ? 1 : 0$
sltiu	rd rt imm	sltu immediate	$rd \leftarrow (rt < imm) ? 1 : 0$
imm: 32-bit. *: Unset (0) otherwise			

RARS RISC-V: Logic and arithmetic instructions

RISC-V instruction		Machine code					
Mne-monic	Operands	for-mat	opcode/ fun3,7	src	tgt	dst	imm
sll	rd rt rs	R	33/1,00	rs	rt	rd	
slli	rd rt num	I	13/1		rt	rd	num
srl	rd rt rs	R	33/5,00	rs	rt	rd	
srlr	rd rt num	I	13/5		rt	rd	num
sra	rd rt rs	R	33/5,20	rs	rt	rd	
srai	rd rt num	I	13/5		rt	rd	num*
num: (dec.) 0...31. *srai: set second bit of imm (imm=0x400+num)							
add	rd rt rs	R	33/0,00	rs	rt	rd	
addi	rd rt imm	I	13/0		rt	rd	num
sub	rd rt rs	R	33/0,20	rs	rt	rd	
mul	rd rt rs	R	33/0,01	rs	rt	rd	
mulh	rd rt rs	R	33/1,01	rs	rt	rd	
mulhu	rd rt rs	R	33/3,01	rs	rt	rd	
div	rd rt rs	R	33/4,01	rs	rt	rd	
divu	rd rt rs	R	33/5,01	rs	rt	rd	
rem	rd rt rs	R	33/6,01	rs	rt	rd	
remu	rd rt rs	R	33/7,01	rs	rt	rd	
num: 12-bit signed							
and	rd rt rs	R	33/7,00	rs	rt	rd	
andi	rd rt imm	I	13/7		rt	rd	imm
or	rd rt rs	R	33/6,00	rs	rt	rd	
ori	rd rt imm	I	13/6		rt	rd	imm
xor	rd rt rs	R	33/4,00	rs	rt	rd	
xori	rd rt imm	I	13/4		rt	rd	imm
imm: 12-bit signed							
slt	rd rt rs	R	33/2,00	rs	rt	rd	
slti	rd rt imm	I	13/2		rt	rd	imm
sltu	rd rt rs	R	33/3,00	rs	rt	rd	
sltiu	rd rt imm	I	13/3		rt	rd	imm
imm: 32-bit							

RARS RISC-V: Jump, branch and memory instructions

MIPS instruction		Meaning	
Mne- monic	Operands	Description	Operation
jal	rd addr	Jump and link	$pc \leftarrow addr, rd \leftarrow pc+4$
jalr	rd rt offs	jal register	$pc \leftarrow rt+offs, rd \leftarrow pc+4$

beq	rt rs addr	Branch if equal	$rt == rs ? pc \leftarrow addr$
bne	rt rs addr	Branch if not equal	$rt != rs ? pc \leftarrow addr$
bge	rt rs addr	Branch if greater/equal	$rt > rs ? pc \leftarrow addr$
bgeu	rt rs addr	bge unsigned	$rt > rs ? pc \leftarrow addr$

if condition is false: $pc \leftarrow pc+4$. See p. 345 for more branching instructions

lb	rd offs(rt)	Load byte	$rd \leftarrow M[rt+offs]$
lbu	rd offs(rt)	lb unsigned	$rd \leftarrow M[rt+offs]$
lh	rd offs(rt)	Load halfword	$rd \leftarrow M[rt+offs]$
lhu	rd offs(rt)	lh unsigned	$rd \leftarrow M[rt+offs]$
lw	rd offs(rt)	Load word	$rd \leftarrow M[rt+offs]$
sb	rs offs(rt)	Store byte	$rs \rightarrow M[rt+offs]$
sh	rs offs(rt)	Store halfword	$rs \rightarrow M[rt+offs]$
sw	rs offs(rt)	Store word	$rs \rightarrow M[rt+offs]$
lui	rd pattern	Load upper immediate	$rd \leftarrow pattern \ll 12$
auipc	rd pattern	Add upper imm. to pc	$rd \leftarrow pc + (pattern \ll 12)$

appropriate bits only, rest unchanged. M: memory. offs(et): 12 bit 2's-complement
pattern: 20-bit pattern

