UiO : Department of Informatics
University of Oslo

Computer architecture

Compendium for INF2270

Philipp Häfliger, Dag Langmyhr and Omid Mirmotahari
Spring 2013

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This compendium is intended to supply required background information to students taking the course INF2270. Together with the lectures and the problems (both the weekly and the mandatory ones) it defines the course curriculum.

One important aim of this course is to give an understanding of the various abstraction levels in a computer:

High-level programming language	Level 5
Assembly language	Level 4
Operating system	Level 3
Machine instructions	Level 2
Micro architecture	Level 1
Digital logic	Level 0

Figure 1.1: Abstraction levels in a computer

One part (part no I) of the course goes upwards from the bottom level to explain how computers are designed; the other (part no II) progresses downwards from the top level to describe how to program the computer at each level. At the end of the course, the two descriptions should meet somewhere around levels 2–3.

The authors would like to thank the following students for valuable contributions: André Kramer Orten and Marius Tennøe.

Part I

Basics of computer architecture

Chapter 2

Introduction to Digital Electronics

The word digital comes from the Latin word ‘digitus’ which means finger. Its meaning today is basically ‘countable’ and since many people use their fingers for counting, that explains the connection to its Latin origin. Its opposite is ‘analog’. Digital electronics refers to electronic circuits that are described by a discrete/countable number of states. The basic building block of almost all digital electronics today is the switch. This has two states, either ‘on’ or ‘off’, and almost all digital electronics today is thus binary, i.e., the number of states of the basic building block and basic signals is two.[1]

First predecessors of the modern computer have been build with mechanical switches (The Analytical Engine by Charles Babbage in 1837), electro mechanical switches/relays (G. Stibitz’ Model-K (1937) and K. Zuse’s Z3 (1941)), and vacuum tubes (ENIAC, 1946). But the veritable computer revolution took off with a sheer incredible miniaturization of a switch: The transistor.

The first transistor based programmable computer has been reported at the university of Manchester in 1953 with ca. 600 transistors. From then on, Moore’s law has kicked in which describes the exponential progression of the miniaturization and sophistication of computers by predicting a doubling of the number of transistors of a central processing unit (CPU, the core of every computer today) every two years. Thus, a state of the art CPU today consists of, for example, 731 million transistors (Intel Core™ i7 Quad Extreme). Once you read this statement, it will most likely already be outdated.

Most electronics today uses so called complementary metal oxide silicon (CMOS) field effect transistors (FET), which are depicted with their schematic symbol in figure 2.1. For digital purposes they do behave almost ideally like a switch. If one were to look closer, however, one would realize that this is quite a simplification, and if one is to tune a digital circuit to its performance limits or even construct analog circuits, this closer look becomes necessary. Be that as it may, for most digital designs the description as a switch has proved to be to a large degree sufficient.

[1] The next most popular number of states for the basic elements is three and there exist a number of ternary electronic circuits as well.

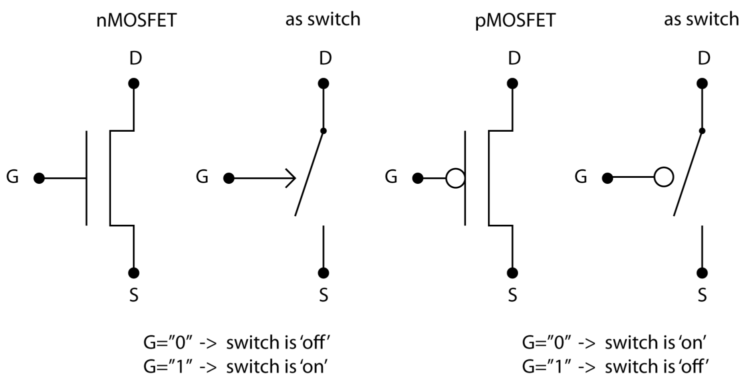


Figure 2.1: Schematic symbol and of CMOS FETs together with a symbol showing them as switches. nMOSFET to the left, pMOSFET to the right.

CMOS transistors can today be realized on a two-dimensional layout measuring 28nm in length, and maybe (educated guess) 50nm in width. With minimal distance requirements between devices, the actual area needed is somewhat larger, but still smaller than our imagination is able to picture. If one wants to attempt to imagine even smaller numbers: the thinnest layer used in building up the transistor in the third dimension is now below 2nm thick, actually a crystal (SiO_2) consisting only of a few atomic layers (ca. 10-20).

With this extreme miniaturization comes also extreme speed. A CMOSFET needs only in the order of hundreds of pico seconds or even less to switch. This allows the high frequencies of several GHz at which CPUs are clocked.

So digital electronics consists of binary switches that control signals that in turn control other switches. The resulting binary signals are refered to as *bits* and are well suited to represent the binary numbers '1' and '0' or the logic states 'true' and 'false'.

Chapter 3

Binary Numbers

3.1 Unsigned Binary Numbers

The numbers we use in everyday life are decimal numbers. The main reason for us to use the decimal system is that we have 10 fingers. The decimal system uses an alphabet of 10 digits: [0123456789]. When writing down a decimal number, the rightmost digit has a unit value of 1 (or 10^0), the next to the left has a unit value of 10 (10^1), the next 100 (10^2) and so on. The number 18 thus means:

$$18 := 1 \times 10^1 + 8 \times 10^0 \tag{3.1}$$

If humankind had but two fingers, things might have turned out quite differently.[1] A binary number system might have evolved, with an alphabet of only 2 digits: [01]. The rightmost digit would again have a 'unit' value of 1 (2^0), but the next would have a unit value of 2 (2^1) and then 4 (2^2), 8 (2^3), 16 (2^4)etc. 18 reads now:

$$10010 := 1 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 \tag{3.2}$$

3.2 Signed Binary Numbers

3.2.1 Sign and Magnitude

If one wants to represent negative integers with binary numbers, a first intuitive solution would be to use a 'sign bit', i.e., the first bit of a binary number indicates a negative number if it is 1 or a positive number if it is 0 and the rest of the bits encode the magnitude of the number. This is known as 'sign and magnitude' encoding.

For example, 8 bit numbers could encode the values from -127 to 127 (7-bit magnitude and 1 sign-bit):

[1] We might never have become intelligent enough to compute, because of the inability to use tools, for instance. Horses, with only two digits to their forelimbs, do (to our knowledge) not have a number system at all.

$$\begin{aligned} 87 &= 01010111 \\ -87 &= 11010111 \end{aligned}$$

A first problem with this scheme is that there is also a 'signed zero', i.e., $+0$ and -0, which is redundant and does not really make sense.

3.2.2 Two's Complement

The *two's complement* (used in most digital circuits today) is a signed binary number representation that does not suffer from the problem of a signed zero and it comes with a few extremely convenient properties. In 8-bit two's complement the unsigned numbers 0 to 127 represent themselves, whereas the unsigned numbers 128 to 255 (all numbers with the first bit='1') represent the numbers -128 to -1 (in other words: read it as an unsigned number and subtract 256 to get the signed value). Thus, also in this representation all numbers with the first bit equal to '1' are negative numbers.

$$\begin{aligned} 87 &= 01010111 \\ -41 &= 11010111 \quad (= 215 - 256) \\ -87 &= 10101001 \quad (= 169 - 256) \end{aligned}$$

One of these convenient properties is the construction of the inverse of a number in two's complement. The same operation is performed for both, positive to negative and negative to positive:

1) invert each bit
2) add 1

This is not quite so simple as in 'sign and magnitude' representation, but still simple.

Example:

1.	87 =	01010111	→	10101000	
2.		10101000+1	=	10101001	= -87
1.	-87 =	10101001	→	01010110	
2.		0 1010110+1	=	01010111	= 87

3.3 Addition and Subtraction

The most convenient property, however, is the simple addition of two's complement numbers, be they negative or positive. This is achieved by simply adding the two numbers as if they were unsigned binary numbers. If the result would be one digit longer, that digit is simply ignored. Surprisingly at first, the result is correct also if the numbers are regarded as two's complement numbers. An exception is the case in which the result of the summation of two n-bit numbers would lie outside the range of an n-bit two's complement number, e.g., when using 8-bits and adding 120 + 118 = 238, which is above the maximal value 127 that can be represented with an 8-bit two's complement number.

Here are some examples:

signed op	equivalent unsigned op	mod 256	signed res
-41-87	215+169 = 384	128	-128
87-41	87+215 = 302	46	46

Why does that work? The key to understanding this is the modulo operation.

Let us consider two positive numbers a and b that can be represented as binary numbers with n bits, i.e., in the range of $[0, 2^n - 1]$ and the numbers a' and b' in the range of $[-2^{n-1}, 2^{n-1} - 1]$ which are the numbers that are represented by the same bit-patterns but interpreted as two's complement binary numbers.

Remember that per definition:

$$a' = \begin{cases} a - 2^n & \text{if} \quad a \in [2^{n-1}, 2^n - 1] \\ a & \text{if} \quad a \in [0, 2^{n-1} - 1] \end{cases} \tag{3.3}$$

A first key-concept is that ignoring an eventual overflow/carry bit of the result of an addition $a + b$ corresponds to computing a modulo with 2^n on the result. Thus, when adding two 8 bit numbers and the result would be 9 bits long, but the 9th bit is ignored, this is equivalent to performing a modulo 256 operation on the result.

A second concept now is that $a' \mod 2^n = a \mod 2^n = a$, since adding or subtracting 2^n does not change the result of the $\mod 2^n$ operation (remember that a' is either the same as a or $a - 2^n$) and a number that is within the range $[0, 2^n - 1]$ is not changed by the $\mod 2^n$ operation.

Yet a third concept is that it does not matter if one computes the $\mod 2^n$ operation of a sum only on the result or also on the summands, i.e., $(a \mod 2^n + b \mod 2^n) \mod 2^n = (a + b) \mod 2^n$

Thus, it follows:

$$\begin{aligned} (a' + b') \mod 2^n &= (a' \mod 2^n + b' \mod 2^n) \mod 2^n \\ &= (a + b) \mod 2^n \\ &= (a + b)' \mod 2^n \end{aligned} \tag{3.4}$$

What this equation says is that for the operation of addition of two two's complement numbers one can also just add their unsigned interpretation, ignore an overflow/carry bit if it occures (modulo operation) and then interpret the result as two's complement. The result is correct, provided it would not exceed the range of an n bit two's complement number.

An example thereof if $n = 8$, $a = 188$ and $b = 241$: It follows that $a' = -68$ and $b' = -15$. Substituting these numbers in the equation (3.4) above:

$$\begin{aligned} (-68-15) \mod 256 &= (188+241) \mod 256 \\ &= 429 \mod 256 = 173 \\ &= -83 \mod 2^n \end{aligned} \tag{3.5}$$

That convenient property is really good news for the design for arithmetic operations in digital hardware, as one does not need to implement both addition and subtraction, since adding a negative number is the same as subtracting. A subtraction can be performed by

1) inverting the number that is to be subtracted (by inverting every bit individually and adding 1, see section 3.2.2) and
2) adding it to the number it is supposed to be subtracted from

3.4 Multiplication and Division

Multiplication with a factor of two of a binary number is simply achieved by shifting the individual bits by one position to the left and inserting a '0' into the rightmost position (referred to as the 'least significant bit' or just LSB). This works for both unsigned and two's complement representation, again provided that the result does not lie beyond the range that can be represented with n-bits.

If you accept that this works for unsigned binaries, one can show this to be true for a negative two's complement binary a' number with the corresponding unsigned interpretation a because:

$$2a' \mod 2^n = 2(a-2^n) \mod 2^n = 2a - 2*2^n \mod 2^n = 2a-2^n \mod 2^n \tag{3.6}$$

A division with a factor of 2 is a shift of all the bits by one position to the right. Note that if the leftmost (the 'most significant bit' or just MSB) bit is filled in with a copy of its state before the shift (This is known as *arithmetic* right shift), again, this works for both unsigned and signed (two's complement) binary numbers, but note that the result is rounded towards $-\infty$ and not towards zero, e.g., right-shifting -3 results in -2.

Examples:

decimal	binary	shifted	decimal
-3	1101	1110	-2
-88	10101000	11010100	-44

A multiplication with 2^k can accordingly be achieved by a left shift by k positions and a division by 2^k with an arithmetic right shift by k positions.

A general multiplication or division can be achieved by splitting it up into a sum of products with 2^k $k \in [0, n-1]$. For example if a and b are represented as a binary number $(a_{n-1}, \ldots, a_0)$ and $(b_{n-1}, \ldots, b_0)$ where a_i stands for a one bit variable. Then

$$a * b = \sum_{k=0}^{n-1} a_k * 2^k * b \tag{3.7}$$

So as an algorithm:

1) Initialize the result binary number r to zero.
2) Add b to r if the MSB of a is ‘1’.
3) Shift r and a to the left.
4) Repeat steps 2) and 3) n times.

3.5 Extending an n-bit binary to n+k bits

A last remark on manipulating binary numbers will explain how to extend the number of bits by which a number is represented in two’s complement. Analogous to an arithmetic right shift one needs to fill in the extra bits with a copy of the former MSB, thus negative numbers are extended with extra 1’s on the left and positive numbers with extra 0’s. A simple explanation based on what we learnt previously, is that this operation is equivalent to extending the number first by adding k zeros to the right, i.e., multiply it with 2^k and then dividing it by 2^k by shifting it by k positions to the right using an arithmetic shift.

Examples:

decimal	4 bit		8 bit
-2	1110	→	11111110
-5	1011	→	11111011
5	0101	→	00000101

Chapter 4
Boolean Algebra

Digital electronics can conveniently be used to compute so called *Boolean functions*, formulated using Boolean algebraic expressions, which are also used in propositional logic. These are functions that project a vector of binary variables onto one (or a vector of) binary variable(s):

$$f_{\text{Boolean}} : \mathrm{B}^k \to \mathrm{B} \text{ where } \mathrm{B} = 0,1 \tag{4.1}$$

In this context one interprets the result often as either 'true' or 'false' rather than '1' or '0', but that does not change anything for the definition of Boolean functions: it's just a renaming of the variables' alphabet.

There are three basic operators in Boolean algebra: NOT, AND, OR. Different notations are sometimes used:

NOT a	$\neg$ a	$\bar{a}$	a'	
a AND b	a $\wedge$ b	a $\times$ b	a$\cdot$b	(do not confuse with multiplication!)
a OR b	a $\vee$ b	a + b		(do not confuse with addition!)

Boolean functions can be defined by truth tables, where all possible input combinations are listed together with the corresponding output. For the basic functions the truth tables are given in table 4.1.

More complicated functions with more input variables can also be defined as truth tables, but of course the tables become bigger with more inputs and more and more impractical. An alternative form to define Boolean functions are Boolean expressions, i.e., to write down a function by combining Boolean variables and operators (just as we are used to with other mathematical functions). An example:

$$f(a,b,c) = a + b \cdot (a + c) \tag{4.2}$$

There are several popular quite basic Boolean functions that have their own operator symbol but are derived from the basic operators:

a XOR b = $(\mathrm{a} \cdot \bar{\mathrm{b}}) + (\bar{\mathrm{a}} \cdot \mathrm{b})$

a XNOR b = $(\mathrm{a} \cdot \mathrm{b}) + (\bar{\mathrm{a}} \cdot \bar{\mathrm{b}})$

a	$\bar{a}$
0	1
1	0

a	b	a·b
0	0	0
0	1	0
1	0	0
1	1	1

a	b	a+b
0	0	0
0	1	1
1	0	1
1	1	1

Table 4.1: Truth tables for the basic Boolean functions

a·b+c = (a·b)+c	a+b·c = a+(b·c)	(priority)
a·b = b·a	a+b = b+a	(commutativity)
(a·b)·c=a·(b·c)	(a+b)+c=a+(b+c)	(associativity)
$\bar{\bar{a}}$=a		(involution)
a·$\bar{a}$=0	a+$\bar{a}$=1	(completness)
a·a=a	a+a=a	(idempotency)
a·1=a	a+0=a	(boundedness)
a·0=0	a+1=1	(boundedness)
a·(a+b)=a	a+(a·b)=a	(absorbtion)
a·(b+c)=(a·b)+(a·c)	a+(b·c)=(a+b)·(a+c)	(distributivity)
$\overline{a+b} = \bar{a} \cdot \bar{b}$	$\overline{a \cdot b} = \bar{a} + \bar{b}$	(deMorgan)

Table 4.2: Table of rules that govern Boolean functions

for NOR:

a	b	$\overline{a+b}$	$\bar{a} \cdot \bar{b}$
0	0		
0	1		
1	0		
1	1		

for NAND:

a	b	$\overline{a \cdot b}$	$\bar{a} + \bar{b}$
0	0		
0	1		
1	0		
1	1		

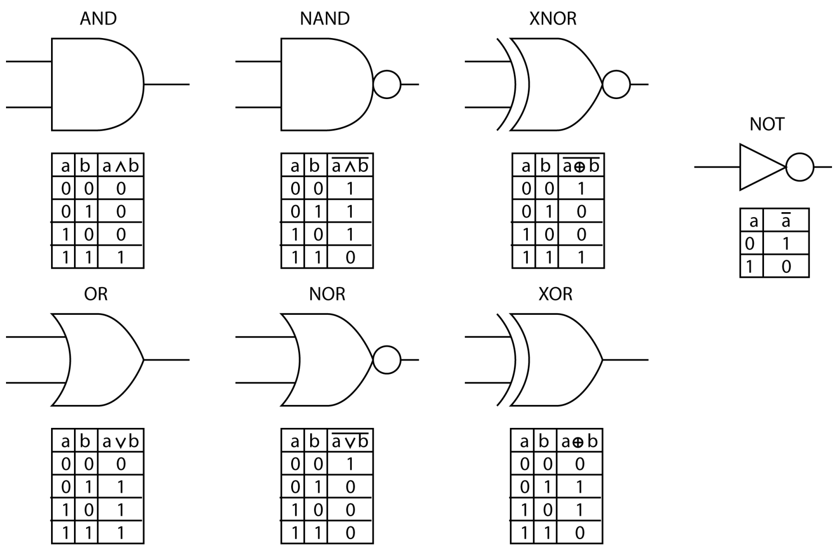
Table 4.3: Exercise to verify deMorgan

a NAND b = $\overline{a \cdot b}$

a NOR b = $\overline{a + b}$

Table 4.2 lists basic rules that govern Boolean functions and that allow to rearrange and simplify them. Note that the equal sign '=' connects two functions that are equivalent, i.e., for every input the output is exactly the same. Equivalent functions can be written in any number of ways and with any degree of complexity. Finding the simplest, or at least a reasonably simple expression for a given function is a very useful goal. It makes the function easier to read and 'understand' and, as we will see later on, reduces the complexity (number of electronic devices, power consumption, delay) of digital electronics that implements the function.

To verify the deMorgan theorem one can fill in the truth tables in table 4.3, and here are two examples on how to apply the rules of table 4.2 to simplify functions:



a	b	$a \wedge b$
0	0	0
0	1	0
1	0	0
1	1	1

a	b	$\overline{a \wedge b}$
0	0	1
0	1	1
1	0	1
1	1	0

a	b	$\overline{a \oplus b}$
0	0	1
0	1	0
1	0	0
1	1	1

a	$\bar{a}$
0	1
1	0

a	b	$a \vee b$
0	0	0
0	1	1
1	0	1
1	1	1

a	b	$\overline{a \vee b}$
0	0	1
0	1	0
1	0	0
1	1	0

a	b	$a \oplus b$
0	0	0
0	1	1
1	0	1
1	1	0

Figure 4.1: Equivalent Boolean operators, truth tables and logic gates

Example 1:

$$\begin{aligned} & a \cdot b + a \cdot \bar{b} \\ = \; & a \cdot (b + \bar{b}) \\ = \; & a \cdot 1 \\ = \; & a \end{aligned}$$

Example 2:

$$\begin{aligned} & a \cdot b \cdot c + \bar{a} \cdot b \cdot c && + && \bar{a} \cdot b \cdot \bar{c} \cdot (a + c) \\ = \; & (a + \bar{a}) \cdot b \cdot c && + && \bar{a} \cdot b \cdot \bar{c} \cdot a + \bar{a} \cdot b \cdot \bar{c} \cdot c \\ = \; & 1 \cdot b \cdot c && + && 0 + 0 \\ = \; & b \cdot c \end{aligned}$$

Applying the rules one can also show that either the NAND or the NOR function is actually *complete*, i.e., they are sufficient to derive all possible Boolean functions. This can be shown by showing that all three basic functions can be derived from a NAND or NOR gate, again employing the rules from table 4.2:

$$\bar{a} = \overline{a \cdot a} = \overline{a + a} \tag{4.3}$$

$$a \cdot b = \overline{\overline{a \cdot b}} = \overline{\bar{a} + \bar{b}} \tag{4.4}$$

$$a + b = \overline{\bar{a} \cdot \bar{b}} = \overline{\overline{a + b}} \tag{4.5}$$

Beyond truth tables and Boolean expressions, Boolean functions can also be expressed graphically with logic gates, i.e., the building blocks of digital electronics. Figure 4.1 summarizes the basic and derived functions and the corresponding operators, logic gates and truth tables. The logic gates will be our main tools as we move on to designing digital electronics. Note that they are somewhat more powerful than Boolean expression and can do things beyond implementing Boolean functions, since one can also connect them in circuits containing loops. These loops can be employed to realize elements that autonomously maintain a stable state, i.e., memory elements. But for a while still, we will stick with pure feed-forward circuits, and thus, Boolean functions.

4.1 Karnaugh maps

Karnaugh maps (or just K-maps) offer a way to use the human ability to find graphical patterns to aid systematically in simplifying Boolean functions. Consider the following example of a Boolean function and its truth table.