csrrw	rd csr rs	CSR read/write	$rd \leftarrow csr \leftarrow rs$
csrrs	rd csr rs	CSR read/set	$rd \leftarrow csr \leftarrow rs$
csrrc	rd csr rs	CSR read/clear	$rd \leftarrow csr \leftarrow rs$
csrrwi	rd csr imm	CSR read/write imm.	$rd \leftarrow csr \leftarrow imm$
csrrsi	rd csr imm	CSR read/set imm.	$rd \leftarrow csr \leftarrow imm$
csrrci	rd csr imm	CSR read/clear imm.	$rd \leftarrow csr \leftarrow imm$

read CSR before write CSR

Execution Control

ecall	System call	
ebreak	Breakpoint (pause)	
wfi	Wait for interrupt	
uret	Return from exception	
fence	Fence (sync. threads)	
fence.i	Fence i (sync. threads)	

RARS RISC-V: Jump, branch and memory instructions

MIPS instruction		Machine code					
Mne- monic	Operands	for- mat	opcode/ fun3	src	tgt	dst	imm
jal	rd addr	J	6f/			rd	(addr-pc)/2
jalr	rd rt offs	I	67/0		rt	rd	offs
offs(et): 12-bit signed value rt+offs: LSB set to 0							
imm: 20-bit (2's-compl) (addr-pc)/2 rearranged bits: 20,10-1,11,19-12							
beq	rt rs addr	S	63/0	rs	rt		addr-pc
bne	rt rs addr	S	63/1	rs	rt		addr-pc
bge	rt rs addr	S	63/5	rs	rt		addr-pc
bgeu	rt rs addr	S	63/7	rs	rt		addr-pc
addr: RARS code address label							
imm: 12-bit signed value in two fields. addr=pc+imm							
lb	rd offs(rt)	I	03/0		rt	rd	offset
lbu	rd offs(rt)	I	03/4		rt	rd	offs
lh	rd offs(rt)	I	03/1		rt	rd	offs
lhu	rd offs(rt)	I	03/5		rt	rd	offs
lw	rd offs(rt)	I	03/2		rt	rd	offs
sb	rs offs(rt)	S	23/0	rs	rt		offs
sh	rs offs(rt)	S	23/1	rs	rt		offs
sw	rs offs(rt)	S	23/2	rs	rt		offs
lui	rd pattern	U	37/			rd	pattern
auipc	rd pattern	U	17/			rd	pattern
appropriate bits only, rest unchanged. pattern: 20-bit pattern							
offs(et): 12-bit value (for S in two fields).							
csrrw	rd csr rs	I	73/1		rs	rd	csr
csrrs	rd csr rs	I	73/2		rs	rd	csr
csrrc	rd csr rs	I	73/3		rs	rd	csr
csrrwi	rd csr imm	I	73/5		imm	rd	csr
csrrsi	rd csr imm	I	73/6		imm	rd	csr
csrrci	rd csr imm	I	73/7		imm	rd	csr
csr: 12-bit value. imm: 5-bit value							

Execution Control

ecall *	I	73/0	0	0	0	0
ebreak	I	73/0	0	0	0	1
wfi	I	73/0	0	0	0	105
uret	I	73/0	0	0	0	2
fence	I	0f/0	0	0	0	0
fence.i	I	0f/1	0	0	0	0