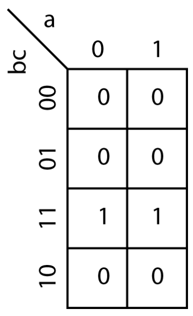
$$F = a \cdot b \cdot c + \bar{a} \cdot b \cdot c + \bar{a} \cdot b \cdot \bar{c} \cdot (a + c) \longrightarrow$$

a	b	c	F
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

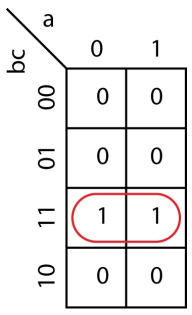
The truth table can now be shown in a socalled Karnaugh map (or K-map), where the outputs are arranged in a array and the axes of the array are labeled with the inputs arranged in a Gray-code, i.e., such that only one input bit shifts between columns/rows:

a	b	c	F
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

$\longrightarrow$



Now one needs to find the so called *minterms* graphically: rectangles that contain 2^n '1's (i.e., 1, 2, 4, 8, ... elements). The goal is to find a minimal number of rectangles that are maximal in size that cover all '1's in the array. They may overlap, and this is even desirable to increase their size. They may also wrap around the edge of the array (see next example!). In this first example this is quite trivial as there are only two '1's that conveniantly are neighbours and thus form a 1×2 rectangle (marked in red).



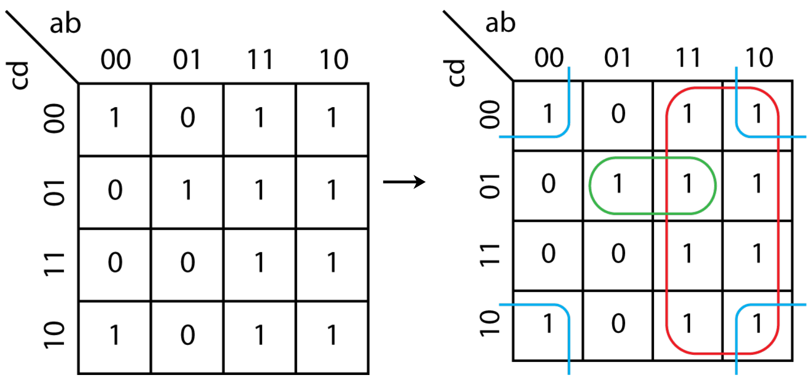
Now for this entire rectangle, all inputs are either constant or undergo all possible binary combinations with each other. Here, variables b and c are constant, and a goes through both its states '0' and '1'.

Now, the rectangles are used to form subexpressions of the *constant* variables combined with AND. In our case: $b \cdot c$. This is somewhat intuitive, since the condition to 'create' the rectangle of output '1's is that both b and c be true. If b or c would be constant '0' they would appear inverted, i.e., $\bar{b}$ or $\bar{c}$. If there would be several rectangles/minterms they would be connected with an OR.

The result for our first example is thus:

$$b \cdot c$$

Let us look at a somewhat more complete example. We'll start directly with the Karnaugh map:



Note the overlap between the 1×2 green and red 4×2 rectangles and the blue 2×2 rectangle is formed by wrapping around the edges of the array. The resulting simple Boolean function is as follows. The brackets are colour coded to correspond to the marked rectangles. Note that the bigger the rectangle, the shorter the minterm expression.

$$(a) + (\bar{b} \cdot \bar{d}) + (b \cdot \bar{c} \cdot d)$$

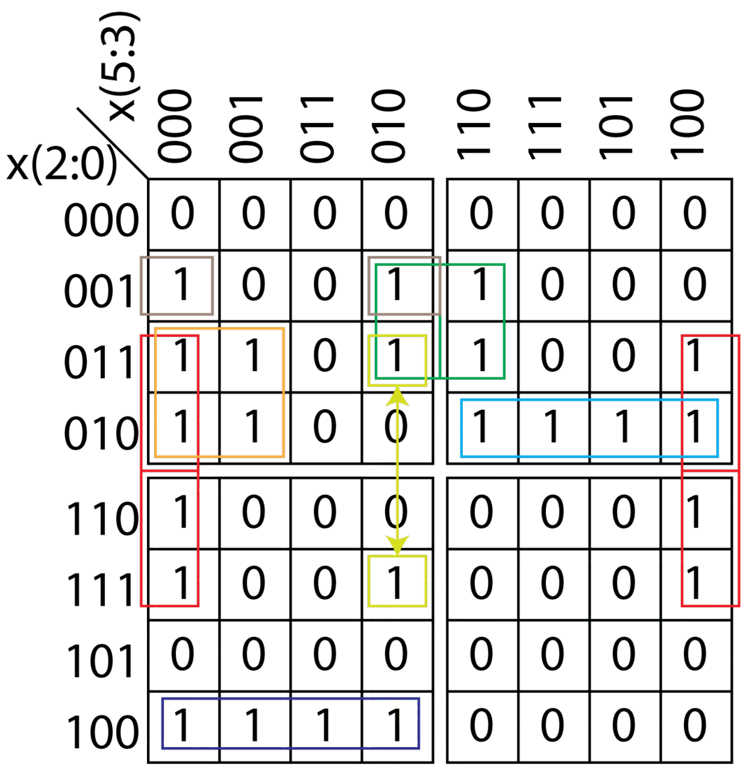


Figure 4.2: 6-bit Karnaugh map

4.1.1 Karnaugh maps with 5 and 6 bit variables

The method shown so far works well up to four input variables, i.e., up to two bits along one axis. Things become more complicated for more input bits. For 5 to 6 input bits, the Karnaugh map becomes 3-dimensional. The property of the 2-bit Gray-code, that any of the two bits of any group of 1, 2 or 4 subsequent codes are either constant or go through both their possible states in all possible combinations with an eventual 2nd non-constant bit, is not maintained in a Gray code with three bits. (Do not worry if you have not understood the last sentence ;-) as long as you understand the resulting method.) Consequently, instead of having a 3-bit Gray code along one axis, a third axis is added to the Karnaugh map and one has now to look for 3D-cuboids with 2^n elements instead of 2D rectangles. Since the 3D map is hard to display on a 2D sheet of paper, the different levels are shown side by side. Classically, the levels are unfolded along one side, so that one has to look for matching rectangles of 1's that are mirrored, as shown in figure 4.2. In this way, it is still a Gray code along the sides. More modern would be to simply change the most significant bit and to copy the Gray code for the two lesser bits. Then one would not need to look for mirrored patterns but patterns of the same shape in the same position in the two (or four) neighbouring squares. The solution for this example is:

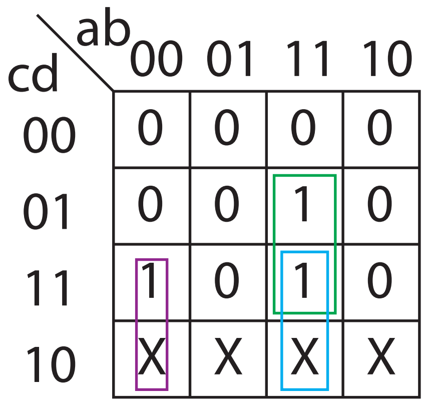


Figure 4.3: K-map with 'X's

$$
\begin{aligned}
&(\overline{x_4} \cdot \overline{x_3} \cdot x_1) \\
&+ \\
&(x_4 \cdot \overline{x_3} \cdot \overline{x_2} \cdot x_0) \\
&+ \\
&(x_5 \cdot \overline{x_2} \cdot x_1 \cdot \overline{x_0}) \\
&+ \\
&(\overline{x_5} \cdot x_2 \cdot \overline{x_1} \cdot \overline{x_0}) \\
&+ \\
&(\overline{x_5} \cdot \overline{x_4} \cdot \overline{x_2} x_1) \\
&+ \\
&(\overline{x_5} \cdot \overline{x_3} \cdot \overline{x_2} \cdot \overline{x_1} \cdot x_0) \\
&+ \\
&(\overline{x_5} \cdot x_4 \cdot \overline{x_3} \cdot x_1 \cdot x_0)
\end{aligned}
\tag{4.6}
$$

From 7 to 8 bit variables the problem becomes 4-Dimensional and the human ability to see patterns starts to be in trouble and other methods for simplification are used.

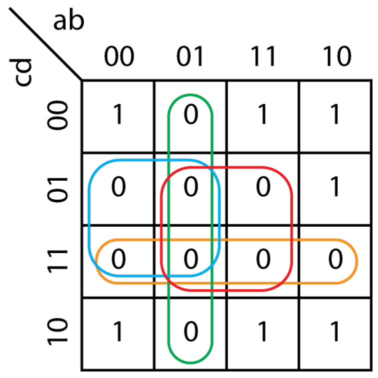
4.1.2 Karnaugh map simplification with 'X's

Some function definitions might contain 'X's as outputs for specific inputs that are of no concern for the particular application. Thus, the output for these cases can be both '1' or '0' with no consequence whatsoever for the intended application. Those 'X's become a kind of Joker in the K-map: you can use them just like '1's to make bigger groups of the '1's that are there. Check the example in figure 4.3. The resulting minterms are:

$$F = (\bar{a} \cdot \bar{b} \cdot c) + (a \cdot b \cdot d) + (a \cdot b \cdot c) \tag{4.7}$$

4.1.3 Karnaugh map simplification based on zeros

If a Karnaugh map contains more zeros than ones, it might be worthwhile to use the zeros instead of the ones to find a simplified expression. For this, one can find rectangles of '0's the same way as '1's. Now by imagining that all '0's are '1' and vice versa one can find an expression for the inverse $\overline{F}$ of the function F that is described by the Karnaugh map. By inverting the whole 'sum' of 'products' and employing deMorgan one can then deduce F as a product of sums. Consider the following example:



One can now deduce $\overline{F}$ as:

$$\overline{F} = (b \cdot d) + (\overline{a} \cdot d) + (\overline{a} \cdot b) + (c \cdot d) \tag{4.8}$$

Employing deMorgan's theorem:

$$\overline{\overline{F}} = F = (\overline{b} + \overline{d}) \cdot (a + \overline{d}) \cdot (a + \overline{b}) \cdot (\overline{c} + \overline{d}) \tag{4.9}$$

In short, if one wants to obtain the end result directly, one takes the inverse of the input variables that are constant for each rectangle to form the minterms as 'OR-sums' and combines these with ANDs.

Chapter 5

Combinational Logic Circuits

Combinational logic circuits are logic/digital circuits composed of feed-forward networks of logic gates (see figure 4.1) with no memory that can be described by Boolean functions.[1]

Logic gates (figure 4.1) are digital circuits that implement Boolean functions with two inputs and one output and are most often implemented to operate on binary voltages as input and output signals:[2] a certain range of input voltage is defined as 'high' or logic '1' and another range is defined as 'low' or '0'. E.g., in a digital circuit with a 1.8V supply one can, for instance, guarantee an input voltage of 0V to 0.5V to be recognised as '0' and 1.2V to 1.8V as '1' by a logic gate.

On the output side the gate can guarantee to deliver a voltage of either >1.75V or <0.05V.

That means that a small mistake at the input of a logic gate is actually 'corrected' at its output which is again closer to the theoretically optimal values of exactly 0V and 1.8V. These safety margins between input and output make (correctly designed!) digital circuits very robust (which is necessary with millions of logic gates in a CPU, where a single error might impair the global function!).

Some term definitions that we are going to use:

Design of a digital circuit is the process of assembling circuit blocks to form a bigger digital circuit.

Analysis of a digital circuit is the process of finding out what it is doing, e.g., (in the case of combinational logic!) by finding an equivalent Boolean function or a complete truth table.

[1] Note what is implied here: logic gates can *also* be connected in ways that include feed-back connections that implement/include *memory* that *cannot* be described as Boolean functions! This is then not 'combinational logic', but 'sequential logic', which will be the topic of chapter 6.

[2] Another possibility is to use socalled 'current mode' logic circuits where the logic states are represented with currents.

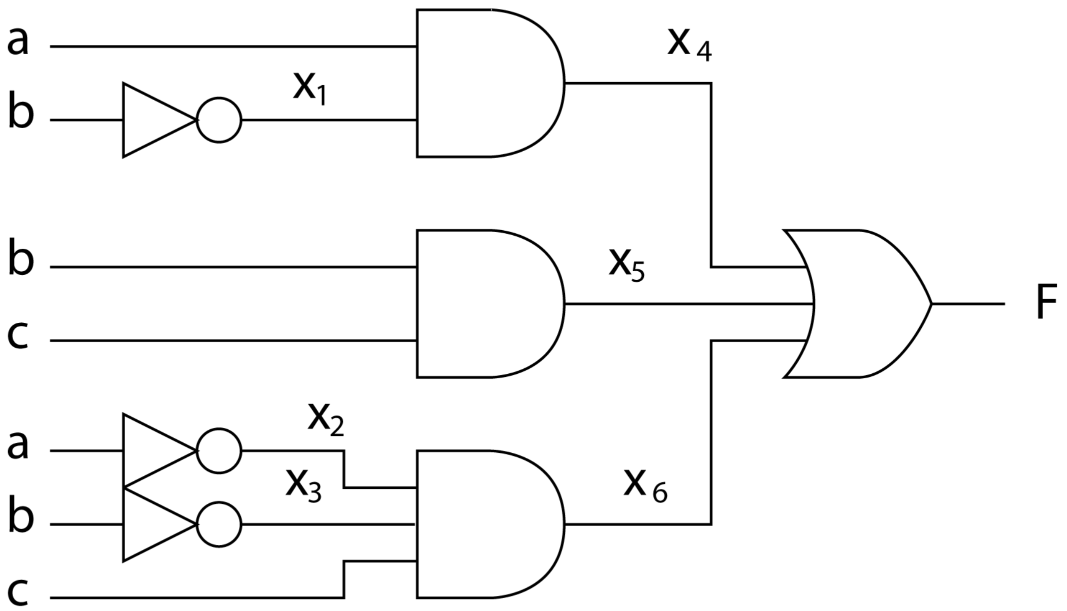


Figure 5.1: An example combinational logic circuit

A complete analysis is quite trivial for small digital circuits but neigh impossible for circuits of the complexity of a modern CPU. Hierarchical approaches in design and analysis provide some help.

The first Pentium on the market actually had a mistake in its floating point unit. Thus, it has been exposed to some ridicule. Here is a common joke of that time:

> After the Intel 286 there was the 386 and then the 486, but the 585.764529 was then dubbed 'Pentium' for simplicity sake.

Consider the example of a combinational logic circuit in figure 5.1. It can be analysed by finding an equivalent Boolean expression, i.e., find equivalent partial expressions for all the electrical nodes 'x_i' and finally for the output node F. The result is:

$$\underbrace{a \cdot \bar{b}}_{x_4} + \underbrace{b \cdot c}_{x_5} + \underbrace{\bar{a} \cdot \bar{b} \cdot c}_{x_6} \tag{5.1}$$

5.1 Standard Combinational Circuit Blocks

Some combinational circuits blocks are repeatedly used in logic circuit design and are often just given as 'black boxes' that provide a known function. Inside these boxes there are a number of equivalent ways to implement them on the logic gate level, even though *equivalent* on a functional level might still result in different performance regarding delay and power consumption or how easy they are scalable (i.e., extendable to handle wider multi-bit input and output signals)

Examples of such standard combinational higher level building blocks are:

- encoder/decoder

I_7	I_6	I_5	I_4	I_3	I_2	I_1	I_0	O_2	O_1	O_0
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	1	0	1
0	1	0	0	0	0	0	0	1	1	0
1	0	0	0	0	0	0	0	1	1	1

Table 5.1: Truth table of a 3-bit encoder

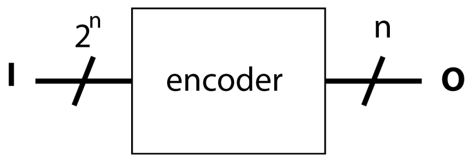


Figure 5.2: Encoder Symbol

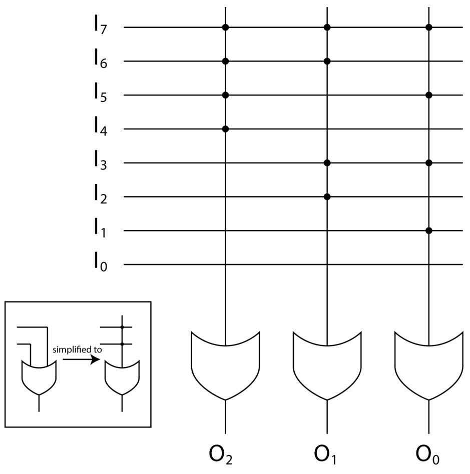


Figure 5.3: A possible implementation of a 3-bit encoder

- multiplexer/demultiplexer
- adder/multiplier
- ⋮

Note that the symbols for those blocks are not as much standardized as the symbols for the basic logic gates and will vary throughout the literature. The symbols given here are, thus, not the only ones you will encounter in other books but will be used throughout this text.

5.1.1 Encoder

An *encoder* in digital electronics refers to a circuit that converts 2^n inputs into n outputs, as specified (for a 3-bit encoder, i.e., $n = 3$) by the truth

I_7	I_6	I_5	I_4	I_3	I_2	I_1	I_0	O_2	O_1	O_0
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	X	0	0	1
0	0	0	0	0	1	X	X	0	1	0
0	0	0	0	1	X	X	X	0	1	1
0	0	0	1	X	X	X	X	1	0	0
0	0	1	X	X	X	X	X	1	0	1
0	1	X	X	X	X	X	X	1	1	0
1	X	X	X	X	X	X	X	1	1	1

Table 5.2: Complete truth table of a 3-bit priority encoder that encodes the highest active bit

I_2	I_1	I_0	O_7	O_6	O_5	O_4	O_3	O_2	O_1	O_0
0	0	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	1	0	0	0	0	0	0	1	0	0
0	1	1	0	0	0	0	1	0	0	0
1	0	0	0	0	0	1	0	0	0	0
1	0	1	0	0	1	0	0	0	0	0
1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0

Table 5.3: 3-bit decoder truth table

table 5.1. The input should be a 'one-hot' binary input, i.e., a bit-vector where only one bit is '1' and all others are '0'. The output then encodes the position of this one bit as a binary number. Note, that the truth table, thus, is not complete. It does not define the output if the input is *not* a one-hot code. Be aware that there are totally valid implementations of encoders that behave as defined if the input is a legal one-hot code, but they may react differently to 'illegal' inputs.

A symbol that is used for an encoder is given in figure 5.2 and a variant on how to implement a 3-bit encoder is depicted in figure 5.3. This particular (rather straight forward) implementation will produce quite arbitrary outputs when given 'illegal' inputs.

There are other implementations that adhere to a more strict definition of an encoder's behaviour. The *complete* truth table 5.2 defines such a behaviour. It is referred to as a *priority* encoder: always the highest order bit that is '1' is encoded. Note that the 'X's stand for 'don't care' and may be set to either '0' or '1' without influencing the output. We will not discuss a circuit implementation of this function, however.

5.1.2 Decoder

A *decoder* is the inverse function of an encoder, in digital circuits usually decoding n inputs into 2^n outputs. The truth table for a 3 bit variant is

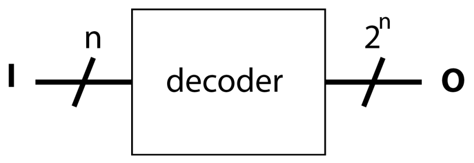


Figure 5.4: Decoder symbol

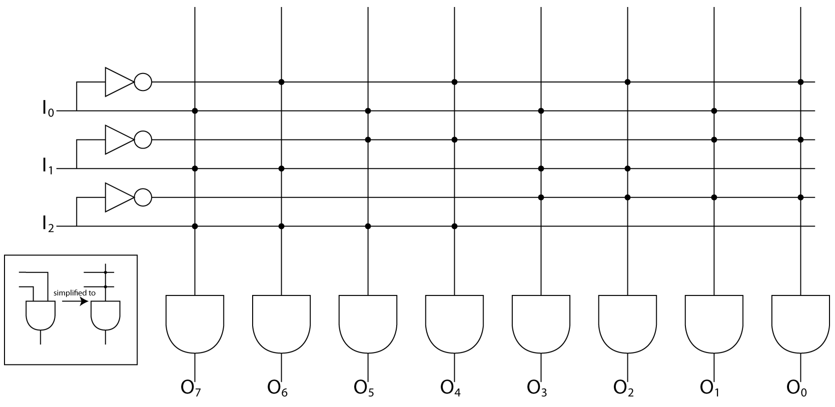


Figure 5.5: Possible 3-bit decoder implementation

S_2	S_1	S_0	O
0	0	0	I_0
0	0	1	I_1
0	1	0	I_2
0	1	1	I_3
1	0	0	I_4
1	0	1	I_5
1	1	0	I_6
1	1	1	I_7

Table 5.4: Multiplexer truth table

given in table 5.3. Note that the truth table is complete, not subjected to the same ambiguity as the decoder. A decoder symbol is shown in figure 5.4 and a possible 3-bit implementation in figure 5.5.

5.1.3 Multiplexer

A *multiplexer* routes one of 2^n input signals as defined by the binary control number **S** to the output. A schematics symbol that is used for a multiplexer is shown in figure 5.6. The truth table of a 3 bit multiplexer in figure 5.4 does not only contain zeroes and ones any longer but also the input variables I_k indicating that the output will depend on the input and the control bits S choose *which* input bit the output depends on. Figure 5.7 shows a possible implementation. Note the way that multiple input logic gates are shown in a simplified, compact way as explained in the small sub-figures.