*: Specify by a7. See appendix L

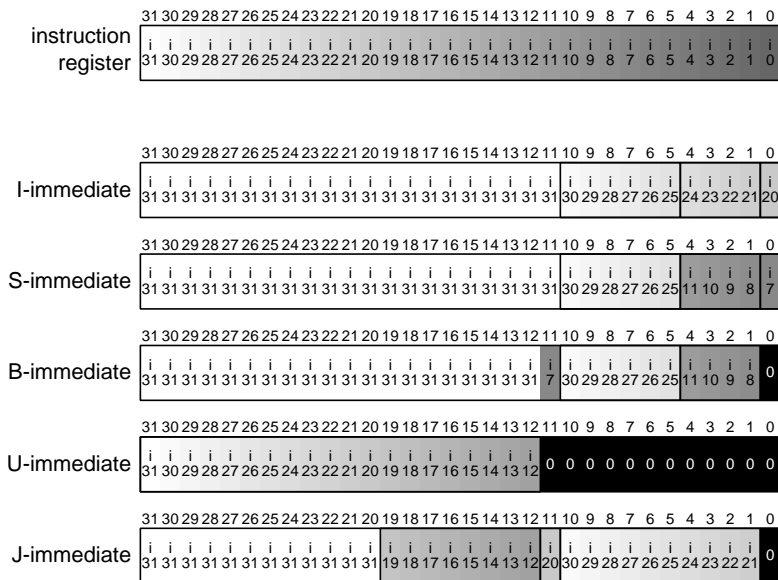
RISC-V: Floating-point instructions

MIPS instruction		Meaning	
Mnemonic	Operands	Description	Operation
flw	fd offs(rt)	Load single	fd←M[rt+offs]
fld	fd offs(rt)	Load double	fd←M[rt+offs]
M: memory. offs(et): 12-bit signed			
fsw	fs offs(rt)	Store single	fs→M[rt+offs]
fsd	fs offs(rt)	Store double	fs→M[rt+offs]
M: memory. offs(et): 12-bit signed			
fQ.SIZE	rd ft fs	FP compare	rd←(ft Q fs)?1:0
Q: le, lt, eq. SIZE: s, d			
fcvt.TO.FROM	fd fs	Convert	fd = T0(fs)
fcvt.TO.U	fd rs	Convert	fd = T0(rs)
fcvt.U.FROM	rd fs	Convert	rd = int(fs)
U: signed w, unsigned wu. T0, FROM: s, d			
fadd.SIZE	fd ft fs	FP add	fd←ft+fs
fsub.SIZE	fd ft fs	FP subtract	fd←ft-fs
fmul.SIZE	fd ft fs	FP multiply	fd←ft*fs
fdiv.SIZE	fd ft fs	FP divide	fd←ft/fs
fmadd.SIZE	fd ft fs fa	FP mul,add	fd←ft*fs+fa
fnmadd.SIZE	fd ft fs fa	FP neg,mul,add	fd←-(ft*fs+fa)
fmsub.SIZE	fd ft fs fa	FP mul,sub	fd←ft*fs-fa
fnmsub.SIZE	fd ft fs fa	FP neg,mul,sub	fd←-(ft*fs)+fa
fsqrt.SIZE	fd fs	FP square root	fd←√fs
fmin.SIZE	fd ft fs	FP min	fd←(ft<fs)?ft:fs
fmax.SIZE	fd ft fs	FP max	fd←(ft>fs)?ft:fs
fsgnT.SIZE	fd ft fs	FP sign injection	fd←ft,sign(fs)
fclass.SIZE	rd fs	FP classify	rd←class(fs)
SIZE: s, d. T: j:sign, jn:inverted, jx:xor. class: See p.344			
fmv.s.x	fd rs	Copy pattern	fd←rs
fmv.x.s	rd fs	Copy pattern	fs→rd

RISC-V: Floating-point instructions

MIPS instruction			Machine code						
Mnemonic	Operands	form	opc	fun3	imm	src	tgt	dst	
flw	fd offs(rt)	I	07	2	offs		rt	fd	
fld	fd offs(rt)	I	07	3	offs		rt	fd	
offs(et): 12-bit signed, see p.344									
Mnemonic	Operands	form	opc	fun3	imm	src	tgt	dst	
fsw	fs offs(rt)	S	27	2	offs	fs	rt		
fsd	fs offs(rt)	S	27	3	offs	fs	rt		
offs(et): 12-bit signed, see p.344									
Mnemonic	Operands	form	opc	fun5	fmt	rm	src	tgt	dst
fQ.SIZE	rd ft fs	F	53	14	SIZE	Q	fs	ft	rd
Q: le=00, lt=01, eq=10. SIZE: s=00, d=01.									
fcvt.T0.FROM	fd fs	F	53	08	T0	7	FROM	fs	fd
fcvt.T0.U	fd rs	F	53	1a	T0	7	U	rs	fd
fcvt.U.FROM	rd fs	F	53	18	FROM	7	U	fs	rd
U: signed w=0, unsigned wu=1. T0, FROM: s=00, d=01									
fadd.SIZE	fd ft fs	F	53	00	SIZE	7	fs	ft	fd
fsub.SIZE	fd ft fs	F	53	01	SIZE	7	fs	ft	fd
fmul.SIZE	fd ft fs	F	53	02	SIZE	7	fs	ft	fd
fdiv.SIZE	fd ft fs	F	53	03	SIZE	7	fs	ft	fd
fmadd.SIZE	fd ft fs fa	F	43	fa	SIZE	7	fs	ft	fd
fnmadd.SIZE	fd ft fs fa	F	4f	fa	SIZE	7	fs	ft	fd
fmsub.SIZE	fd ft fs fa	F	47	fa	SIZE	7	fs	ft	fd
fnmsub.SIZE	fd ft fs fa	F	4b	fa	SIZE	7	fs	ft	fd
fsqrt.SIZE	fd fs	F	53	0b	SIZE	7	0	fs	fd
fmin.SIZE	fd ft fs	F	53	05	SIZE	0	fs	ft	fd
fmax.SIZE	fd ft fs	F	53	05	SIZE	1	fs	ft	fd
fsgnT.SIZE	fd ft fs	F	53	04	SIZE	T	fs	ft	fd
fclass.SIZE	fd ft fs	F	53	1c	SIZE	1	0	fs	rd
SIZE: s=00, d=01. T: j=0, jn=1, jx=2									
fmv.s.x	fd rs	F	53	1e	0	0	0	rs	fd
fmv.x.s	rd fs	F	53	1c	0	0	0	fs	rd

Immediate-value construction based on instruction register bits:



Class(FP)	
rd bit	fs is
0	$-\infty$
1	negative normal
2	negative subnormal
3	-0
4	$+0$
5	positive subnormal
6	positive normal
7	$+\infty$
8	signaling NaN
9	quiet NaN

RARS RISC-V: (Some) pseudo-instructions

bgt	rt rs addr	Branch if greater than	$rt > rs ? pc \leftarrow addr$
	blt rs rt addr		
bgtu	rt rs addr	Branch if greater than	$rt > rs ? pc \leftarrow addr$
	bltu rs rt addr		
ble	rt rs addr	Branch if less/equal	$rt \leq rs ? pc \leftarrow addr$
	bge rs rt addr		
bleu	rt rs addr	Branch if less/equal	$rt \leq rs ? pc \leftarrow addr$
	bgeu rs rt addr		
beqz	rt addr	Branch if equal zero	$rt == 0 ? pc \leftarrow addr$
	beq rt x0 addr		
bnez	rt addr	Branch if not equal 0	$rt \neq 0 ? pc \leftarrow addr$
	bne rt x0 addr		
bltz	rt addr	Branch if less than 0	$rt < 0 ? pc \leftarrow addr$
	blt rt x0 addr		
bgtz	rt addr	Branch if greater than 0	$rt > 0 ? pc \leftarrow addr$
	blt x0 rt addr		
blez	rt addr	Branch if less or equal 0	$rt \leq 0 ? pc \leftarrow addr$
	bge x0 rt addr		
bgez	rt addr	Branch if greater or equal 0	$rt \geq 0 ? pc \leftarrow addr$
	bge rt x0 addr		

mv	rd rs	Copy integer	$rd \leftarrow rs$
	add rd x0 rs		
fmv.SIZE	fd fs	Copy s(ingle)/d(ouble)	$fd \leftarrow fs$
	fsgnj.SIZE fd fs fs		
nop		No operation	
	addi x0 x0 0		

j	addr	Jump	$pc \leftarrow addr$
	jal x0 addr		
jal	addr	Jump and link	$ra \leftarrow pc+4, pc \leftarrow addr$
	jal ra addr		
jr	rs offset	Jump register	$pc \leftarrow rs + offset$
	jalr x0 rs offset		
jalr	rd offset	Jump and link register	$ra \leftarrow pc+4, pc \leftarrow rd + offset$
	jalr ra rd offset		
call	addr	Call (jump-and-link)	$ra \leftarrow pc+4, pc \leftarrow addr$
	jal ra addr		
ret		Return (from call)	$pc \leftarrow ra$
	jalr x0 ra 0		

RARS RISC-V: (Some) pseudo-instructions (cont.)