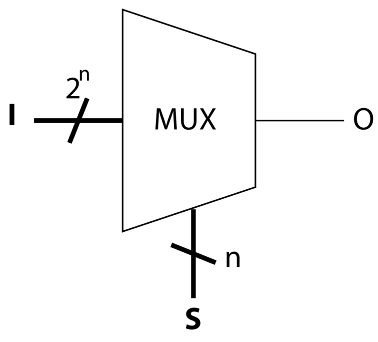


Figure 5.6: A multiplexer symbol

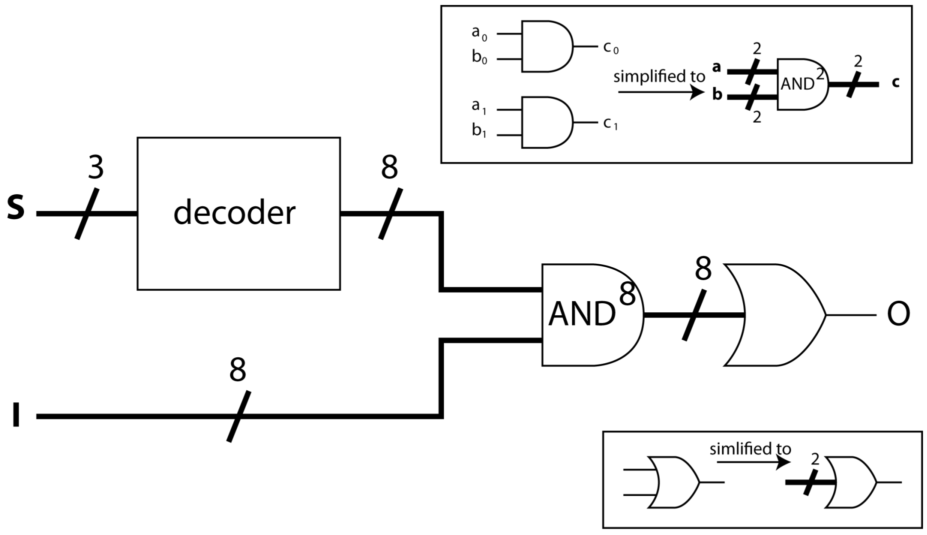


Figure 5.7

Multiplexers are used in many a context, for example when buses (parallel or serial data lines, see later in this text) are merged.

5.1.4 Demultiplexer

A *demultiplexer* performs the inverse function of a multiplexer, routing one input signal to one of 2^n outputs as defined by the binary control number **S**. Table 5.5 is the corresponding truth table, figure 5.8 is a possible symbol and figure 5.9 shows a possible implementation.

Demultiplexer find their use where a shared data line is used to convey data to several destinations at different times.

5.1.5 Adders

Addition of binary numbers is a basic arithmetic operation that computers execute innumerable times which makes the combinational adder circuit very important.

S_2	S_1	S_0	O_7	O_6	O_5	O_4	O_3	O_2	O_1	O_0
0	0	0	0	0	0	0	0	0	0	I
0	0	1	0	0	0	0	0	0	I	0
0	1	0	0	0	0	0	0	I	0	0
0	1	1	0	0	0	0	I	0	0	0
1	0	0	0	0	0	I	0	0	0	0
1	0	1	0	0	I	0	0	0	0	0
1	1	0	0	I	0	0	0	0	0	0
1	1	1	I	0	0	0	0	0	0	0

Table 5.5: 3-bit demultiplexer truth table

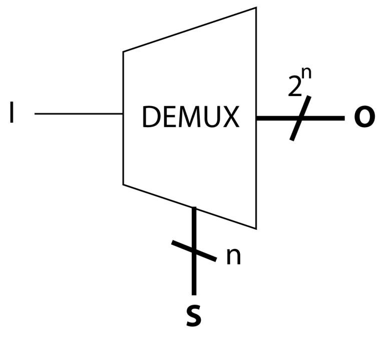


Figure 5.8: Demultiplexer symbol

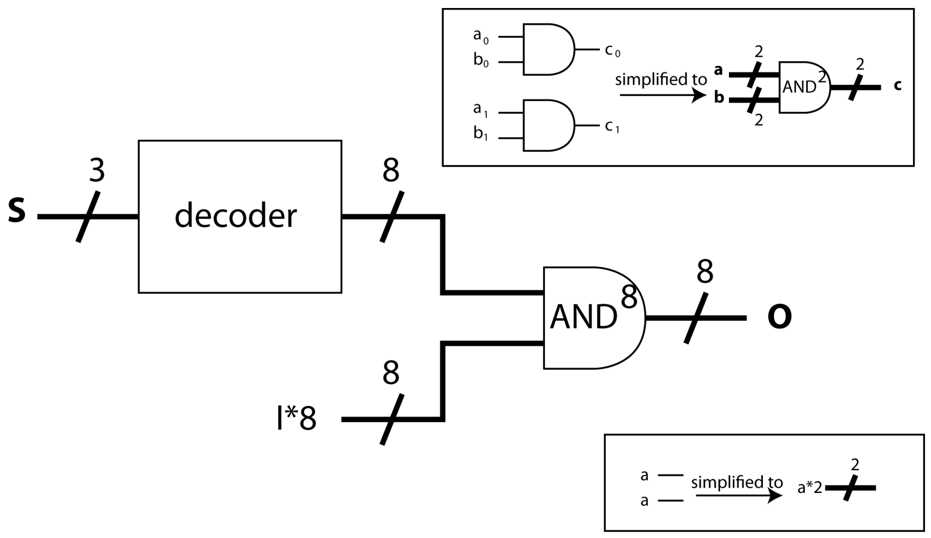


Figure 5.9: Possible 3-bit demultiplexer implementation

a	b	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Table 5.6: Truth table for a 1-bit half adder

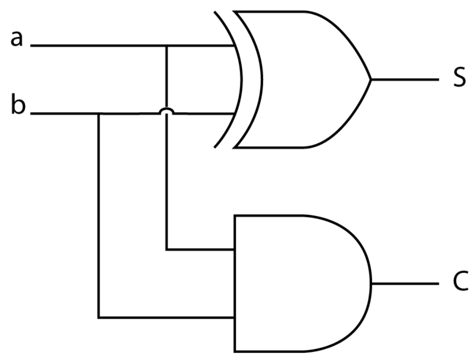


Figure 5.10: Schematics/circuit diagram of a 1-bit half adder

C_{in}	a	b	S	C_{out}
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

Table 5.7: Full Adder truth table

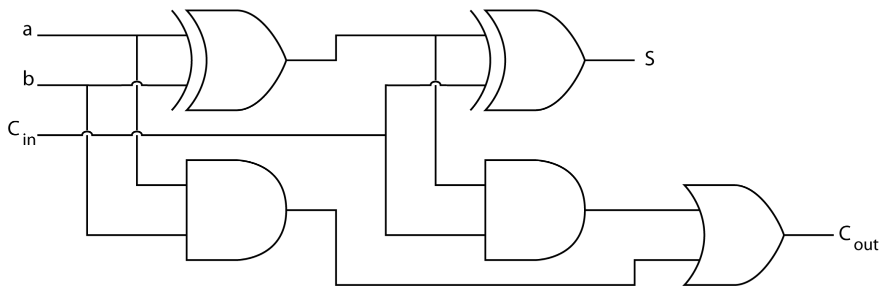


Figure 5.11: Full adder schematics

5.1.5.1 Half Adder

A *half adder* can add two 1-bit binary numbers. One bit binary numbers can code the values '0' and '1'. If two are added, the result may be either '0', '1', or '2'. The later coded as a binary number is '10'. Thus the result may require a digit more to be represented as a binary number, so the output of the half adder consists of two bits. the MSB is denoted as *carry* bit. The truth table of this addition is given in table 5.6 and the circuit that implements it in figure 5.10.

5.1.5.2 Full Adder

A half adder cannot be cascaded to a binary addition of an arbitrary bit-length since there is no carry *input*. An extension of the circuit is needed, a socalled *full adder*. Its truth table is given in table 5.7. The carry input bit is in essence just another input to add, on par with the two other inputs.

Thus, the result can now be either ‘0’, ‘1’, ‘2’/‘10’, or ‘3’/‘11’, still up to two bits in binary representation. The circuit implementation basically consists of two half adders, where the second receives the result of the first as one input and the carry in as the other. The circuit implementation is shown in figure 5.11. The full adder can be cascaded to add any length of binary number, connecting the carry out of a stage to the next higher order stage/more significant bit. The first stage’s carry in should be set to zero, or the first stage might simply consist of a half adder. This implementation of an adder is known as *ripple carry adder*, since the carry bits may ‘ripple’ from the LSB to the MSB and there might be a significant delay until the MSB of the result and its carry out become stable.

Chapter 6

Sequential Logic Circuits

Sequential logic circuits go beyond the concept of a Boolean function: they contain internal memory elements and their output will also depend on those internal *states*, i.e., on the input *history* and not just the momentary input.

6.1 Flip-Flops

Flip-flops are digital circuits with two stable, self-maintaining states that are used as storage/memory elements for 1 bit. The term 'flip-flop' refers in the more recent use of the language more specifically to *synchronous* binary memory cells (e.g., D-flip-flop, JK-flip-flop, T-flip-flop). These circuits change their state only at the rising edge (or falling edge) of a dedicated input signal, the *clock* signal. The term 'latch' (e.g., D-latch, SR-latch) is used for the simpler more basic *asynchronous* storage elements that do not have a dedicated clock input signal and may change their state at once if an input changes, but this naming convention is not consequently applied throughout the literature.

Often, sequential logic circuits are described using flip-flops as a further basic building block besides the logic gates that have been introduced so far, and that is what will be done in this compendium too, for the most part. However, be aware that flip-flops are themselves composed of logic gates that are connected in feedback loops and we will just briefly touch on the basic principle here, with one specific flip-flop: the D-latch.

The behaviour of a flip-flop can be expressed with a *characteristic table*: a truth table expressing the relation between the input and the present state, and the next state. An alternative is the *characteristic equation* which defines the dependency of the next state on the input and present state as a Boolean expression. See the following definitions of the flip-flop types for examples.

6.1.1 Asynchronous Latches

Asynchronous digital circuits in general, are circuits whose state changes are not governed by a dedicated clock signal, i.e., they can change state any

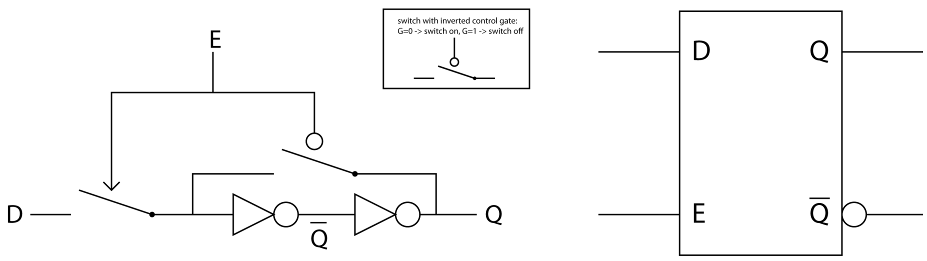

switch with inverted control gate:
G=0 -> switch on, G=1 -> switch off
E
D
Q̄
Q
D
Q
E
Q̄

Figure 6.1: Gated D-latch/transparent latch

time as an immediate consequence of a change of an input. See also below for an explanation of *synchronous* circuits, since a more concise definition of ‘asynchronous’ is ‘not synchronous’.

The design of more complex asynchronous circuits can be very challenging. One has to be very careful about signal timing, avoiding race conditions (the badly controlled order of changes of input signals causing unintentional effects), self maintaining loops of sequential state changes (that’s unintentional oscillators), deadlocks (states that cannot be changed anymore by any combination of inputs (That’s also a problem of synchronous circuits!)), ... On the other hand, asynchronous circuits can be very fast, since a new state is computed as quickly as the hardware allows[1].

We’ll limit our discussion of asynchronous sequential circuits to nothing more advanced than asynchronous latches, mostly, in this compendium.

6.1.1.1 Gated D-Latch/Transparent Latch

The *D-latch* is the simplest flip-flop type. *Gated* (in contrast to *clocked*, see section 6.1.2) means that the output state may change with an input signal while a gating signal (‘E’ in figure 6.1) is high and does no more change when the gating signal is low (or vice versa).

The D-latch’s behaviour is defined by the *characteristic equation* (6.1):

$$Q_{\text{next}} = D \cdot E + \bar{E} \cdot Q_{\text{present}} \tag{6.1}$$

Note that often the subscripts ‘present’ and ‘next’ are not explicitly written but it is assumed that the left hand side of the equation refers to the next state and the right hand to the present. This will also be applied in this compendium.

Figure 6.1 shows a possible implementation of the D-latch and its symbol. The double inverter feedback loop is the classic implementation of a binary memory cell. It has two possible states: Q is either equal to 1 and $\overline{Q}$ is equal to 0 or vice versa. Once the feedback loop is connected, that state has no way to change, but if the feedback loop is open, then Q and $\overline{Q}$ will simply be dependent on the input D. Thus the name ‘transparent latch’, that is also sometimes used, since the latch will simply convey the input to the output up until E is drawn low, whereupon the last state of D just before that event is stored.

[1] Think of this as a digital circuit that is automatically ‘overclocked’ to its possible limit, for those of the readers that have been into this ;-)

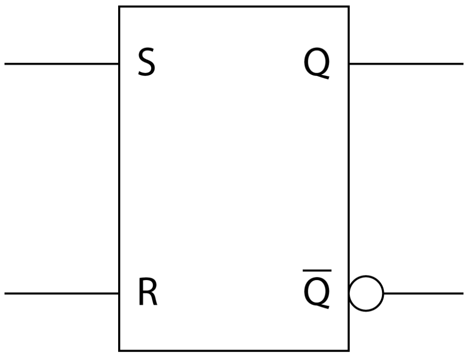


Figure 6.2: SR-latch symbol

S	R	Q	Q_{next}
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	?
1	1	1	?

Table 6.1: Full SR-latch characteristic table

S	R	Q
0	0	Q
0	1	0
1	0	1
1	1	?

Table 6.2: Abbreviated SR-latch characteristic table

6.1.1.2 SR-latch

Another asynchronous latch is the *SR-latch*. The symbol is shown in figure 6.2. We will not look at its internal workings but define its behaviour with the characteristic table 6.1. These tables can often be written more compactly by again useing variables of inputs and/or (implicitly 'present') states in the table (table 6.2).

In words, the SR-latch can be asynchronously *set* ($Q \rightarrow 1$ and $\overline{Q} \rightarrow 0$) by signal 'S' and *reset* ($Q \rightarrow 0$ and $\overline{Q} \rightarrow 1$) by signal 'R'. While both 'S' and 'R' are low, the state is maintained. Note the unique feature of the question mark in the characteristic table! They are caused by an 'illegal' input configuration, i.e., when both 'S' and 'R' are high. The basic definition of a general SR-latch does not define what the output should be in this case and different implementations are possible that will behave differently in that situation. If a circuit designer uses an SR-latch as a black box, he cannot rely on the output, if he permits this situation to occur.

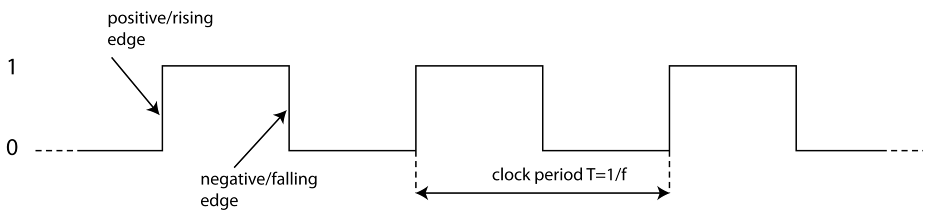


Figure 6.3: Clock signal

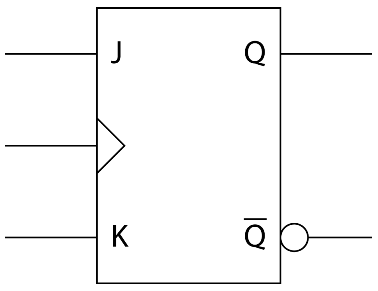


Figure 6.4: JK-flip-flop symbol

The SR-latches behaviour expressed with a characteristic equation but without correctly covering the uncertainty of the output in case of the illegal input situation(!):

$$Q = S + \bar{R} \cdot Q \tag{6.2}$$

6.1.2 Synchronous Flip-Flops

Synchronous digital circuits have a dedicated input signal called *clock* (CLK). State changes of synchronous circuits will only occur in synchrony with a change of this clock signal, i.e., either at the rising or falling edge of the clock (figure 6.3). A clock signal toggles back and forth between 0 and 1 with a regular frequency, the inverse of which is the clock *period* or clock *cycle*. In circuit symbols the clock signal is often marked with a triangle just inside of the clock pin. If the pin is connected to the symbol with a circle in addition the falling edge of the clock will be used for synchronization, otherwise it's the rising edge.

6.1.2.1 JK-Flip-Flop

The *JK-flip-flop* is the synchronous equivalent to the SR-latch. 'J' corresponds to 'S', but since it's synchronous, a change of 'J' from low (0) to high (1) will not immediately set the flip-flop, i.e., rise the output 'Q'. This will only happen later, at the very moment that the clock signal 'C' rises (provided that 'J' is still high!). Correspondingly, if 'K' is 1 when the clock signal changes to 1, the flip-flop is reset and 'Q' goes low, and if both 'J' and 'K' are low, the state does not change. The 'illegal' input state of the SR-latch, however, is assigned a new functionality in the JK-flip-flop: if both 'J' and 'K' are high, the flip-flop's output will *toggle*, i.e., Q will change state and become 1 if it was 0 and vice versa. This behaviour is defined in the characteristic tables tables 6.3 and 6.4 and/or by the characteristic equation:

J	K	Q_t	Q_{t+1}
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

Table 6.3: Full JK-flip-flop characteristic table

J	K	Q_{t+1}
0	0	Q_t
0	1	0
1	0	1
1	1	$\overline{Q}_t$

Table 6.4: Abbreviated JK-flip-flop characteristic table

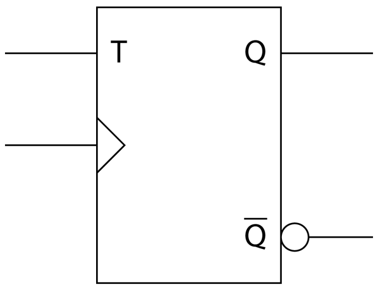


Figure 6.5: T-flip-flop symbol

$$Q = J \cdot \overline{Q} + \overline{K} \cdot Q \tag{6.3}$$

Note that the characteristic equation of synchronous flip-flops and other sequential circuits implicitly assumes that state changes only occur in synchrony with the clock!

6.1.2.2 T-Flip-Flop

The *T-flip-flop* (toggle flip-flop) is a reduced version of the JK-flip-flop, i.e., the signals J and K are shorted and named 'T'. So this flip-flop either maintains it state when T is 0 or it toggles, i.e., changes its state at the start of each clock cycle, when T is 1.

Its symbol is depicted in figure 6.5 and its characteristic table in tables 6.5 and 6.6. The characteristic equation is:

$$Q = T \oplus Q \tag{6.4}$$

T	Q_t	Q_{t+1}
0	0	0
0	1	1
1	0	1
1	1	0

Table 6.5: Full T-flip-flop characteristic table

T	Q_{t+1}
0	Q_t
1	$\overline{Q_t}$

Table 6.6: Abbreviated T-flip-flop characteristic table

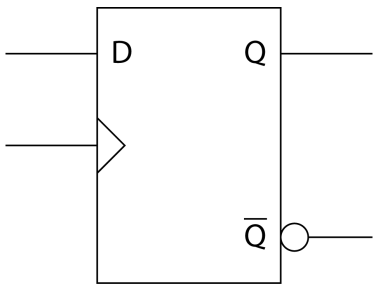


Figure 6.6: D-flip-flop symbol

A typical use for the T-flip-flop is in the design of counters.

6.1.2.3 D-Flip-Flop

The *D-flip-flop* (symbol in figure 6.6) can also be seen as a reduced version of the JK-flip-flop, this time if J is connected to K through an inverter and J is named 'D': the output of the D-flip-flop follows the input D at the start of every clock cycle as defined in the characteristic tables 6.7 and 6.8. Its characteristic equation is:

$$Q = D \tag{6.5}$$

It can in general be used to synchronize an input signal to an internal clock cycle for further synchronous processing circuits. Thus, it's also often used to build synchronous finite state machines (see section 6.2) that use their internal state stored in several D-flip-flops as input to combinational logic that in turn are connected again to the inputs of the D-flip-flops. This way one does not have to worry about the signal timing within the combinational logic as it's ensured that the states of the D-flip-flops are only allowed to change once the combinational logic is finished with its computation. If you do not understand this statement just yet, do not worry but try again after having read the next section on finite state machines.

D	Q_t	Q_{t+1}
0	0	0
0	1	0
1	0	1
1	1	1

Table 6.7: Full D-flip-flop characteristic table

D	Q_{t+1}
0	0
1	1

Table 6.8: Abbreviated D-flip-flop characteristic table

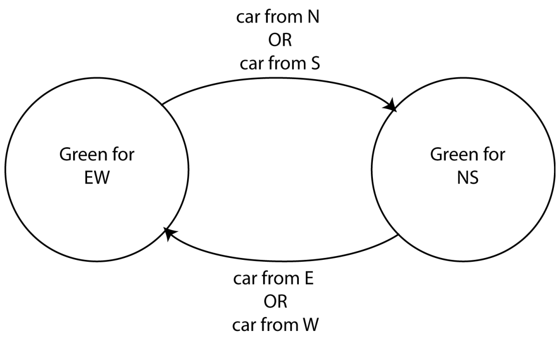


Figure 6.7: State transition graph for a simple traffic light controller equipped with sensors that detect cars waiting coming from the north, the south, the west, or the east

6.2 Finite State Machines

Finite State Machines (FSM) are a formal model suited to describe sequential logic, i.e., logic circuits whose output does not only depend on the present input but on internal memory and thus, on the history or sequence of the inputs. They describe circuits composed of combinational logic and flip-flops.

6.2.1 State Transition Graphs

A common way to describe/define a FSM is by a *state transition graph*: a graph consisting of the possible *states* of the FSM represented as bobbles and *state transitions* represented as arrows that connect the bobbles. The state transitions are usually labeled with a *state transition condition*, i.e., a Boolean function of the FSM inputs.

Consider the simple example in figure 6.7. It defines a controller for a traffic light, where pressure senors in the ground are able to detect cars waiting coming from either of the four roads. There are two states of this system, either north-south or east-west traffic is permitted. This FSM is governed by a slow clock cycle, let's say of 20 seconds. Equipped with sensors, the controller's behaviour is somewhat more clever than simply switching back and forth between permitting east-west and north-south traffic every cycle:

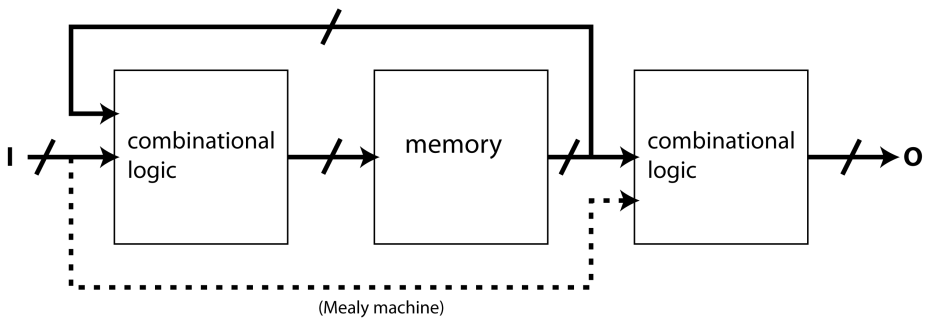


Figure 6.8: The principle block diagram model of a Moore FSM. If the dashed connection is included, it becomes a Mealy FSM.

it only switches, if there are cars waiting in the direction it switches to and will not stop the cars travelling in the direction that is green at present otherwise.

This FSM is an example of a so called Moore finite state machine. FSMs are often categorized as either Moore- or Mealy machines. These two models differ in the way the output is generated. Figure 6.8 illustrates the distinction.

Moore FSM: In a *Moore machine* the output depends solely on the internal states. In the traffic light example here, the traffic lights are directly controlled by the states and the inputs only influence the state transitions, so that is a typical Moore machine.

Mealy FSM: In a *Mealy machine* the outputs may also depend on the input signals directly. A Mealy machine can often reduce the number of states (naturally, since the 'state' of the input signals is exploited too), but one needs to be more careful when designing them. For one thing: even if all memory elements are synchronous the outputs too may change asynchronously, since the inputs are bound to change asynchronously.

That brings us to a further differentiation of FSMs: they can be implemented both asynchronously or synchronously.

Synchronous FSM: In general, it is simpler to design fully *synchronous FSMs*, i.e., with only synchronous flip-flops that all receive the same global clock signal. The design methods and especially the verification methods of the design are much more formalized and, thus, easier to perform.

Asynchronous FSM: On the other hand, *asynchronous FSM* implementations are potentially a good deal faster, since a state transition can occur as quickly as the state transition condition can be computed by the combinational logic, whereas a clocked FSM has to chose the clock period long enough such that the slowest of all possible state transition condition computation can be completed within a clock cycle. The design and verification, however, is tremendously more difficult and full of pitfalls.

car EW	car NS	go NS	go_{next} NS
0	0	0	0
1	0	0	0
0	1	0	1
1	1	0	1
0	0	1	1
1	0	1	0
0	1	1	1
1	1	1	0

Table 6.9: Traffic light controller characteristic table

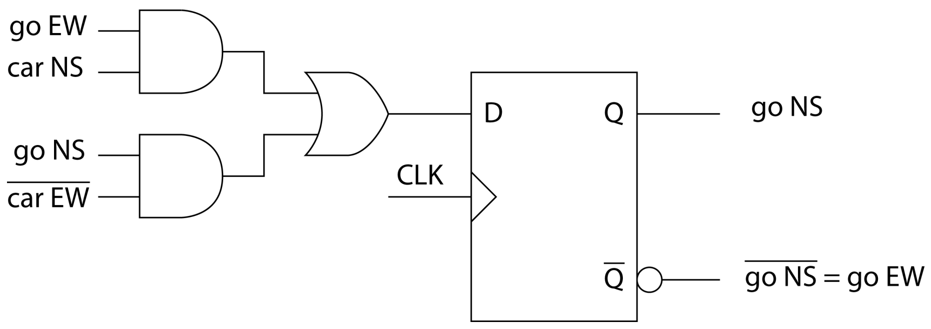


Figure 6.9: Traffic light controller schematics

For the most part, this compendium will stick with the design of synchronous sequential circuits.

If we go back to the traffic light example, it can be implemented as a synchronous Moore machine with D-flip-flops by first deriving the characteristic-/state transition table from the state transition graph. It is given in table 6.9, where the conditions 'car from E or car from W' have been combined to 'car EW'. It has been chosen to represent the two states by a single D-flip-flop. Note, that also the implicit conditions for a state to be maintained have to be included in the table, even if they are not explicitly stated in the graph.

From the characteristic table one can derive the combinational circuit. In more complicated cases on might employ a Karnaugh map to find a simple functional expression first. Here, it is rather straight forward to find the circuit in figure 6.9.

A further systematic derivation will be conducted for counters in section 6.4.1

6.3 Registers

Registers are a concept that will simplify following discussions of more complex logic. They are nothing more fancy than an array of flip-flops that are accessed in parallel (e.g., as memory blocks in a CPU), controlled by

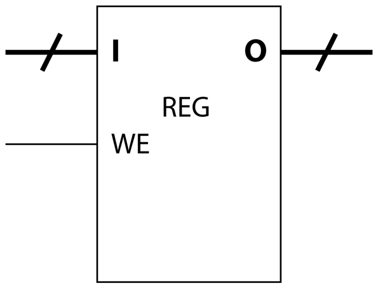


Figure 6.10: Symbol of a simple register

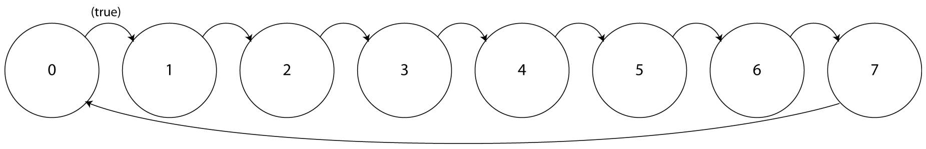


Figure 6.11: State transition graph of a 3-bit counter

present			in	next		
S_2	S_1	S_0	NA	S_2	S_1	S_0
0	0	0		0	0	1
0	0	1		0	1	0
0	1	0		0	1	1
0	1	1		1	0	0
1	0	0		1	0	1
1	0	1		1	1	0
1	1	0		1	1	1
1	1	1		0	0	0

Table 6.10: State transition table of a 3-bit counter

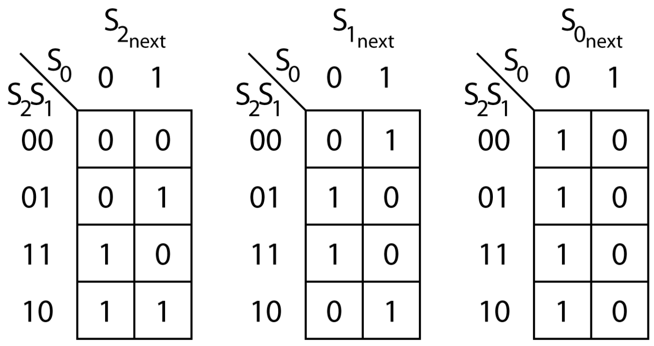
shared control signals. The array is usually of a size that is convenient for parallel access in the context of a CPU/PC, e.g., one Byte or a Word. Possibly most common is the use of an array of D-flip-flops. A typical control signal is a 'write enable' (WE) or synchronous load (LD). In a D-flip-flop based register, this signal is 'and-ed' with the global clock and connected to the D-flip-flop clock input, such that a new input is loaded into the register only if WE is active. Other control signals might be used to control extra functionality (e.g., in shift-registers, see section 6.4).

A simple register symbol is depicted in figure 6.10.

6.4 Standard Sequential Logic Circuits

6.4.1 Counters

Counters are a frequently used building block in digital electronics. A counter increases a binary number with each clock edge. The state transition graph of a 3-bit counter is given in figure 6.11



S_{2next}

S_2S_1 \ S_0	0	1
00	0	0
01	0	1
11	1	0
10	1	1

S_{1next}

S_2S_1 \ S_0	0	1
00	0	1
01	1	0
11	1	0
10	0	1

S_{0next}

S_2S_1 \ S_0	0	1
00	1	0
01	1	0
11	1	0
10	1	0

Figure 6.12: Karnaugh maps for the state transition table of a synchronous 3-bit counter

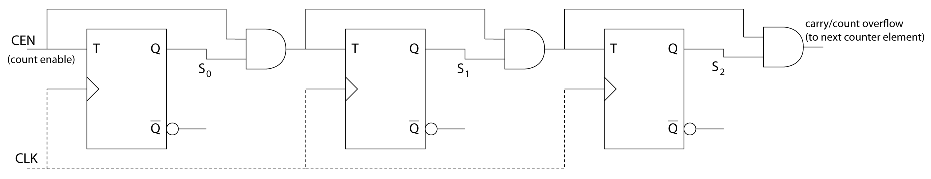


Figure 6.13: 3-bit synchronous counter

As indicated, let us represent the 8 states in D-flip-flops with the corresponding binary numbers. Thus, the state is the output and there is in a first instance no output combinational logic necessary. (Note that output combinational logic will be necessary if the numbers should, for example, appear on an LED-display.) We can thus further define the FSM by its *state transition table* (table 6.10), a table that shows the relation between present state and present input, and the next state. Actualy it is also just a characteristic table for a more complex circuit, and sometimes this text will also refer to it as such. Note that our simple counter does actually not have any inputs and that the state transitions are all unconditional, i.e., they occur with each clock cycle and do not depend on any inputs.

Since the state transition table defines an input/output relationship, where the input is the present state plus the inputs (not available in this example) and the output the next state, we can deduce the simplified combinational logic that will connect the output of the D-flip-flops (present state) to the input of the D-flip-flops (next state) with the help of Karnaugh maps (figure 6.12), as was introduced in section 4.1.

Equation (6.7) is the resulting sum of minterms for bit S_2 and can be further simplified to equation 6.9.

$$\begin{aligned}
S_{2_{next}} &= (S_2 \cdot \overline{S_1}) + (S_2 \cdot \overline{S_0}) + (\overline{S_2} \cdot S_1 \cdot S_0) && (6.6)\\
&= S_2 \cdot (\overline{S_1} + \overline{S_0}) + (\overline{S_2} \cdot (S_1 \cdot S_0)) && (6.7)\\
&= S_2 \cdot \overline{(S_1 \cdot S_0)} + \overline{S_2} \cdot (S_1 \cdot S_0) && (6.8)\\
&= S_2 \oplus (S_1 \cdot S_0) && (6.9)
\end{aligned}$$

Contemplating this equation somewhat we can deduce a general expression for the nth bit of a counter ($\wedge$=AND):

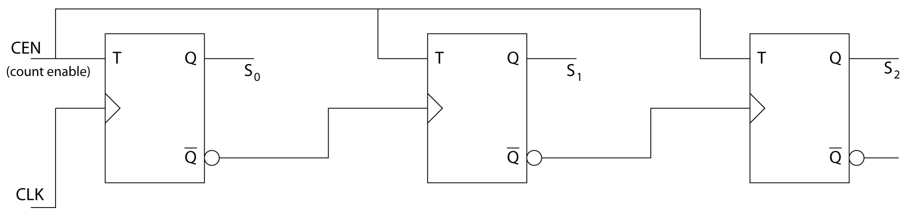


Figure 6.14: 3-bit ripple counter

control			next		
LD	SE	LS	O_2	O_1	O_0
1	X	X	I_2	I_1	I_0
0	0	X	O_2	O_1	O_0
0	1	0	RSin	O_2	O_1
0	1	1	O_1	O_0	LSin

Table 6.11: Shift register state transition table

$$S_{n_{\text{next}}} = S_n \oplus \left(\bigwedge_{k=0}^{n-1} S_k \right) \tag{6.10}$$

And if we consult the characteristic equation of a T-flip-flop (6.4) we note that it is an XOR of its present state and its T-input, so we may conveniently use it to implement the counter, by connecting the product term in (6.10) to the input T and the resulting circuit implementing a *synchronous 3-bit counter* is shown in figure 6.13. One more control signal has been added, the 'counter enable' (CE) which can be used to stop the counting entirely.

Counters may also be equipped with even more control signals to control extra functionality such as:

- Possibility for loading an initial number (control signal LD and an input bus)
- Reset to zero (control signal RES)
- Switching between up and down counting (control signal UpDown)

A simple and popular asynchronous variant (only the first T-flip-flop is clocked with the global clock) of a counter is the *ripple counter* shown in figure 6.14. A possible disadvantage is that the output signal 'ripples' from the lowest to the highest bit, i.e., the highest bits are updated with a delay. This must be taken into account, if this kind of counter is used.

6.4.2 Shift Registers

Shift registers are registers that can shift the bits by one position per clock cycle. The last bit 'falls out' of the register when shifting. The first bit that is 'shifted in' can for example:

- be set to zero

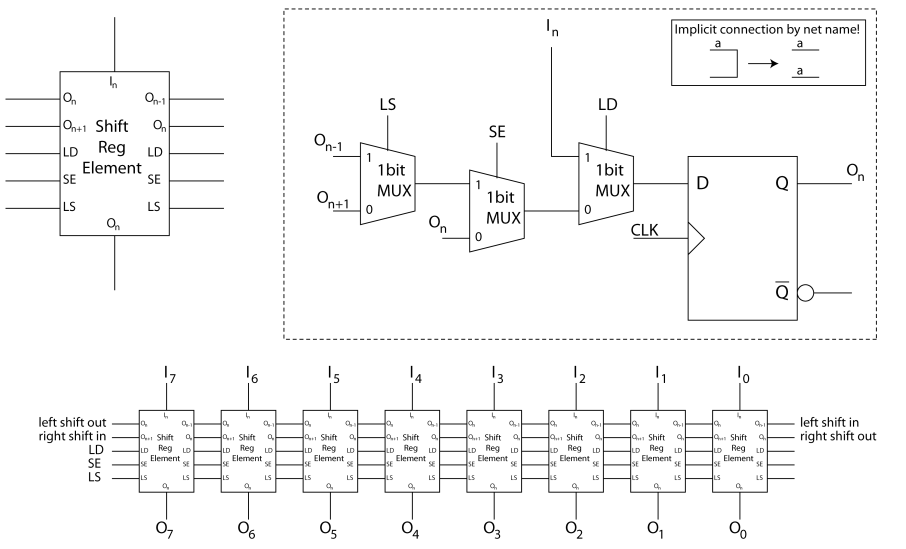


Figure 6.15: Shift register

- be connected to the last bit (cycling/ring counter)
- be connected to a serial input for serial loading

Typical control signals are:

- load (LD, for parallel loading)
- shift enable (SE, to enable or stop the shifting)
- left shift (LS, for controlling the direction of the shift

Note that a left shift of a binary number corresponds to a multiplication with 2, an arithmetic right shift (shift in the former MSB from the left) to a division with 2 (rounded towards $-\infty$!, see also section 3.4)

Shift registers find their use, for example in:

- Parallel to serial and serial to parallel conversion
- Binary multiplication
- Ring 'one-hot' code counters/scanners
- Pseudo random number generators

Chapter 7

Von Neumann Architecture

In 1945 John von Neumann published his reference model of a computer architecture that is still the basis of most computer architectures today. The main novelty was that a single memory was used for both, program *and* data.

Figure 7.1 shows an outline of the architecture, composed of 5 main units:

- control unit
- execution unit
- memory unit
- input unit
- output unit

The control unit (CU) is the central finite state machine that is controlled by input from the memory, i.e., the program. Its internal states are its registers. Typical registers of the CU are:

PC: (program counter, also called instruction pointer (IP)) the register holding the memory address of the next machine code instruction.

IR: (instruction register) the register holding the machine code of the instruction that is executed.

MAR: (memory address register) half of the registers dedicated as interface of the memory with the CPU, holding the memory address to be read or written to.

MBR: (memory buffer register) the other half of the CPU-memory interface, a buffer holding the data just read from the memory or to be written to the memory. Typically the MBR can be connected as one of the inputs to the ALU.

The MAR and the MBR may also be assigned to a subunit of the CU, the memory controller, which even has been implemented on a separate IC and been placed outside the CPU to allow for more system flexibility, i.e., to allow to reuse the same CPU with different generations and/or sizes of

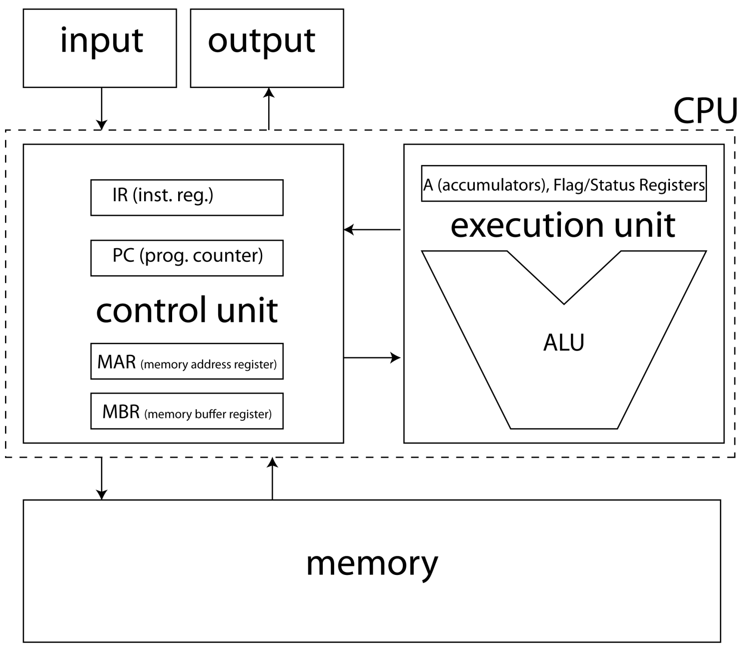

input
output
CPU
IR (inst. reg.)
PC (prog. counter)
control unit
MAR (memory address register)
MBR (memory buffer register)
A (accumulators), Flag/Status Registers
execution unit
ALU
memory

Figure 7.1: Von Neumann architecture

memory. For the sake of speed, however, many modern CPUs do again physically contain a memory controller.

The execution unit is the work horse of the CPU that manipulates data. In its simplest form it would merely be combinational logic with input and output registers, performing all of its tasks within a single clock cycle. A core component is the arithmetic and logic unit (ALU) which is instructed by the CU to execute arithmetic operations such as addition, subtraction, multiplication, arithmetic shifts and logical operations such as bit-wise and, or, xor and logic shifts etc. A simple ALU is a combinational circuit and is further discussed in section 7.2. More advanced execution unit will usually contain several ALUs, e.g., at least one ALU for pointer manipulations and one for data manipulation. Registers of the CPU that usually are assigned to the execution unit are:

accumulator: a dedicated register that stores one operand and the result of the ALU. Several accumulators (or general purpose registers in the CPU) allow for storing of intermediate results, avoiding (costly) memory accesses.

flag/status register: a register where each single bit stands for a specific property of the result from (or the input to) the ALU, like carry in/out, equal to zero, even/uneven, ...

The memory, if looked upon as a FSM has a really huge number of internal states, so to describe it with a complete state transition graph would be quite a hassle, which we will not attempt. Its workings will be further explained in section 7.3

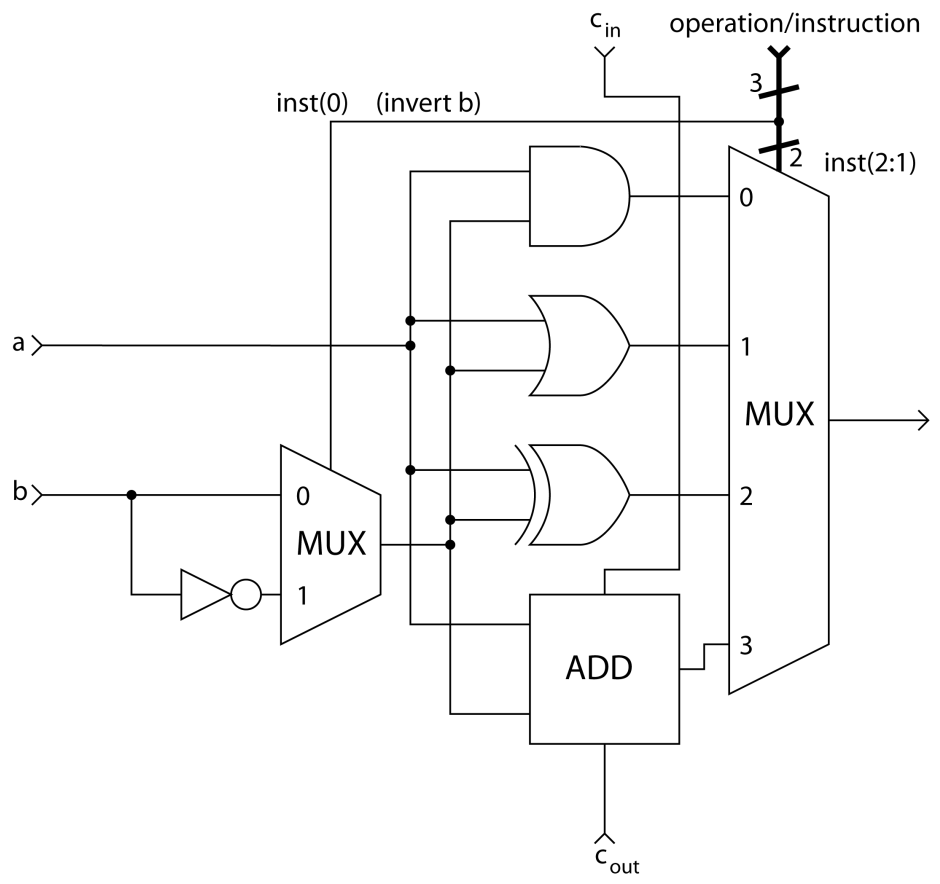


Figure 7.2: Example schematics of a 1-bit arithmetic logic unit (ALU)

7.1 Data Path and Memory Bus

The *data path* refers to the internal bus structure of the CPU. Buses are connections between registers, the functional units (such as the ALU), the memory, and I/O units. They are often shared by more than two of those units and usually only one unit sends data on the bus to one other at a time. Internally in the CPU there are usually several buses for data exchange among the registers and functional units, allowing parallel access to several registers at the same time, i.e., within one clock cycle. However, there is only one bus between the memory and the CPU for both instruction and data transfer in a von Neumann architecture (actually two: one for the address and one for the data), this bus can be a main speed limiting factor which is known as the *von Neumann bottleneck*.