lw	rd addr	Load word from address	rd←M[addr]
	auipc rd pc≫20 addi rd rd (addr-pc)&0000FFF		
li	rd word	Load immediate	rd←word
	lui rd word≫12 addi rd rd word & 0x00000FFF		
li	rd 12-bit	Load immediate	\$rd←12-bit
	addi rd x0 12-bit		
la	rd addr	Load address	rd←addr
	auipc rd pc≫20 addi rd rd (addr-pc)&0000FFF		

sgt	rd rt rs	Set if greater than	rd←(rt>rs)?1:0
	slt rd rs rt		
sgtu	rd rt rs	Set if greater than	rd←(rt>rs)?1:0
	sltu rd rs rt		
seqz	rd rt	Set if equal 0	rd←(rt==0)?1:0
	sltiu rd rt 1		
snez	rd rt	Set if not equal 0	rd←(rt!=0)?1:0
	sltu rd x0 rt		
sltz	rd rt	Set if less than 0	rd←(rt<0)?1:0
	slt rd x0 rt		
sgtz	rd rt	Set if greater than 0	rd←(rt>0)?1:0
	slt rd rt x0		

To be implemented by macros (see Section 10.9):

inc	rt	Increment	rt←rt+1
	addi rt rt 1		
dec	rt	Decrement	rt←rt-1
	addi rt rt -1		
push	rs	Push onto stack	M[--sp]←rs
	addi sp sp -4 sw rs 0(sp)		
pop	rd	Pop from stack	rd←M[sp++]
	lw rd 0(sp) addi sp sp 4		
done		Terminate	
	li a7 10 ecall		

L | (RARS) RISC-V system calls

function	a7	argument(s)	return value(s)
print integer	1	a0 = integer	
print float	2	fa0 = float	
print double	3	fa0 = double	
print string	4	a0 = address of null-terminated string	
read integer	5		a0 integer read
read float	6		fa0 float read
read double	7		fa0 double read
read string	8	a0 = address of buffer a1 = max. length	
malloc	9	a0 = amount (bytes)	a0 address
exit (terminate execution)	10		
print character	11	a0 = character	
read character	12		a0 character read
dir	17	a0 = write buffer a1 = buffer length	
close file	57	a0 = file descriptor	
read from file	63	a0 = file descriptor a1 = addr. input buffer a2 = max length	a0 number of chars read (0:end-of-file, <0:error)
write to file	64	a0 = file descriptor a1 = addr. output buffer a2 = number of chars	a0 number of chars written (<0: error)

function	a7	argument(s)	return value(s)
get time	30		a0 low 32 bits a1 high 32 bits
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 40	a0 = generator int a1 = seed int	
random int	41	a0 = integer	a0: next random int
random int in range	42	a0 = generator int a1 = upper limit	a0: next random int in range 0...a1-1
random float	43	a0 = generator int	fa0: 0.0...0.999...
random double	44	a0 = generator int	fa0: 0.0...0.999...

M | 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	^	126	7E	1111110	~
63	3F	0111111	?	95	5F	1011111	_	127	7F	1111111	delete

i | Index

#, 198
0x, 19
65xx, 267

4004, 263

accumulator, 129
action table, 99
ADC, 277
add, 204
AI, 299
ALU, 16, 127
and, 205
API, 320
Aristotle, 42
arithmetic and logic unit, 16, 127
arithmetic shift, 205
array, 217
artificial intelligence, 299, 301
ASCII, 16, 32, 349
.ascii, 198, 333
.asciiz, 198, 333
assembler directives, 196, 333
associative law, 45
asynchronous, 101
asynchronous computing, 294
Atmel, 276
audion, 5

back-side bus, 183
balanced ternary, 53

BCD, 18, 19
beq, 209
beqz, 209
BER, 186
Berkeley machine, 98, 99
bge, 209
bgez, 209
bgt, 209
bgtz, 209
big endian, 160
binary numbers, 16
binary-coded decimal, 19
BIOS, 180
bit, 150
bit-error rate, 186
bitcoin, 248
bitmap, 153
ble, 209
blez, 209
blockchain, 248
blt, 209
bltz, 209
bmp, 153
bne, 209
bnez, 209
(George) Boole, 43
Boole machine, 98, 99
Boolean algebra, 42
boot loader, 179
bootstrapping, 179

- brains, 296
- branching, 2, 206
- BSB, 183
- BSOD (blue screen of death), 283
- bus, 182
- byte, 158

- C (programming language), 1
- cache, 168
- callee, 232
- caller, 232
- carry, 112
- carry-look-ahead, 116
- central processing unit, 130
- Chinese room, 301
- chip-select, 122
- CMOS, 73
- code segment, 170, 196
- COM port, 177
- comment, 198
- Commodore 64, 272
- commutative law, 45
- comparator, 122
- computer, 1
- control logic, 130
- CPU, 129, 130
- CRC, 185