7.2 Arithmetic and Logic Unit (ALU)

Figure 7.2 shows a possible implementation of a basic ALU as a combinational logic circuit. The instruction code or operation code (inst/opcode) determines which function the ALU applies to its input as detailed in table 7.1. Note that the subtraction will not just require that the b is inverted, but also that the carry in signal be set to 1 (in an n-bit ALU only for the LSB). Remember that in two's complement, a binary number is inverted by inverting each bit *and* adding 1. The addition of 1 is achieved by using the

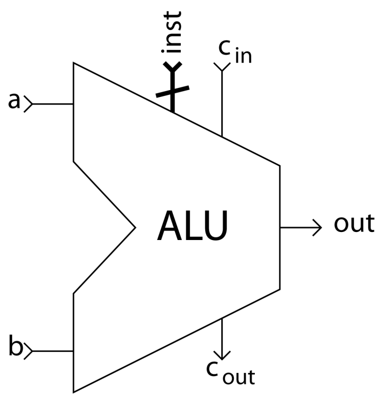


Figure 7.3: A possible symbol used for an 1-bit ALU

inst	computation
000	$a \cdot b$
001	$a \cdot \overline{b}$
010	$a + b$
011	$a + \overline{b}$
100	$a \oplus b$
101	$a \oplus \overline{b}$
110	$a + b$
111	$a - b^{(!)}$

Table 7.1: 1-bit ALU truth table

carry in signal. This is explicitly shown in figure 7.4 that combines n 1-bit ALUs. The symbol for a 1-bit ALU that is used is shown separately in figure 7.3, and a possible symbol for an n-bit ALU in figure 7.5.

More complicated ALUs will have more functions as well as flags, e.g., overflow, divide by zero, etc.

Modern CPUs contain several ALUs, e.g., one dedicated to memory pointer operations and one for data operations. ALUs can be much more complex and perform many more functions in a single step than the example shown here, but note that even a simple ALU can compute complex operations in several steps, controlled by the software. Thus, there is always a trade-off of where to put the complexity: either in the hardware or in the software. Complex hardware can be expensive in power consumption, chip area and cost. Furthermore, the most complex operation may determine the maximal clock speed. The design of the ALU is a major factor in determining the CPU performance!

7.3 Memory

The memory stores program and data in a computer. In the basic von Neumann model the memory is a monolithic structure, although in real computer designs, there is a memory hierarchy of memories of different types and sizes, but we will come back to this later (section 8.1). The

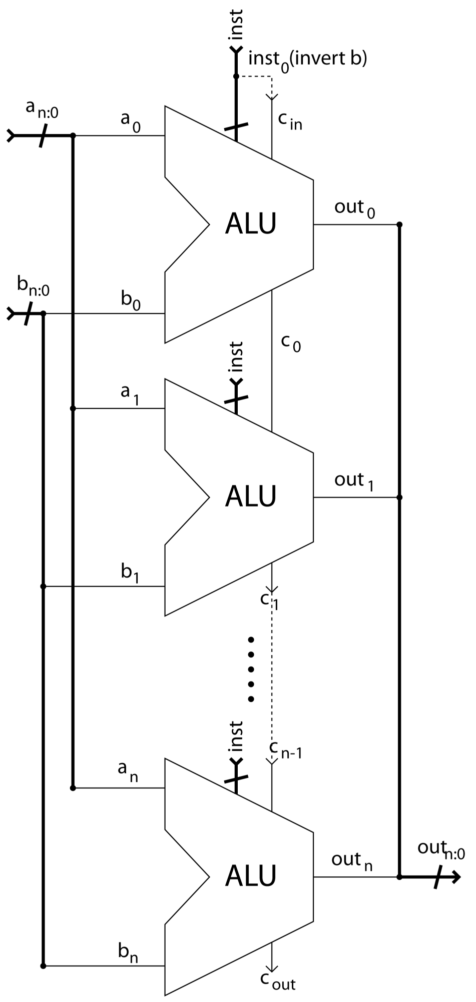


Figure 7.4: Example of an n-bit ALU schematics

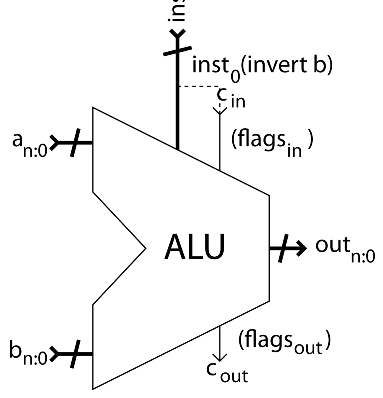


Figure 7.5: n-bit ALU symbol

inputs					state	out
I	A	$\overline{CS}$	$\overline{WE}$	$\overline{OE}$	M(A)	O
X	X	1	X	X	M(A)	Z
X	X	0	1	1	M(A)	Z
I	A	0	0	1	I	Z
I	A	0	1	0	M(A)	M(A)
I	A	0	0	0	I	M(A)

Table 7.2: Characteristic table of a synchronous RAM with separate input and output port. Note that there are many variants of RAM with different characteristics. This table is but an example of one variant. Also note that the effect of $\overline{CS}$ and $\overline{OE}$ is usually asynchronous, i.e., immediate, while the effect of $\overline{WE}$ is synchronous, i.e., triggered by the next clock edge.

basic type of memory that is usually employed within a basic von Neumann architecture is *random access memory* (RAM). A RAM has an address port, and a data input and an output port, where the later two might be combined in a single port. If the address port is kept constant, the circuit behaves like a single register that can store one *word* of data, i.e., as many bits as the input and output ports are wide. Changing the address, one addresses a different word/register. 'Random access' refers to this addressed access that allows to read and write any of the words at any time, as opposed to the way that, for instance, pixels on a computer screen are accessible only in sequence.

Some terminology:

address space: The number of words in a RAM. In figure 7.6 it is equivalent to the number of rows.

word length: The number of bits or bytes that can be accessed in a single read/write operation, i.e., the number of bits addressed with a single address. In figure 7.6 the number of columns.

memory size: The word length multiplied with address space.

Note that the x86 architecture (and other modern CPUs) allows instructions to address individual bytes in the main memory, despite the fact that the word length of the underlying RAM is actually 32 bits/4 bytes or 64 bits/8 bytes. Thus, the address space that a programmer sees is in fact bigger than the address space of the RAM.

Typical signals of a RAM:

A((n-1):0): The address port indicating which word should be accessed. 2^n is the address space.

I/D((m-1):0) and O((m-1):0) or D((m-1):0): I (sometimes also referred to as D) and O are the input and output port respectively. Sometimes a single port D is used for both, in which case a control signal $\overline{OE}$ is needed to distinguish between the use of port D as input or output. m is the memory word length in bits.

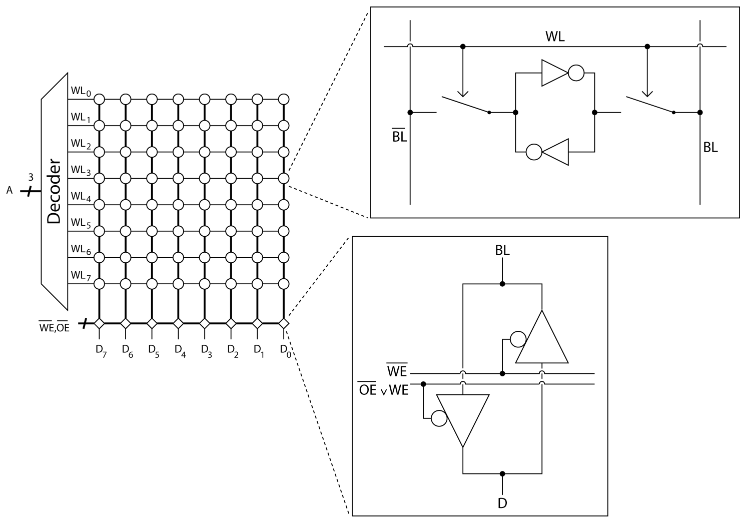


Figure 7.6: Static random access memory principle

$\overline{\text{WE}}$, write enable (often active low): This signal causes a write access writing I/D at address A, either immediately (asynchronous write), with a following strobe signal (see $\overline{\text{RAS}}/\overline{\text{CAS}}$) or with the next clock edge (synchronous write).

$\overline{\text{RAS}}/\overline{\text{CAS}}$, row/column access strobe: appear usually in DRAM (see section 7.3.2) that actually has a 3-D structure: one decoder for the row address, one for the column address and the word (conceptually) extends into the third dimension. The address bus is reused for both row and column address. First the row address is asserted on the address bus A and $\overline{\text{RAS}}$ is pulsed low, then the column address is asserted on the address bus and $\overline{\text{CAS}}$ is pulsed low. $\overline{\text{CAS}}$ is the final signal that triggers either the read (latching of the word into a read buffer)) or write operation. The other control signals are asserted *before*. Several column accesses can be made for a single row access for faster access times.

$\overline{\text{OE}}$, output enable: A signal that lets the RAM drive the data line while asserted, but lets an external source drive the data lines while deasserted. This can be used to regulate the access if there are several devices connected to a single bus: Only one of them should be allowed to drive the bus at anyone time. Deasserted, it can also allow a common data line to be used as input port.

$\overline{\text{CS}}$, chip select: A control line that allows to use several RAMs instead of just one on the same address and data bus and sharing all other control signals. If $\overline{\text{CS}}$ is not asserted all other signals are ignored and the output is not enabled. Extra address bits are used to address one specific RAM and a decoder issues the appropriate $\overline{\text{CS}}$ to just one RAM at a time. This extends the address space.

RAM comes in either of two main categories: static- and dynamic random access memory (SRAM and DRAM).

7.3.1 Static Random Access Memory (SRAM)

Figure 7.6 shows an example block diagram of a static RAM with asynchronous $\overline{WE}$. Its basic 1-bit storage element are flip-flops as depicted in the upper right corner of the figure, quite similar to the basic D-latch introduced in section 6.1.1. An active high signal on the word line (WL) will connect that latches content to the bit lines (BL and $\overline{BL}$). The bit-lines connect the same bit in all words, but only ever one word line is active at anyone time. This is ensured by the decoder (lefthand side of the figure) that decodes the address A. Thus, a single word at a time can either be read or written to.

The figure introduces yet another digital building block within the lower right corner circuit, the *tri-state buffer*. Tri-state buffers allow the outputs of different logic circuits to be connected to the same electrical node, e.g., a bus-line. usually, only one of these outputs at a time will be allowed to drive that line and determine its states, while all others are in principle disconnected from that line. So, the tri-state output can actually have three different states, as controlled by its control input and the input: while its control input is active, it conveys the usual ‘high’/1 and ‘low’/0 from input to output. If the control input is inactive, the output is set to a ‘high impedance’ output denoted with ‘Z’. To say it more plainly, in this third state, the buffer acts like a switch that disconnects its input from the output.

The write access is somewhat less elegant than in a D-latch (section 6.1.1): in the D-latch, the feedback loop that maintains the state is disconnected during a write access. Here, however, the feedback loop is maintained, also during a write access. An active low $\overline{WE}$ activates a tri-state buffer that drives the bit-lines. This buffer needs to be stronger than the feedback loop in order to overrule it. This way, one saves an extra switch in the storage cell making it more compact, and compactness is the main criterion for memory cells, since the main goal is to get as much memory in as little a space as possible. The price one pays is a considerable power consumption while the feedback loop ‘struggles’ against the write input and a degradation of the writing speed. Note that in this figure only the non-inverted bit-line BL is driven during a write access, which is a possible design. Usually, also the inverted bit-line $\overline{BL}$is actively driven during a write operation to increase the writing speed.

7.3.2 Dynamic Random Access Memory (DRAM)

While a SRAM needs 6 transistors per storage cell (2 per inverter and 1 per switch), a DRAM merely needs two capacitors and two transistors. Instead of an active feedback loop to maintain a state, the DRAM relies on charge stored on a capacitor. Capacitive storage, however, is not self maintaining. Charge on a capacitor is leaking and lost over time. Thus, a *sense amplifier* has to be connected to every memory cell within a given time period, while the memory is idle. In modern DRAM internal state machines take care of this refresh cycle.

The sense amplifier reconstructs the digital state from the analog state that the memory cell has decayed to. In principle, it is nothing but a flip-flop itself, and if a flip flop should ever find itself in a state where its content is neither ‘1’ nor ‘0’, it will quickly recover to the closer of those

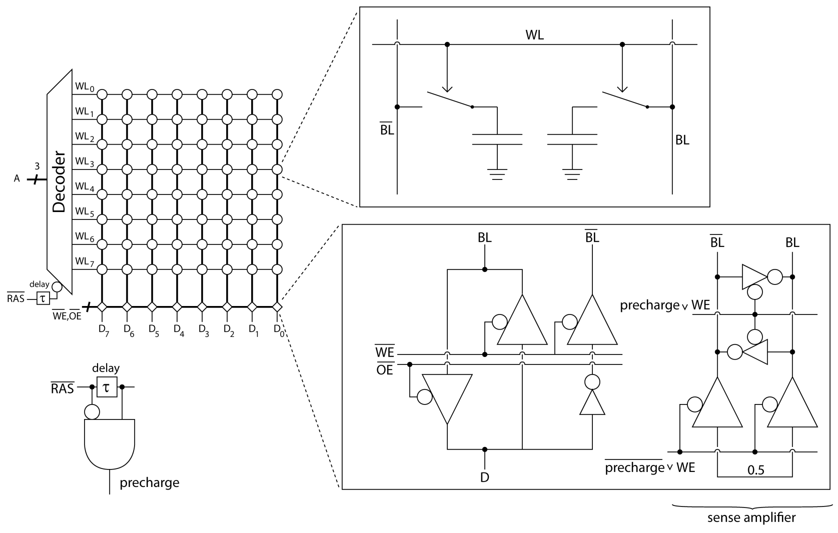


Figure 7.7: Dynamic random access memory (DRAM) principle

two states, due to the amplification in the feedback loop. In the lower right corner of figure 7.7, such a sense amplifier is depicted: it's a pair of tri-stated inverters connected in a feedback loop. To refresh a cell, first the differential bit-lines BL and $\overline{\text{BL}}$ are *precharged* to a state between '0' and '1', i.e., '0.5', then the memory cell that is to be refreshed is connected to those lines, passively pulling them in the direction of either (1,0) or (0,1) dependent on the charge that remains on the two memory cell capacitors. Then the sense-amplifier is turned on, pulling them further apart actively in the same direction, until the digital state is reached again.

The rest of the circuit in figure 7.7 is quite similar to the SRAM in figure 7.6. One difference is that here you need to drive both BL and $\overline{\text{BL}}$ during a write operation. Another difference is the introduction of a write strobe signal pulse $\overline{\text{RAS}}$, so it is not $\overline{\text{WE}}$ that directly triggers the writing of the memory cell but this strobe signal, this pulse is also used to generate a shorter precharge pulse during which the bit-lines are precharged, and after which, the decoder output is enabled, connecting the memory cell to the precharged bit-lines. In the case of a read access the sense amplifier is also turned on and the memory cell content is, thus, simultaneously read and refreshed. The sense amplifier will retain the memory cell state after the cell itself is disconnected until the next $\overline{\text{RAS}}$ strobe.

DRAMs can be more compact than SRAMs to the lesser number of transistors per memory cell, and thus one can produce bigger memory sizes more cheaply. One problem that arises with this is that the number of address lines that are needed to address the entire address space becomes bigger and bigger. To avoid enormously wide address buses, the address is split into two parts, denoted as row and column address. We will not delve deeply int this here and now. To tell a somewhat simplified story that still conveys the principle operation: the row address is loaded into a internal register within the DRAM first. The $\overline{\text{RAS}}$ strobe will trigger that latching. Then the column address is placed on the address bus and a separate column address strobe $\overline{\text{CAS}}$ triggers the actual decoder output and the latching of the memory content into sense amplifier. Repeated fast

	SRAM	DRAM
access speed	+	-
memory density	-	+
no refresh needed	+	-
simple internal control	+	-
price per bit	-	+

Table 7.3: Comparison of SRAM and DRAM

expression	meaning
X	register X or unit X
[X]	the content of X
$\leftarrow$	replace/insert or execute code
M()	memory M
[M([X])]	memory content at address [X]

Table 7.4: RTL grammar

memory access with the same row address and only changing the column address is another consequence of this address splitting.

In general, however the major drawback of DRAM is access speed and the need for active refreshing which requires some administrative circuitry overhead. A summary of the advantages and drawbacks of the two variants is given in table 7.3.

7.4 Control Unit (CU)

7.4.1 Register Transfer Language

To describe the FSM that is the control unit one may employ the socalled 'register transfer language' (RTL), since moving data among registers is a central par of what the CU does, besides telling the execution unit to manipulate some of these date. The syntax of RTL is illuminated in table 7.4.

An example: [IR] $\leftarrow$ [MBR] transfer the content of the MBR to the IR

7.4.2 Execution of Instructions

Let us look at a simple example CPU that is executing machine code. At start-up of the CPU the program counter is initialized to a specific memory address. Here to address 0. The memory content is as shown in table 7.5.

At the beginning of a instruction cycle a new instruction is fetched from the memory. A finite state machine in the control unit generates the right sequence of control signals. Actually the CU *is* nothing but a finite state machine controlled by the instructions. The following RTL code describes what is happening:

mem adr	content
0	move 4
1	add 5
2	store 6
3	stop
4	1
5	2
6	0
⋮	⋮

Table 7.5

[MAR] ← [PC]
[PC] ← [PC] + 1
[MBR] ← [M([MAR])]
[IR] ← [MBR]

As a last stage of the fetch phase the operation code ('move') of the instruction is decoded by the control unit (CU):

$$\text{CU} \leftarrow [\text{IR}(\text{opcode})]$$

and triggers a cycle of the finite state machine with the appropriate signals to execute a sequence of operations specific to the instruction. The order, type and number of the individual operations may vary among different instructions and the set of instructions is specific to a particular CPU.

The other part of the first *machine code* of the first instruction in our (16-bit) processor is the 'operand' 4. What we have written as 'move 4' is actually a bit pattern:

$$\underbrace{10110010}_{\text{opcode: move}} \quad \underbrace{00000100}_{\text{operand: 4}}$$

As mentioned before, the set of instructions and the machine codes are specific to a CPU. Machine code is *not portable* between different CPUs.

After the opcode has been decoded into appropriate control signals it is executed. In this first instruction the data from memory location 4 (1) is moved to the accumulator A (often the accumulator is the implicit target of instructions without being explicitly defined):

[MAR] ← [IR($operand$)]
[MBR] ← [M([MAR])]
[A] ← [MBR]

Thus, the first instruction is completed and a new instruction fetch is initiated, now from the next memory location but otherwise exactly like before:

mem adr	content
0	move 4
1	add 5
2	store 6
3	stop
4	1
5	2
6	3
⋮	⋮

Table 7.6

[MAR] ← [PC] (now [PC]=1)
[PC] ← [PC] + 1
[MBR] ← M([MAR])
[IR] ← [MBR]

The instruction in the IR is now 'add 5'.

CU ← [IR($opcode$)]

5 denotes the memory location of the variable that is to be added to the content of the accumulator A:

[MAR] ← [IR($operand$)]
[MBR] ← [M([MAR])]

The ALU receives the appropriate instruction from the state machine triggered by the opcode (sometimes parts of the opcode simply *are* the instruction for the ALU, avoiding a complicated decoding).

ALU ← [A]; ALU ← [MBR]
[A] ← ALU

The number from memory location 5 (2) has been added and the result (3) is stored in the accumulator. the second instruction is complete.

It follows another instruction fetch and decode like before (not shown).

⋮

and then the execution of the third instruction which causes a write access to the memory:

[MBR] ← [A]
[MAR] ← [IR($operand$)]
[M([MAR])] ← [MBR]

Then, the forth instruction is a stop instruction which halts the execution of the program. The memory content is now changed as shown in figure 7.6.