- D flip-flop, 95
- .data, 196
- data alignment, 160
- data register, 129
- data segment, 170, 196
- datapath, 130
- De Morgan's laws, 48, 82, 85
- Debian Linux, 321
- decimal system, 11
- declaration of variables, 217
- demultiplexer, 120
- deMUX, 120
- denormalized numbers, 139
- Descartes, 300
- destination operand, 196
- destructive read, 167

- Difference Engine, 3, 262
- digital electronics, 5
- DIL, 263
- diode, 4
- direct immediate, 178
- direct indexed, 178
- disassemble, 292
- distributive law, 46
- div, 204
- DMA, 169
- do-while loop, 211
- don't-care, 61, 66
- double (float), 138
- DRAM, 162
- dyadic, 45, 129
- dynamic memory allocation, 173

- Ebers-Moll, 5
- edge, 102, 103
- edge-triggered, 93, 94
- ELIZA, 300
- encoder, 121
- engineering notation, 35
- entropy, 151
- .eqv, 197, 333
- ergodic, 153
- Ethernet, 185
- exception, 180, 181
- excitation table, 97, 98, 100
- extended (float), 138
- external memory, 159

- factorial, 234
- falling-edge-triggered, 93
- FEC, 186, 187
- fifth-generation programming
 language, 9
- finite-state machine, 3, 99
- firmware, 180
- flag, 130
- flip-flop, 93
- floating point, 137, 223
- floating point division, 142
- floating point range, 141

- floating-point notation, 35
- floating-point numbers, 137
- FLOPS, 298
- for loop, 211
- forward error correction, 186, 187
- fourth-generation programming
 - languages, 8
- front-side bus, 183
- FSB, 183
- full-adder, 111, 113, 114
- functions, 2, 228
- fundamental gates, 78

- gated S/R-latch, 93
- gif, 153
- glitch, 88
- global variables, 174
- `.globl`, 333
- `goto`, 207
- Gray code, 33, 107

- half (float), 138
- half adder (HA), 54
- half-adder, 111
- half-adder (HA), 112
- Hamming, 186
- handshake, 299
- Harvard architecture, 169
- hazard, 88
- heap, 172, 200, 232
- heuristics, 168
- hexadecimal, 18
- hexal system, 11
- HI, 204
- Huntington postulates, 44, 47

- IDE, 9
- idempotent, 48
- IEEE 754, 137
- if ... then `goto`, 207
- imitation game, 299
- immediate, 196
- implicit, 207
- implied 1., 139
- infinity, 139
- information, 150
- initialization, 172, 217
- instruction register, 129
- integer division, 215, 217
- integer multiplication, 214
- Intel 4004, 263
- Intel x86, 282
- interrupt controller, 181
- interrupt handler, 180
- interrupt vector, 180
- interrupts, 180
- inverter, 73, 75, 79
- IR, 129
- IRQ, 180

- j, 207
- J/K flip-flop, 94, 95
- `jal`, 229
- jpeg, 153
- jpg, 153
- jump, 206
- jump-and-link, 229
- jump-to-register, 229

- Karatsuba, 136
- Karnaugh maps, 5, 59, 85
- (Jack) Kilby, 94
- Kleene logic, 53

- 1a, 202
- largest denormalized number, 140
- largest normalized number, 140
- latch, 92
- li, 200
- LIFO, 174
- little endian, 160
- L0, 204
- load address, 202
- load immediate, 200
- local variables, 174
- logic gates, 5
- logic shift, 205
- long division, 135

- loop, 2
- LSB, 18
- lw, 202

- machine code, 134
- machine language, 134
- macro code, 134
- macro-assembly, 8
- macros, 235
- magnetic core memory, 167
- mask, 214
- masking, 213
- MASM, 286
- material implication, 6
- Maya number system, 13
- MCU, 276
- Mealy machine, 98, 99
- memory mapping, 218
- memory-management unit, 176
- mfhi, 205
- mflo, 205
- MFS, 262
- micro-assembly, 8
- micro-controller, 276
- minimum feature size, 262
- minterm, 56
- MIPS file I/O, 204
- MIPS pseudo-instructions, 330, 345
- MMU, 176
- mnemonic, 195
- Moore machine, 98, 99
- Moore's law, 7
- MOS 65xx, 267
- move, 200
- MSB, 18
- mult, 204
- multiplexer, 107, 118
- multiplication, 134
- multiplication look-up table, 137
- MUX, 107, 118

- NaN, 139
- NAND-gate, 74, 76
- NASM, 287
- negative numbers, 29
- Newton-Raphson, 144
- nibble, 158
- no-operation, 160, 207
- non-linear electronics, 4
- nop, 160, 207, 226
- nor, 205
- NOR-gate, 76, 77
- normalized numbers, 139
- Northbridge, 183
- not a number, 139
- NOT-gate, 74
- number conversion, 25

- object-oriented programming, 169
- octal, 18
- octuple (float), 138
- odds, 152
- offset, 252
- Ohm's law, 4
- opamp, 293
- opcode, 131, 195
- operand, 196
- operating system, 170
- operational amplifier, 293
- or, 205
- overflow, 157
- overlay, 176, 277

- paging, 176, 208
- parity, 103, 185
- passing by reference, 2, 230
- passing by value, 2, 230
- PC, 129
- png, 153
- pointer, 2, 170
- pointers, 194
- polling, 182
- pop, 174, 232
- PoS, 55
- pow, 148
- powf, 148

- Priest's logic, 53
- Princeton architecture, 169
- product-of-sums, 55
- program counter, 129, 130
- pseudo instruction, 192, 200, 202, 209, 235
- push, 174, 232

- quadruple (float), 138
- quantum computing, 294
- qubit, 295
- Quine-McCluskey method, 64

- race condition, 96
- RAM, 159, 172
- random logic, 133
- random-access memory, 172
- RARS, 250
- recursive, 173
- recursive function, 234
- recursivity, 2
- redundant information, 185
- registers, 129
- reserving space, 170
- resistance, 72
- ripple-carry adder, 114, 115
- RISC, 1
- rising-edge-triggered, 93
- RJ45, 185
- rol, 206
- ROM, 165, 179
- Roman numbers, 14
- ror, 206
- rotate, 206
- Russian-peasant algorithm, 23, 214

- s-registers, 193, 232
- S/R-latch, 92, 93
- scientific notation, 35
- score, 12
- second-generation programming language, 7
- segmentation, 283

- sequencer, 98, 99
- Set theory, 42
- SHA-256, 249
- Shannon entropy, 151
- Shannon-Hartley channel capacity, 184
- shift, 205
- sign-magnitude, 15, 29
- significand, 139
- single (float), 138
- sll, 206
- sllv, 206
- slt, 209
- smallest denormalized number, 141
- smallest normalized number, 140
- SMD, 276
- SoP, 55
- source operand, 196
- Southbridge, 183
- .space, 333
- square root, 226
- sra, 206
- SRAM, 162
- srav, 206
- srl, 206
- srlv, 206
- stack, 172, 173, 228, 232
- state diagram, 102
- static memory allocation, 173
- static-0 hazard, 89
- static-1 hazard, 88
- status register, 130
- steady-state, 88
- strobe, 131
- struct, 217
- sub, 204
- sum-of-products, 55
- supercomputer, 298
- sw, 202
- syscall, 199
- system call, 180, 199, 321

- T flip-flop, 95

- t-registers, 193, 232
- target operand, 196
- Taylor expansion, 146
- ternary logic, 53, 112
- `.text`, 196
- third-generation programming
 - language, 8
- three-bit counter, 101
- toggle, 94
- transient behavior, 88
- transistor, 5, 72
- transition table, 99
- tri-state, 54, 79
- trit, 54, 112
- truth tables, 55
- (Alan) Turing, 299
- Turing test, 299
- twisted pair, 185

- UART, 176
- underflow, 157
- unsigned integer, 18

- USB, 185

- variable-length instructions, 264
- variables, 150, 217
- Vice, 272
- virtual memory, 176
- volatile memory, 164
- Von Neumann architecture, 130, 169
- Von Neumann bottleneck, 168

- while loop, 211
- `.word`, 196
- WYSIWYG, 33

- x86, 282
- xor, 205
- XOR-gate, 78

- YWIYGI, 33, 172, 197

- zero, 13, 139
- zero-page addressing, 267