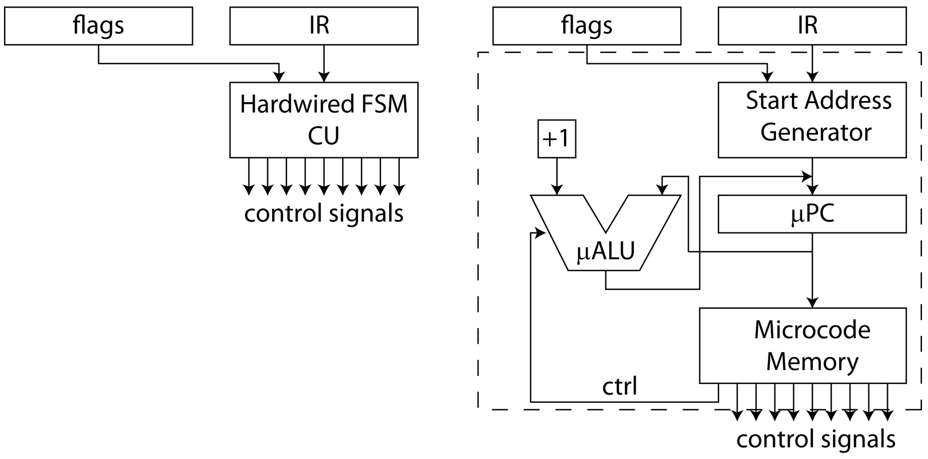


Figure 7.8: Hardwired and Microprogrammed CU

	Microarchitecture	Hardwired
Occurrence	CISC	RISC
Flexibility	+	-
Design Cycle	+	-
Speed	-	+
Compactness	-	+
Power	-	+

Table 7.7: Pros and Cons of hardwired and microarchitecture CU

7.4.3 Microarchitecture

If the CU were implemented like the example FSMs in section 6.2 the result would be a socalled *hardwired* CU architecture, where a hardwired FSM issues the right sequence of control signals in response to a machine code in the IR.

A more flexible alternative is to use *microcode* and a simple 'processor within the processor' that simply issues a sequence of control signals stored as *micro-instructions*. (Note that the term has later been used (and will be used in this compendium, section 8.2) differently in modern pipelined CPUs) in the *microprogram memory*, typically a fast read only memory (ROM) but sometimes also a flash memory (i.e., electrically erasable programmable read only memory (EEPROM)). The two concepts are illustrated in figure 7.8. In the microarchitecture, a normal instruction is decoded to a start address of the microinstruction memory and the micro program at that location is executed. Normally these micro programs are really limited to issuing a sequence of control signals. Sometimes, unconditional jumps are also implemented (thus, the 'ctrl' connection to the micro ALU) that allows the execution of micro-subprograms to save space in the microcode memory if certain sequences of microcode occur in several instructions.

Table 7.7 summarizes the advantages and disadvantages of the two approaches. Obviously, microcode is a very convenient way to implement the generation of sequences of control signals, especially in CPUs with a rather complex instruction set of instructions of variable length (complex instruction set computers, CISC, as opposed to reduced instruction set computers, RISC. see section 7.4.4). The micro memory might be a 'program once' read only memory (ROM) or even a flash ROM that can be programmed repeatedly. Thus, a flexible redesign of the CU can be achieved by programming instead of a full hardware redesign, considerably reducing the time needed for a design cycle. The drawbacks are the general drawbacks often found when comparing a more flexible design to a highly specialized one: a highly specialized hardwired implementation will be more efficient in power and speed.

7.4.4 Complex and reduced instruction sets (CISC/RISC)

The distinction in this section's title is somewhat historical only. An initial CPU design tendency had been to increase the number and complexity of instructions, to allow shorter and more efficient programs, i.e., to move the complexity from software to hardware. This led to socalled *complex instruction set computers* (CISC). At some point, CPU designers realized that the implementation of a smaller and simpler instruction set could be much more optimized. In particular the idea of instruction pipelines (see section 8.2) that enormously accelerated execution of instructions, was first achieved in *reduced instruction set computers*. However, this proved only a temporary setback for complex instructions as the tendency to ever bigger instruction sets, set in again at once. The pipelined architectures these days easily rival the old CISC architectures in number and complexity of instructions. They use a trick internally of decoding complex instructions into several simple *micro-instructions* (section 8.2). (Note that the meaning of this word is different than in the context of micro-architectures introduced earlier!)

7.5 Input/Output

A computer is connected to various devices transferring data to and from the main memory. This is referred to as Input/output (I/O). Examples: Keyboard, Graphics, Mouse, Network (Ethernet, Bluetooth, ...), USB, Firewire, PCI, PCI-express, SATA, ...

Early computer architectures had a monolithic system interface, a combined I/O and memory bus, like depicted in figure 7.9. Today the system interface is more sectioned (e.g., by the north- and south-bridge in the Intel chip-sets) to grant faster CPU access to privileged system components, i.e., the main memory and the graphics card, but for this discussion, we will keep to the simpler monolithic view. Thus, the CPU interfaces to I/O devices in a similar way than to its main memory.

I/O controllers (depicted in figure 7.10) translate and synchronize peripheral device *protocols* (communication languages) to the protocol of the system bus. They normally have at least one data buffer referred to as *I/O port*, a control register that allows some SW configuration and a status register with information for the CPU.

I/O addressing from the CPUs point of view usually follows one of two principles:

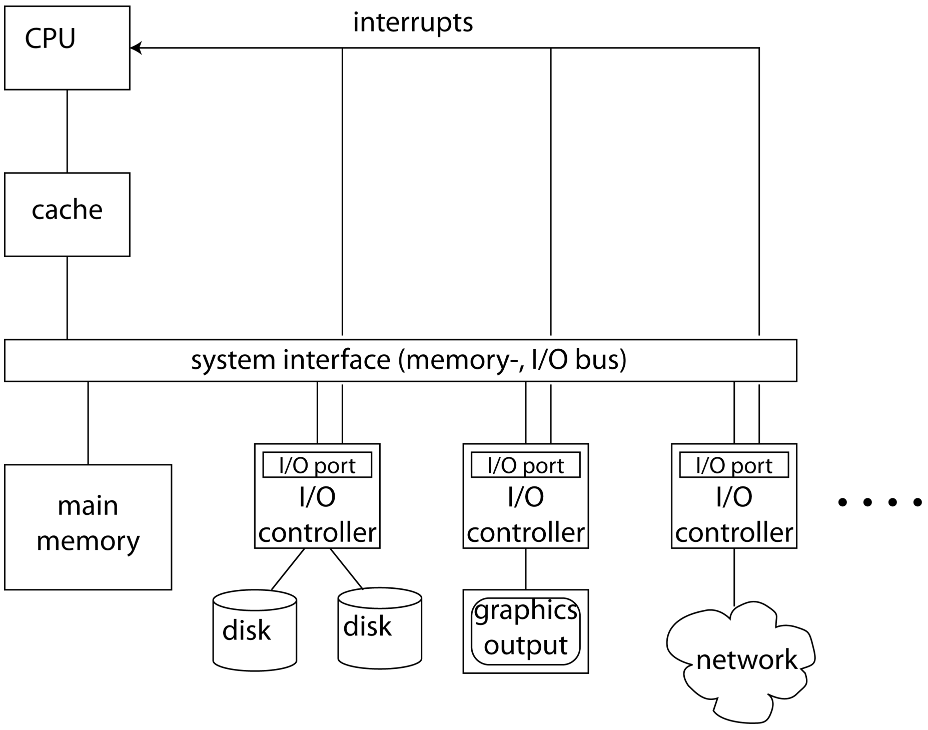


Figure 7.9: Simple I/O block diagram

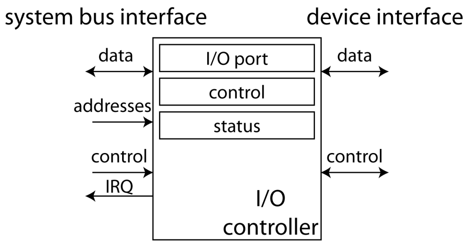


Figure 7.10: I/O controller principle

Memory mapped I/O is to access I/O ports and I/O control- and status registers (each with its own address) with the same functions as the memory. Thus, in older systems, the system interface might simply have been a single shared I/O and memory bus. A disadvantage is that the use of memory addresses may interfere with the expansion of the main memory.

Isolated I/O (as in the 80x86 family) means that separate instructions accessing an I/O specific address space are used for I/O. An advantage can be that these functions can be made *privileged*, i.e., only available in certain modes of operation, e.g., only to the operating system.

Modes of Transfer:

Programmed/Polled: The processor is in full control of all aspects of the transfer. It *polls* the I/O status register in a loop to check if the controller has data to be collected from the port or is ready to receive data to the port. Polling uses up some CPU time and prevents the CPU from being used for other purposes while waiting for I/O.

Interrupt Driven: The I/O controller signals with a dedicated 1bit data line (*interrupt request* (IRQ)) to the CPU that it needs servicing. The CPU is free to run other processes while waiting for I/O. If the interrupt is not *masked* in the corresponding CPU status register, the current instruction cycle is completed, the processor status is saved (PC and flags pushed onto stack) and the PC is loaded with the starting address of an *interrupt handler*. The start address is found, either at a fixed memory location specific to the interrupt priority (*autovectored*) or stored in the controller and received by the CPU after having sent an *interrupt acknowledge* control signal to the device (*vectored*)

Direct Memory Access (DMA): The processor is not involved, but the transfer is negotiated directly with the memory, avoiding copying to CPU registers first and the subroutine call to the interrupt handler. DMA is used for maximum speed usually by devices that write whole blocks of data to memory (e.g., disk controllers). The CPU often requests the transfer but then relinquishes control of the system bus to the I/O controller, which only at the completion of the block transfer notifies the CPU with an interrupt.

(DMA poses another challenge to the cache as data can now become *stale*, i.e., invalid in the cache)

Chapter 8

Optimizing Hardware Performance

8.1 Memory Hierarchy

We have discussed two major types of memory, SRAM and DRAM, and stated that the former was faster but less dense and more expensive, whereas the latter was slower but denser and cheaper. Which was one to choose to design a computer? Computer designs try to optimize speed, which would speak for SRAM, but also require as much storage space as possible, which favours DRAM. Therefore, computers actually use *both*: a small fast memory *cache* composed of SRAM for data that is accessed often and a large DRAM for the *main memory* that is used for longer term storage of data that is not accessed quite so often. Cache, in fact, is subdivided in several levels, where L1 cache is smallest and fastest. Note that there is a trade off between size and speed, since larger memories require more extensive cabling/routing of signals which limits access time due to parasitic capacitance.

The challenge now is, how to know which data to keep in the fastest memory. In fact, the fastest type of memory are the CPU internal registers, and even slower but more massive than the DRAM main memory is a computers hard-drive. Thus, one speaks of a *memory hierarchy* as depicted in figure 8.1 and the following text describes the mechanisms and architecture that are employed to assign different data to these different storage media.

Table 8.1 gives an indication on access time and size of the different types of memory. Note that SSD/flash memory is competing to replace hard-drives, but is at present still more expensive, not quite so big and has a shorter lifetime, but these shortcomings are slowly being overcome as this text is written.

8.1.1 Cache

Cache is used to ameliorate the von Neumann memory access bottleneck. Cache refers to a small high speed RAM integrated into the CPU or close to the CPU. Access time to cache memory is considerably faster than to the main memory. Cache is small to reduce cost, but also because there is always a trade off between access speed and memory size. Thus,

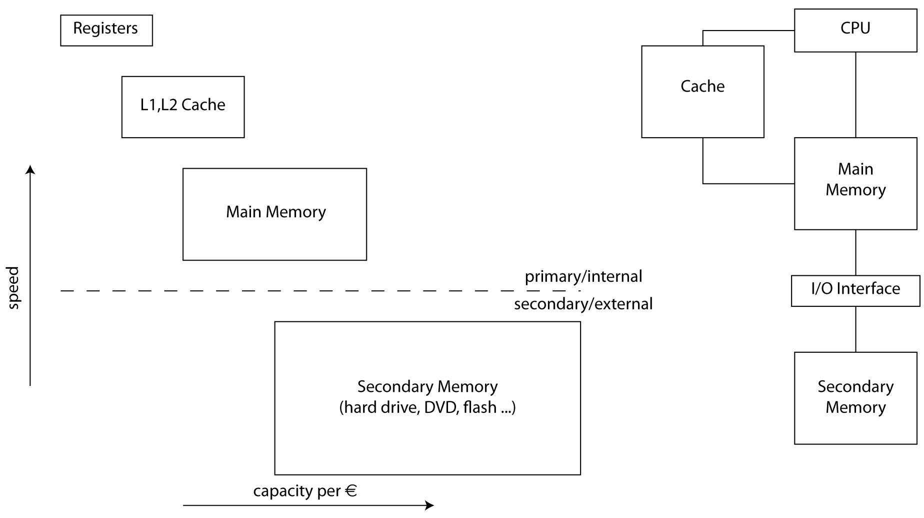


Figure 8.1: Memory hierarchy

registers	~ 1ns	~ 100B
L1 (on CPU) cache	~≥ 1ns	~ 10kB
L2,L3 (off CPU) cache	~ 2ns-10ns	~ 1MB
main memory (DRAM)	~ 20ns-100ns	~ 1GB
SSD/flash	~ 100ns-1μs	~ 10-100GB
hard disc	~ 1ms	~ 0.1-1TB

Table 8.1: Memory hierarchy summary table

modern architectures include also several hierarchical levels of cache (L1, L2, L3, ...).

Cache uses the principle of *locality* of code and data of a program, i.e., that code/data that is used close in time is often also close in space (memory address). Thus, instead of only fetching a single word from the main memory, a whole block around that single word is fetched and stored in the cache. Any subsequent load or write instructions that fall within that block (a cache *hit*) will not access the main memory but only the cache. If an access is attempted to a word that is not yet in the cache (a cache *miss*) a new block is fetched into the cache (paying a penalty of longer access time).

8.1.1.1 Cache mapping strategies

Checking for hits or misses quickly is a prerequisite for the usefulness of cache memory. Thus, memory addresses are mapped onto cache addresses in particular ways to simplify this check:

associative cache: Memory blocks can be stored in any position in the cache. A tag at the beginning of a block in the cache identifies the block in the memory. Fast parallel search on these tags is implemented using extremely specialized HW. Figure 8.2 shows the concept. The drawn out pointers indicate which memory blocks that are at present stored where in the cache.

direct mapped cache: A hash-function assigns each memory block to only one cache slot (e.g., by using only the LSBs of the memory address, but

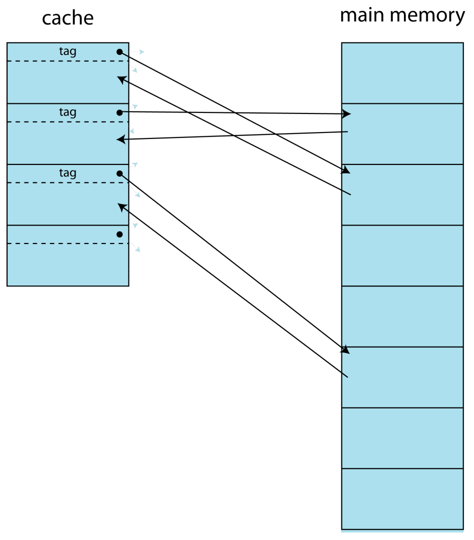


Figure 8.2: Associative cache

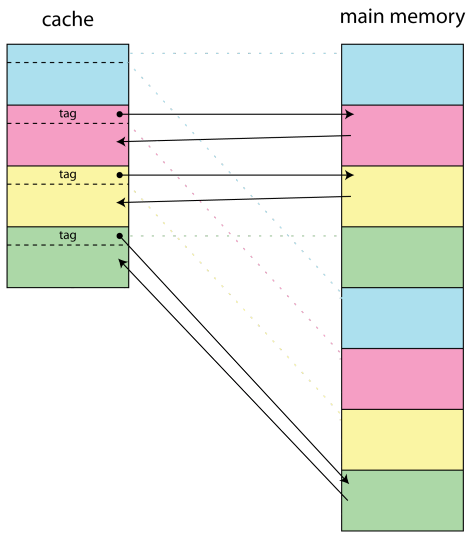


Figure 8.3: Directly mapped cache

other functions can be employed too). Checking for a hit is extremely simple: only one tag needs to be checked. However, if a program uses data blocks that are hashed to the same cache address, the efficacy of the caching is significantly reduced. It is illustrated in figure 8.3. The colours indicate which memory block can be stored at which location in the cache.

set-associative cache: A combination of the previous two: each memory block is hashed to one block-set in the cache consisting of several *ways* (slot to store one block). Quick search for the tag needs only to be conducted within the set. The concept is shown in figure 8.4. The colours again indicate which memory block can be stored at which location in the cache, i.e., block-sets of several ways in the cache have the same colour. Note that they do not necessarily need to be co-localized like in this illustration.

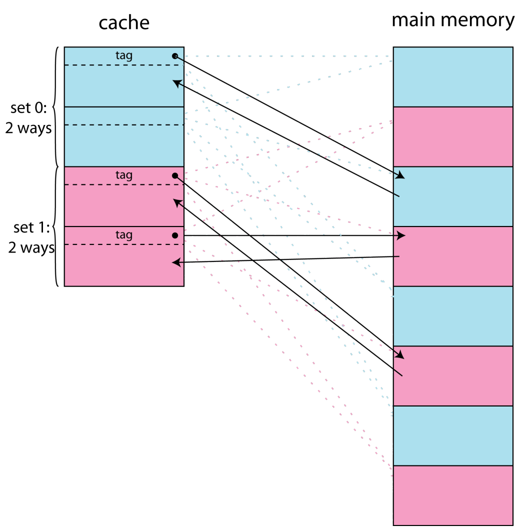


Figure 8.4: Set associative cache

8.1.1.2 Cache coherency

A write operation will lead to a temporary inconsistency between the content of the cache and the main memory, socalled *dirty* cache. Several strategies are used in different designs to correct this inconsistency with varying delay. Major strategies are:

write through: a simple policy where each write to the cache is followed by a write to the main memory. Thus, the write operations do not really profit from the cache.

write back: delayed write back where a block that has been written to in the cache is marked as *dirty*. Only when dirty blocks are reused for another memory block will they be written back into the main memory.

Another problem occurs if devices other than the CPU or multiple cores in the CPU with their own L1 cache have access to the main memory. In that case the main memory might be changed and the content in the cache will be out of date, socalled *stale* cache, but we will not delve into the methods to handle these situations within this issue of the compendium.

8.1.1.3 Cache block replacement strategies

Another strategic choice that has an influence on the caching performance arises in associative and set associative cache, if a new block needs to replace a block previously stored in the cache. (In directly mapped cache there is no choice as to which block to replace.) Again, different stratagems are possible:

Least recently used (LRU): this seems intuitively quite reasonable but requires a good deal of administrative processing (causing delay): Usually a 'used' flag is set per block in the cache when it is accessed. This flag is reset in fixed intervals and a time tag is updated for all blocks that have been used. These time tags have either to be searched before replacing a block or a queue can be maintained and updated whenever the time tags are updated.

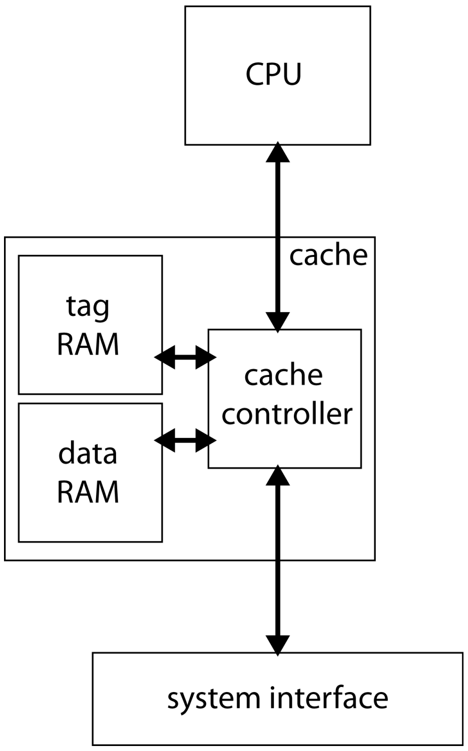


Figure 8.5: Look-through architecture

First in – first out (FIFO): is simpler. The cache blocks are simply organized in a queue (ring buffer).

Random: Both LRU and FIFO are in trouble if a program works several times sequentially through a portion of memory that is bigger than the cache: the block that is cast out will very soon be needed again. A random choice will do better here.

Hybrid solutions, e.g., using FIFO within a set of blocks that is randomly chosen are also used in an attempt to combine the positive properties of the approaches

8.1.1.4 Cache Architecture

There are two distinct cache architectures with respect to where to place the cache in relation to the bus between the CPU and the main memory referred to as *look-through* and *look-aside*.

Look-through architecture: The cache is physically placed between CPU and memory (system interface), see figure 8.5.

- Memory access is initiated *after* a cache miss is determined (i.e., with a delay).
- Only if a cache miss is determined, is a memory access initiated.
- CPU can use cache while memory is in use by other units.

Look-aside architecture: The cache shares the bus between CPU and memory (system interface), see figure 8.6.

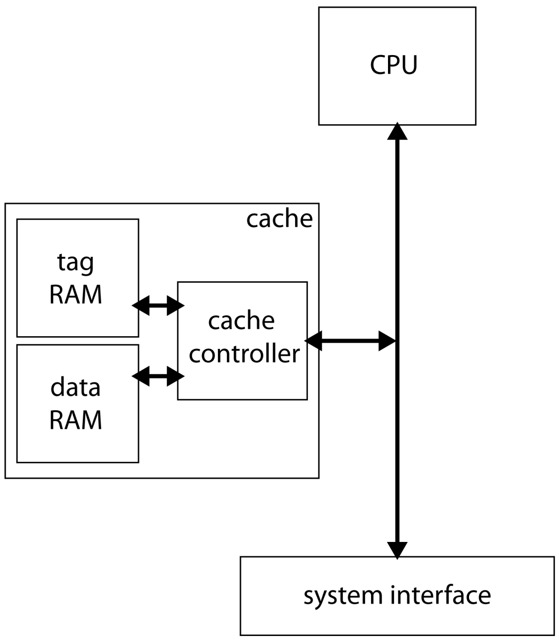


Figure 8.6: Look-aside architecture

- Memory access is initiated *before* a cache miss is determined (i.e., with no delay).
- With a miss, the cache just listens in and 'snarfs' the data.
- Only if a cache hit is determined, does the cache takes over.
- CPU cannot use cache while other units access memory.

8.1.2 Virtual Memory

Virtual memory extends the amount of main memory as seen by programs/ processes beyond the capacity of the *physical* memory. Additional space on the hard drive (*swap space*) is used to store a part of the virtual memory that is, at present, not in use. The task of the virtual memory controller is quite similar to a cache controller: it distributes data between a slow and fast storage medium.

A virtual memory controller may simply be part of the operating system rather than a hardware component, but most in most designs today there is a HW memory management unit (MMU) using a translation look-aside buffer (TLB) that supports virtual memory on the hardware level.

The principle of virtual memory is that each *logic* address is translated into a *physical* address, either in the main memory or on the hard drive as depicted in figure 8.7. Processes running on the CPU only see the logic addresses and a coherent virtual memory.

A pointer for each individual logic address would require as much space as the entire virtual memory. Thus, a translation table is mapping memory blocks (called *pages* (fixed size) or *segments* (variable size)). A logic address

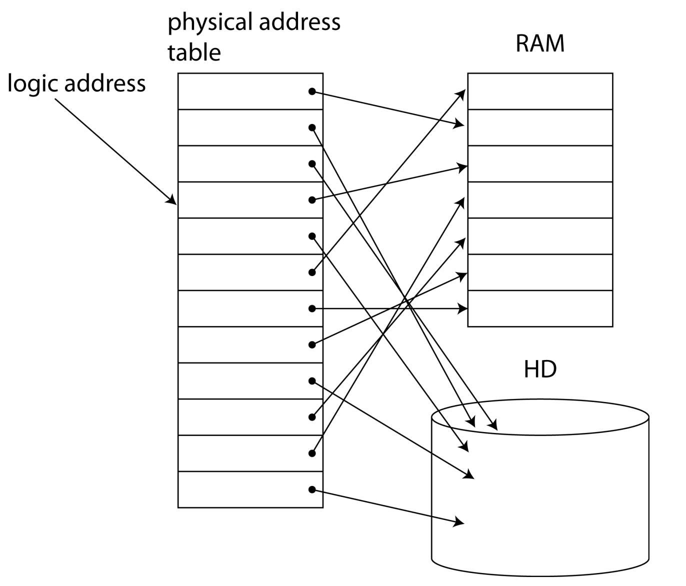


Figure 8.7: The principle of virtual memory

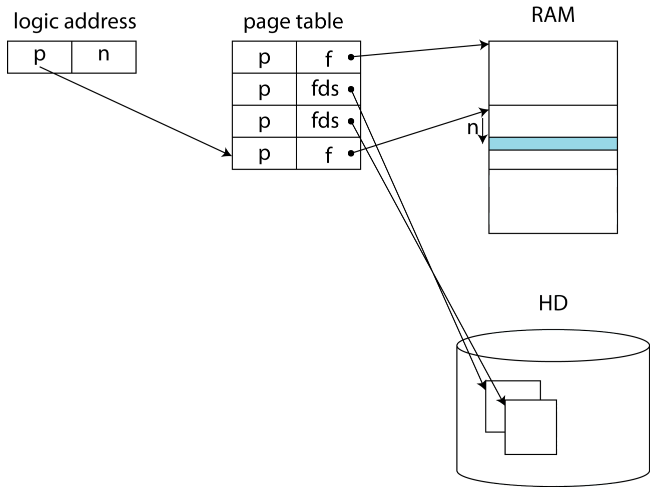


Figure 8.8: Virtual memory paging

can, thus, be divided into a page number and a page offset. A location in memory that holds a page is called page *frame* (figure 8.8).

A translation look-aside buffer (TLB) is a cache for the page table, accelerating the translation of logic to physical address by the MMU. The interaction of these elements is shown in the block diagram and flow chart of figure 8.9.

Note that the penalty for page-failures is much more severe than that for cache misses, since a hard drive has an access time that is up to 50000 times longer than that of the main memory and about 1 million times longer than a register or L1 cache access (compare table 8.1), whereas the penalty of a cache miss is roughly less than a 100 times increased access time.

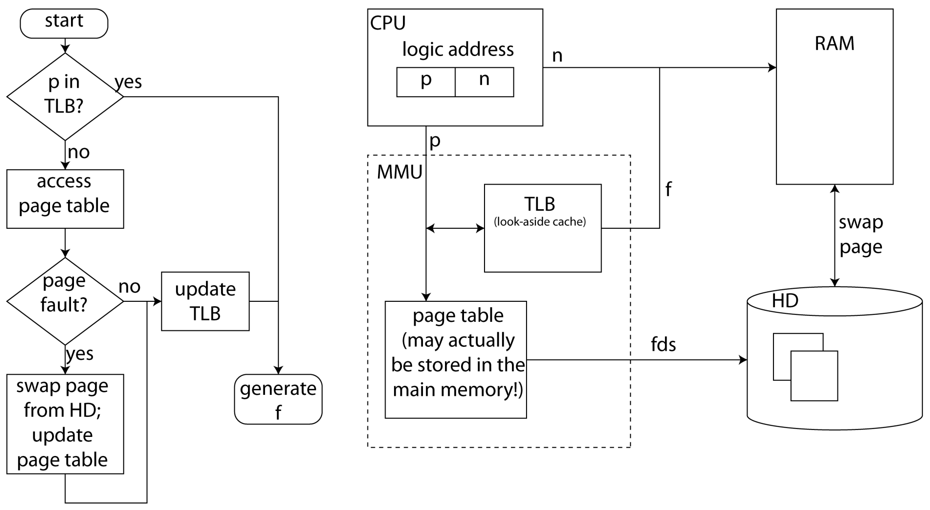


Figure 8.9

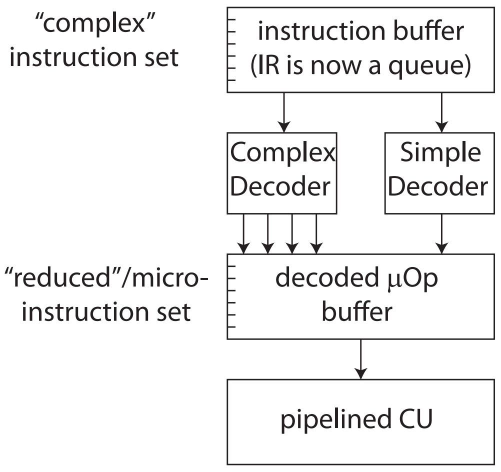

"complex" instruction set
instruction buffer (IR is now a queue)
Complex Decoder
Simple Decoder
"reduced"/micro-instruction set
decoded µOp buffer
pipelined CU

Figure 8.10: Decoding of complex instructions into simple micro-instructions in modern pipelined CPUs.

8.2 Pipelining

To accelerate the execution of instructions computer architectures today divide the execution into several stages. The CPU is designed in a way that allows it to execute these stages by independent subunits and such that each stage needs one clock cycle.[1] Thus, the first stage's sub-unit can already fetch a new instruction, while the second stage's sub-unit is still busy with the first instruction.

To achieve the necessary uniformity of the instructions (all going through the same sub-steps) the set of instructions had initially been reduced from what was normal at the time. This used to be known as reduced instruction set computer (RISC) architecture as opposed to complex instruction set computer (CISC). Today, however, the instruction sets tend to become more

[1] It would also work if all steps used more than one but the same number of clock cycles.

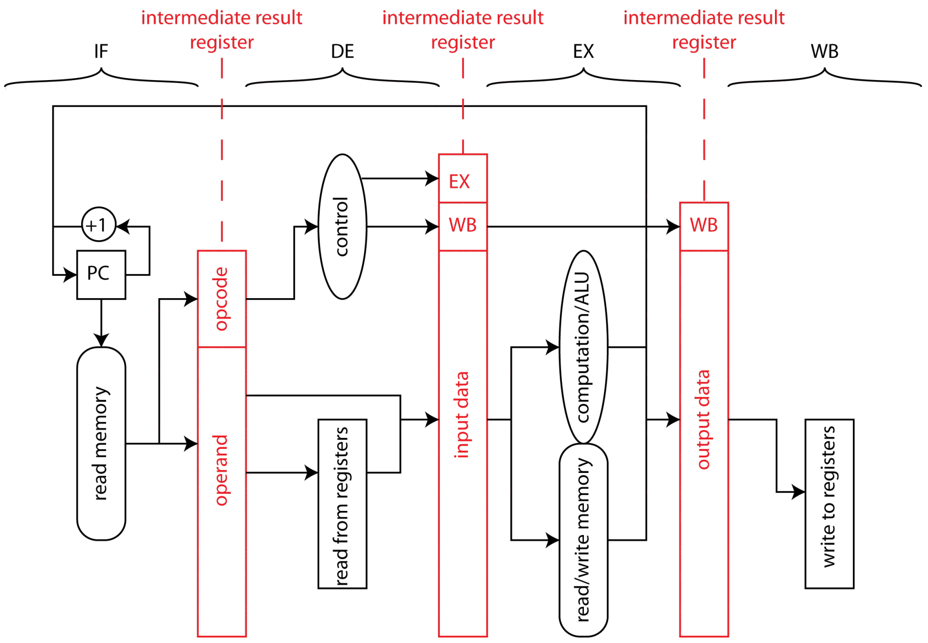


Figure 8.11: 4-stage pipeline simplified block diagram

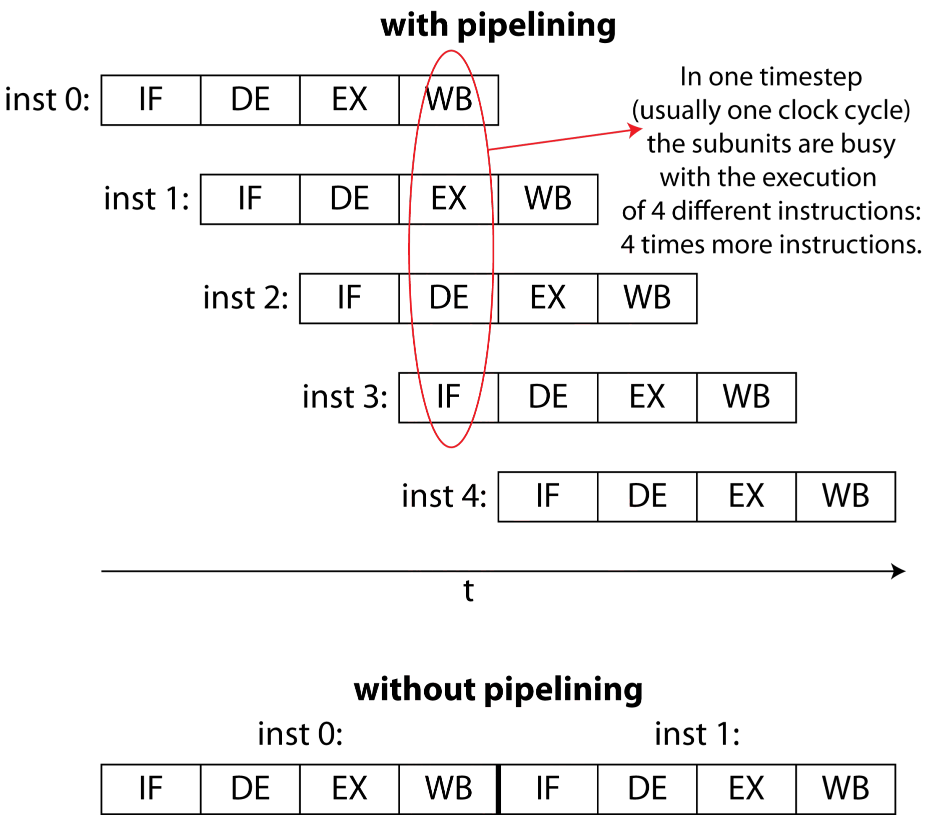


Figure 8.12: 4-stage pipeline execution

complex again, still allowing pipelining, and have in that regard surpassed the CISC architectures of old. To still achive pipelining a trick is used internally, a kind of “hardware compiler” for complex instructions: they are decoded into several simple micro-instructions before execution as illustrated in figure 8.10. Be aware that the term *micro-instruction* has been used historically and previously in this compendium with different meaning in the context of micro-architectures. Do not confuse them!

The Pentium III has a pipeline consisting of 16 and the Pentium 4 even 31 stages, but the first pipelines used these following 4, that we will employ to explain the concept:

1) IF: instruction fetch (get the instruction)
2) DE: decode and load (from a register)
3) EX: execute (e.g., use the ALU or access the memory)
4) WB: write back (write the result to a register)

Figure 8.11 shows a rough block diagram on how a 4-stage pipeline might be implemented. Intermediate registers pass on intermediate results and the remaining instruction/operation codes for the remaining stages.

The pipelined execution of 5 instructions is depicted in the top graph in figure 8.12. At the bottom the execution of just two equivalent instructions on a CPU that simply executes them in sequence is shown for comparison. Note that the pipelined architecture with four stages has a 4 times higher throughput of instructions. Still, the speed-up (the ratio of the execution time of a non-pipelined and a pipelined execution) is not quite 4, since the initial delay of the first instruction cannot be avoided. Thus, in our little illustration, the pipeline executes 5 instructions (and not 8) in the time the sequential version needs to execute 2. To express this in terms of speed-up of 5 instructions: the pipelined version needs 4 clock cycles to fill the pipeline and then issues the result of one instruction per clock, i.e., the execution time is 8 clock cycles. The sequential execution of 5 instructions will need 20 clock cycles. Thus, the speedup is 8/5=1.6.

In general for a k stage pipeline and the execution of n instructions, the speedup is:

$$\frac{nk}{k+n-1} \tag{8.1}$$

This equation approaches the ideal speedup of k for large programs. However, there are other hindrances as well: pipelining hazards (see section 8.2.1).

Another popular measure of the performance of the pipeline is the *average clock cycles per instructions* (CPI). Ideally, it would be 1, but due to the initial delay of ‘filling’ the pipeline, it is:

$$\mathrm{CPI} = \frac{k+n-1}{n} \tag{8.2}$$

More pipeline stages split up instructions in ever simpler parts and thus allow faster clock speeds but pose a bigger design challenge.

8.2.1 Pipelining Hazards

Other causes that limit the pipelining speed-up are called pipelining hazards. There are three major classes of these hazards:

- resource hazards
- data hazards
- control hazards

Hazards can be reduced by clever program compilation. In the following however, we will look at hardware solutions. In practice both are used in combination.

8.2.1.1 Resource Hazards

An example of instructions typically exposed to resource hazards are memory accesses. We have earlier referred to the von Neumann bottle neck as the limitation to only one memory access at a time. For pipelined operation, this means that only one instruction in the pipeline can have memory access at a time. Since always one of the instructions will be in the instruction fetch phase, a load or write operation of data to the memory is not possible without *stalling* the pipeline.

Several techniques are employed to ameliorate the problem:

Register File Instructions in pipelined architectures make extensive use of local general purpose registers, in some architectures organized in a *register file* for data input and output, and avoid access to the main memory. The register file is in effect a small RAM (e.g., with only a 3-bit address space) with (commonly) two parallel read ports (addresses and data) and (commonly) one parallel write port. It does, thus, allow three parallel accesses at the same time. In addition it is a specialized very fast memory within the CPU allowing extremely short access times. Still, also registers in the register file can be cause for resource hazards if two instructions want to access the same port in different pipeline stages.

Separate Data and Instruction Cache Another improvement is the so-called Harvard architecture, different from the von Neumann model insofar as there are two separate memories again for data and instructions, on the level of the cache memory. Thus, the instruction fetch will not collide with data access unless there is a cache miss of both.

Memory access still constitutes a hazard in pipelining. E.g., in the first 4-stage SPARC processors memory access uses 5 clock cycles for reading and 6 for writing, and thus always impede pipe-line speed up.

Dependent on the CPU architecture a number of other resources may be used by different stages of the pipeline and may thus be cause for resource hazards, for example:

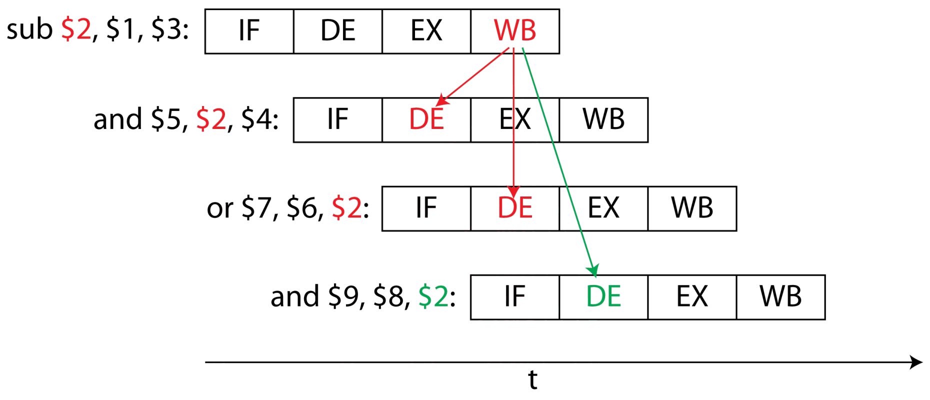


Figure 8.13: Data hazard illustration

- memory, caches,
- registers (register file)
- buses
- ALU
- ...

8.2.1.2 Data Hazards

Data hazards can occur when instructions that are in the pipeline simultaneously access the same data (i.e., register). Thus, it can happen that an instruction reads a register, before a previous instruction has written to it. This is illustrated in figure 8.13.

There are a number of measures one can take:

Stalling: A simple solution that ensures correctness but degrades performance, is to detect a dependency in the IF stage and stall the execution of subsequent instructions until the crucial instruction has finished its WB.

Shortcuts/Forwarding: In this solution there is a direct data path from the EX/WB intermediate result register to the execution stage input (e.g., the ALU). If a data hazard is detected this direct data path supersedes the input from the DE/EX intermediate result register.

8.2.1.3 Control Hazards

Simple pipelining assumes in a first approximation that there are no program jumps and 'pre-fetches' always the next instruction from memory into the pipeline. The target of jump instructions, however, is usually only computed in the EX stage of a jump instruction. A this time, two more instructions have already entered the pipeline and are in the IF and DE stage. If the jump is taken these instructions should not be executed and need to be prevented from writing their results in the WB stage or accessing the memory in the EX stage. The pipeline needs to be *flushed*.

Possible measures:

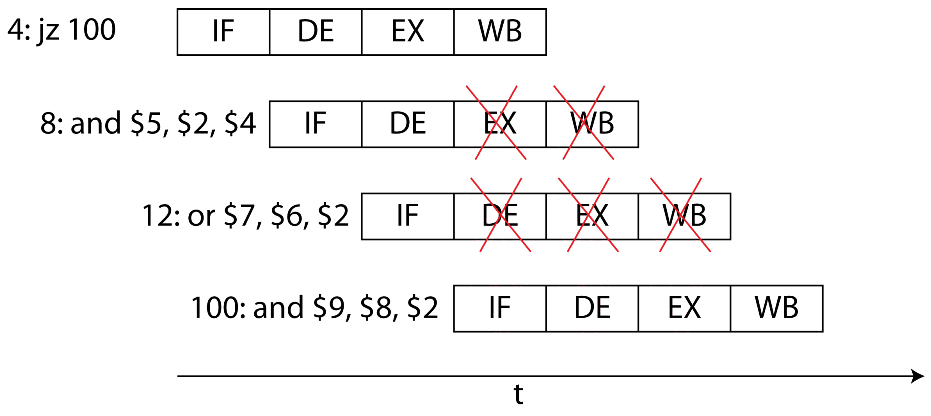


Figure 8.14: Control hazard illustration

Always Stall: Simply do not fetch any more instructions until it is clear if the branch is taken or not. This ensures correctness but is of course a burden upon the performance.

Jump Prediction: Make a prediction and fetch the predicted instructions. Only if the prediction is proven wrong, flush the pipeline. Variants:

- assume branch not taken (also a static prediction)
- static predictions (assumes that a branch is always taken or not according to the type of instruction)
- dynamic predictions (uses some local memory to keep a short history of previous branching behaviour upon which to base the predictions on)

Hardware Doubling: By doubling the hardware of some of the pipeline stages one can continue two pipelines in parallel for both possible instruction addresses. After it is clear, if the branch was taken or not, one can discard/flush the irrelevant pipeline and continue with the right one.

Of course, if there are two jumps or more just after each other, this method fails on the second jump and the pipeline needs to stall.

The occurrence of branching instructions is quite frequent in normal program code and thus, the handling of control hazards is possibly the most important of all the hazards. Good jump prediction can significantly increase performance.

An example: a pipelined CPU is executing a program and a fraction of P_b (very program dependent but often quite high, e.g., in the vicinity of 20%) of the executed instructions are branching instructions. The CPU pursues a 'assume branch not taken' strategy, but the probability of a branch being taken is P_t (often quite high, i.e., more than 50%) and the penalty in that case is s additional clock cycles to flush the pipeline. Then the CPI becomes (assuming the number of instructions is large enough to rend the 'filling' of the pipeline negligible):

$$\text{CPI} = (1 - P_b) + P_b(1 - P_t) + P_b P_t (1 + s) = n + P_b P_t s \qquad (8.3)$$

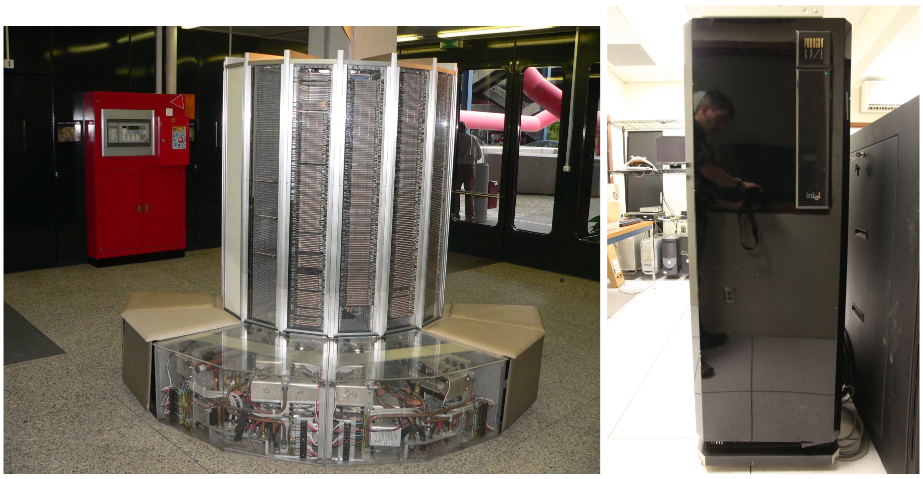


Figure 8.15: The Cray-1 with transparent panels (left) and the Paragon XP-E (right)

Using some realistic example of $P_b = 0.2$, $P_t = 0.7$ and $s = 2$, the CPI is 1.28, so 28% less performance due to control hazards.

8.2.2 Conclusion

Pipelining speeds up the instruction throughput (although the execution of a single instruction is not accelerated). The ideal speed-up cannot be reached, because of this, and because of instruction inter-dependencies that sometimes require that an instruction is finished before another can begin. There are techniques to reduce the occurrence of such hazards, but they can never be avoided entirely.

8.3 Superscalar CPU

The concept of a CPU that we have discussed so far was that of a *scalar* processors, in as far as it does not execute operations in parallel and produce only a single result data item at a time.

8.3.1 Brief Historical Detour into Supercomputing

Vector processors (as opposed to scalar processors) were fashionable in high performance computing for a period, most prominently the Cray-1 (figure 8.15, to the left) in 1976 that had 8 vector registers of 64 words of 64-bit length. Vector processors perform ‘single instruction multiple data-stream’ (SIMD) computations, i.e., they execute the same operation on a vector instead of a scalar. Some machines used parallel ALU’s but the Cray-1 used a dedicated pipelining architecture that would fetch a single instruction and then execute it efficiently, e.g., 64 times, saving 63 instruction fetches.

Vector computers lost popularity with the introduction of multi-processor computers such as Intel’s Paragon series (figure 8.15, to the right) of *massively parallel supercomputers*: It was cheaper to combine multiple (standard) CPU’s rather than designing powerful vector processors, even considering a bigger *communication overhead*, e.g., in some architectures with a single shared memory/system bus the instructions and the data need to be fetched and written in sequence for each processor, making the von Neumann bottleneck more severe. Other designs, however, had local memory and/or parallel memory access and many clever solutions were introduced.

But even cheaper and obtainable for the common user are Ethernet clusters of individual computers, or even computer grids connected over the Internet. Both of these, obviously, suffer from massive communication overhead and especially the latter are best used for socalled 'embarrassingly parallel problems', i.e., computation problems that do require no or minimal communication of the computation nodes.

Designing more complicated integrated circuits has become cheaper with progressing miniaturization, such that several processing units can now be accommodated on a single chip which has now become standard with AMD and Intel processors. These multi-core processors have many of the advantages of multi processor machines, but with much faster communication between the cores, thus, reducing communication overhead. (Although, it has to be said that they are most commonly used to run individual independent processes, and for the common user they do not compute parallel problems.)

8.3.2 Superscalar Principle

Superscalar processors were introduced even before multi-core and all modern designs belong to this class. The name is supposed to indicate that they are something more parallel than scalar processors but not quite vector processors. Like vector processors with parallel ALUs, they are actually capable of executing instructions *in parallel*, but in contrast to vector computers, they are *different* instructions. Instead of replication of the basic functional units n-times in hardware (e.g., the ALU), superscalar processors exploit the fact that there already *are* multiple functional units. For example, many processors do sport both an ALU and a FPU. Thus, they should be able to execute an integer- and a floating-point operation *simultaneously*. Data access operations do not require the ALU nor the FPU (or have a dedicated ALU for address operations) and can thus also be executed at the same time.

For this to work, several instructions have to be fetched in parallel, and then *dispatched*, either in parallel if possible, or in sequence if necessary, into parallel pipelines dedicated to one particular *type* of instruction. Some additional stages that deal with instruction reordering are added to the pipelining structure. In the following it is assumed they are integrated in the IF and WB stages, to stick with the 4-stage model.

The principle of superscalar execution is illustrated in figure 8.16. In this example the CPU is equiped with three 4-stage pipelines, one for arithmetic instructions, one for floating point instructions and one for memory access instructions. The IF stage can get up to 3 instructions in parallel and deposits them in an instruction reorder buffer (ROB) which holds all instructions awaiting execution. In our example the ROB has only 3 entries (usually it should have a good deal more than there are parallel pipelines).

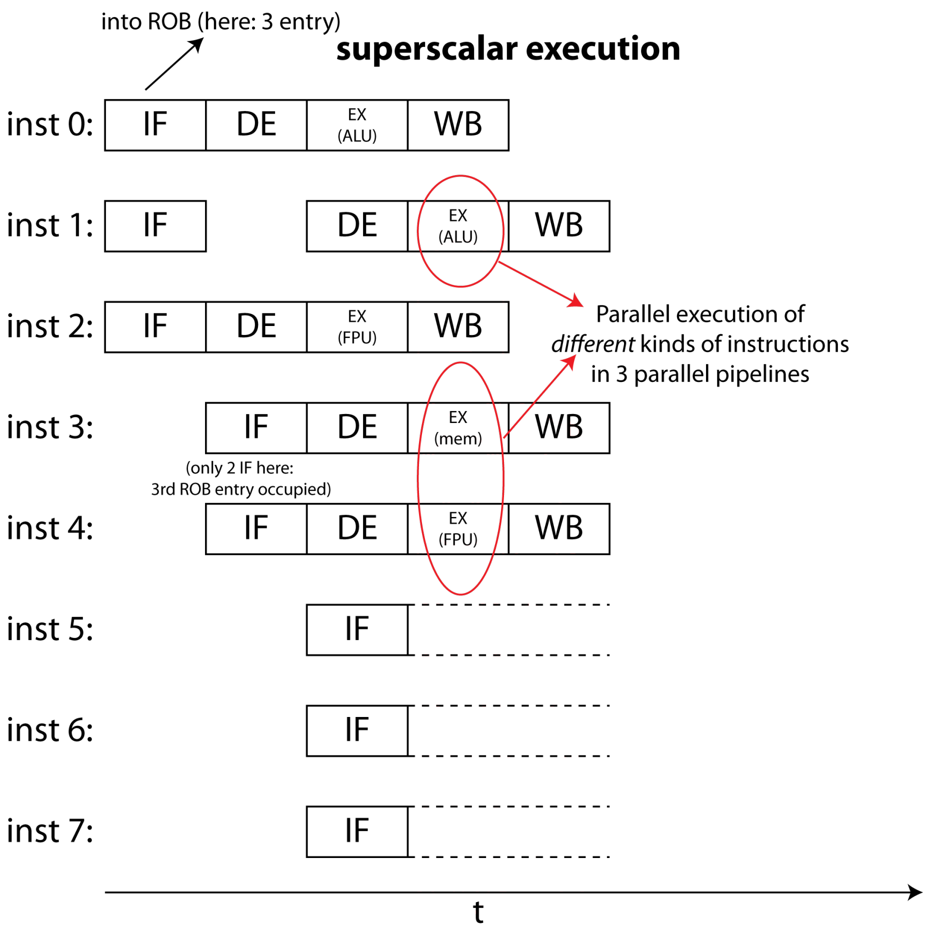


Figure 8.16: Principle of superscalar execution

From the ROB, the instructions are dispatched to the appropriate pipelines. Among the first three instructions, there are two arithmetic instructions, so only one of them can immediately be dispatched. The other has to wait and is dispatched in the next clock cycle together with the two next instructions, a memory access and a floating point operation.

Obviously, the reordering of instructions and their parallel execution can again lead to a number of hazards, the resolution of which will not be discussed in this issue of the compendium.

Superscalar processors can ideally achieve an average clock cycle per instruction (CPI) smaller than 1, and a speedup higher than the number of pipelining stages k (which is saying the same thing in two different ways).

Compiler level support can group instructions to optimize the potential for parallel execution.

As an example: the Intel Core 2 microarchitecture has 14 pipeline stages and can execute up to 4-6 instructions in parallel.

Some Elements in Superscalar Architectures:

(Micro-)instruction reorder buffer (ROB): Stores all instructions that await execution and dispatches them for *out-of-order execution* when appropriate. Note that, thus, the order of execution may be quite different from the order of your assembler code. Extra steps have to be taken to avoid and/or handle hazards caused by this reordering.

Retirement stage: The pipelining stage that takes care of finished instructions and makes the result appear consistent with the execution sequence that was intended by the programmer.

Reservation station registers: A single instruction reserves a set of these registers for all the data needed for its execution on its functional unit. Each functional unit has several slots in the reservation station. Once all the data becomes available and the functional unit is free, the instruction is executed.

Part II

Low-level programming

Chapter 9

Introduction to low-level programming

This part of the INF2270 compendium describes **low-level programming**, i.e., programming very close to the computer hardware.

Chapter 10 on page 83 is a reference guide to the programming language **C** which is generally regarded as a rather low-level language among the high-level ones. The chapter is no introduction to programming in general but rather a compilation of information you might need when programming.

Chapter 11 on page 87 explains the most common encodings used today to store characters. Anyone programming at a low level ought to know about these encodings.

Chapter 12 on page 91 describes programming in assembly language. Like chapter 10 it is no "teach yourself" guide but (hopefully) a useful place to consult when you are programming. In particular, the instruction tables 12.2 and 12.3 on pages 95–96 should provide a lot of information in a very compact form.

Chapter 10

Programming in C

C is one of the most popular languages today and has been so for more than 30 years. One of the reasons for its success has been that it combines quite readable code with very high execution speed.

There have been several versions of C; in this course we will use ANSI C from 1988.

10.1 Data

C has quite a substantial set of data types.

10.1.1 Integer data

C can store 1-, 2- and 4-byte integer data, as shown i table 10.1 on the next page.[1]

10.1.1.1 Logical values

C har no specialised data type for logical (often called “Boolean”) values; instead any integer or pointer type is used with the following meaning:

$$\begin{aligned} 0 &: \quad \text{false} \\ \neq 0 &: \quad \text{true} \end{aligned}$$

10.1.1.2 Characters

C has no special type for characters either; they are stored in `unsigned char` variables using the encoding described in 11.1 on page 88.[2]

10.1.2 Texts

Since C has no native data type for texts, an array of `unsigned char`s is used; the end of the text is marked by a byte containing 0.

[1] Most C implementations can also store 8-byte integer data which have the type `long long`, but this is not in the ANSI C standard.

[2] More recent versions of C have support for more extensive character sets, like Unicode, but this is not covered in this course.

Name	Alternativ name	# bytes
signed char	char†	1
unsigned char	char†	1
short	signed short	2
unsigned short		2
int	signed int	2–4
unsigned int	unsigned	2–4
long	signed long	4
unsigned long		4

Table 10.1: Integer data types in C († the exact meaning of char is undefined.)

10.1.3 Floating-point data

In C, you can choose between using float or double for your floating-point data, as shown in table 10.2.

Name	# bytes	Max value	Precision
float	4	$\approx 3{,}4 \cdot 10^{38}$	7 digits
double	8	$\approx 1{,}8 \cdot 10^{308}$	16 digits

Table 10.2: Floating-point data types in C

10.2 Statements

The statements in C are listed in table 10.3.

Block	`{ ⟨S⟩ ⟨S⟩ ... }`

Break	`break;       /* exit loop/switch */`
	`continue;    /* go to top of loop */`

Expression	`⟨expr⟩;`

Loop	`while (⟨expr⟩) ⟨S⟩`
	`do ⟨S⟩ while (⟨expr⟩)`
	`for (⟨expr⟩; ⟨expr⟩; ⟨expr⟩) ⟨S⟩`

Null statement	`;`
Return	`return ⟨expr⟩;`

Selection	`if (⟨expr⟩) ⟨S⟩`
	`if (⟨expr⟩) ⟨S⟩ else ⟨S⟩`
	
	`switch (⟨expr⟩) {`
	`  case ⟨const⟩: ⟨S⟩ ⟨S⟩ ...`
	`  case ⟨const⟩: ⟨S⟩ ⟨S⟩ ...`
	`  default:      ⟨S⟩`
	`}`

Table 10.3: The statements in C

10.3 Expressions

C has quite an extensive set of expression operators with a confusing number of precedence level; use parenthesis if you are in any doubt. The whole set of operators is shown in table 10.4 on the next page.

Level	Op	Meaning
15	( )	Function call
	[ ]	Array element
	.	Member (of struct or union)
	->	Member (accessed via pointer)
14	!	Logical negation
	~	Masking negation
	-	Numeric negation
	++	Increment
	--	Decrement
	&	Address
	*	Indirection
	(*type*)	Type cast
	sizeof	Size in bytes
13	*	Multiplication
	/	Division
	%	Remainder
12	+	Addition
	-	Subtraction
11	<<	Left shift
	>>	Right shift
10	<	Less than test
	<=	Less than or equal test
	>	Greater than test
	>=	Greater than or equal test
9	==	Equality test
	!=	Inequality test
8	&	Masking and
7	^	Masking exclusive or
6	|	Masking or
5	&&	Logical and
4	||	Logical or
3	? ;	Conditional evaluation
2	=	Assignment
	*= /= %= += -=	Updating
	<<= >>= &= ^= !=	
1	,	Sequential evaluation

Table 10.4: The expression operators in C

Chapter 11

Character encodings

A **character encoding** is a table of which numbers represent which character. There are dozens of encoding; the four most common today are **ASCII**, **Latin-1**, **Latin-9** and **Unicode.**

11.1 ASCII

This very old 7-bit encoding survives today only as a subset of other encodings; for instance, the left half of Latin-1 (see Table 11.1 on the following page) is the original ASCII encoding.

11.2 Latin-1

The official name of this 8-bit encoding is ISO 8859-1; it is shown in Table 11.1 on the next page.

11.3 Latin-9

This encoding is a newer version of Latin-1; its official name is ISO 8859-15. Only eight characters were changed; they are shown in Table 11.2 on page 89.

11.4 Unicode

Unicode is a gigantic 21-bit encoding intended to encompass all the world's characters; for more information, see http://www.unicode.org/.

11.4.1 UTF-8

UTF-8 is one of several ways to store Unicode's 21-bit representation numbers. One advantage of UTF-8 is that it is quite compact; the most commonly used characters are stored in just one byte, others may need two or three or up to four bytes, as shown in Table 11.3 on page 89.

ISO 8859-1

0	000	00		32	040	20		64	100	40	@	96	140	60	‘	128	200	80		160	240	A0		192	300	C0	À	224	340	E0	à
1	001	01		33	041	21	!	65	101	41	A	97	141	61	a	129	201	81		161	241	A1	¡	193	301	C1	Á	225	341	E1	á
2	002	02		34	042	22	"	66	102	42	B	98	142	62	b	130	202	82		162	242	A2	¢	194	302	C2	Â	226	342	E2	â
3	003	03		35	043	23	#	67	103	43	C	99	143	63	c	131	203	83		163	243	A3	£	195	303	C3	Ã	227	343	E3	ã
4	004	04		36	044	24	$	68	104	44	D	100	144	64	d	132	204	84		164	244	A4	¤	196	304	C4	Ä	228	344	E4	ä
5	005	05		37	045	25	%	69	105	45	E	101	145	65	e	133	205	85		165	245	A5	¥	197	305	C5	Å	229	345	E5	å
6	006	06		38	046	26	&	70	106	46	F	102	146	66	f	134	206	86		166	246	A6	¦	198	306	C6	Æ	230	346	E6	æ
7	007	07		39	047	27	’	71	107	47	G	103	147	67	g	135	207	87		167	247	A7	§	199	307	C7	Ç	231	347	E7	ç
8	010	08		40	050	28	(	72	110	48	H	104	150	68	h	136	210	88		168	250	A8	¨	200	310	C8	È	232	350	E8	è
9	011	09		41	051	29	)	73	111	49	I	105	151	69	i	137	211	89		169	251	A9	©	201	311	C9	É	233	351	E9	é
10	012	0A		42	052	2A	*	74	112	4A	J	106	152	6A	j	138	212	8A		170	252	AA	ª	202	312	CA	Ê	234	352	EA	ê
11	013	0B		43	053	2B	+	75	113	4B	K	107	153	6B	k	139	213	8B		171	253	AB	«	203	313	CB	Ë	235	353	EB	ë
12	014	0C		44	054	2C	,	76	114	4C	L	108	154	6C	l	140	214	8C		172	254	AC	¬	204	314	CC	Ì	236	354	EC	ì
13	015	0D		45	055	2D	−	77	115	4D	M	109	155	6D	m	141	215	8D		173	255	AD	-	205	315	CD	Í	237	355	ED	í
14	016	0E		46	056	2E	.	78	116	4E	N	110	156	6E	n	142	216	8E		174	256	AE	®	206	316	CE	Î	238	356	EE	î
15	017	0F		47	057	2F	/	79	117	4F	O	111	157	6F	o	143	217	8F		175	257	AF	¯	207	317	CF	Ï	239	357	EF	ï
16	020	10		48	060	30	0	80	120	50	P	112	160	70	p	144	220	90		176	260	B0	°	208	320	D0	Ð	240	360	F0	ð
17	021	11		49	061	31	1	81	121	51	Q	113	161	71	q	145	221	91		177	261	B1	±	209	321	D1	Ñ	241	361	F1	ñ
18	022	12		50	062	32	2	82	122	52	R	114	162	72	r	146	222	92		178	262	B2	²	210	322	D2	Ò	242	362	F2	ò
19	023	13		51	063	33	3	83	123	53	S	115	163	73	s	147	223	93		179	263	B3	³	211	323	D3	Ó	243	363	F3	ó
20	024	14		52	064	34	4	84	124	54	T	116	164	74	t	148	224	94		180	264	B4	´	212	324	D4	Ô	244	364	F4	ô
21	025	15		53	065	35	5	85	125	55	U	117	165	75	u	149	225	95		181	265	B5	µ	213	325	D5	Õ	245	365	F5	õ
22	026	16		54	066	36	6	86	126	56	V	118	166	76	v	150	226	96		182	266	B6	¶	214	326	D6	Ö	246	366	F6	ö
23	027	17		55	067	37	7	87	127	57	W	119	167	77	w	151	227	97		183	267	B7	·	215	327	D7	×	247	367	F7	÷
24	030	18		56	070	38	8	88	130	58	X	120	170	78	x	152	230	98		184	270	B8	¸	216	330	D8	Ø	248	370	F8	ø
25	031	19		57	071	39	9	89	131	59	Y	121	171	79	y	153	231	99		185	271	B9	¹	217	331	D9	Ù	249	371	F9	ù
26	032	1A		58	072	3A	:	90	132	5A	Z	122	172	7A	z	154	232	9A		186	272	BA	º	218	332	DA	Ú	250	372	FA	ú
27	033	1B		59	073	3B	;	91	133	5B	[	123	173	7B	{	155	233	9B		187	273	BB	»	219	333	DB	Û	251	373	FB	û
28	034	1C		60	074	3C	<	92	134	5C	\	124	174	7C	|	156	234	9C		188	274	BC	¼	220	334	DC	Ü	252	374	FC	ü
29	035	1D		61	075	3D	=	93	135	5D	]	125	175	7D	}	157	235	9D		189	275	BD	½	221	335	DD	Ý	253	375	FD	ý
30	036	1E		62	076	3E	>	94	136	5E	^	126	176	7E	~	158	236	9E		190	276	BE	¾	222	336	DE	Þ	254	376	FE	þ
31	037	1F		63	077	3F	?	95	137	5F	_	127	177	7F		159	237	9F		191	277	BF	¿	223	337	DF	ß	255	377	FF	ÿ

Table 11.1: The ISO 8859-1 (Latin-1) encoding. (The numbers in each cell are the character's encoding number in decimal, octal and hex.)

	$A4_{hex}$	$A6_{hex}$	$A8_{hex}$	$B4_{hex}$	$B8_{hex}$	BC_{hex}	BD_{hex}	BE_{hex}
Latin-1	¤	¦	¨	´	¸	¼	½	¾
Latin-9	€	Š	š	Ž	ž	Œ	œ	Ÿ

Table 11.2: The difference between Latin-1 and Latin-9

0_{hex}–$7F_{hex}$	0xxxxxxx			
80_{hex}–$7FF_{hex}$	110xxxxx	10xxxxxx		
800_{hex}–$FFFF_{hex}$	1110xxxx	10xxxxxx	10xxxxxx	
10000_{hex}–$10FFFF_{hex}$	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx

Table 11.3: UTF-8 representation of Unicode characters

Chapter 12
Assembly programming

A computer executes **machine code** programs in which instructions are encoded as bit patterns; for instance, on an x86 processor, the five bytes

$$B8_{\text{hex}} \quad 13_{\text{hex}} \quad 00_{\text{hex}} \quad 00_{\text{hex}} \quad 00_{\text{hex}}$$

tell the processor to move the value 17 (=13_{hex}) to the %EAX register.

Since machine code is difficult to write, programmers use **assembly code** with mnemonic names instead. The instruction above is written as

```
        movl    $19, %eax
```

12.1 Assembler notation

An assembly language is quite simple.

12.1.1 Instruction lines

All instruction lines have the following parts:

$\langle$label:$\rangle$ $\langle$instr$\rangle$ $\langle$param$_1\rangle$, $\langle$param$_2\rangle$, … # $\langle$comment$\rangle$

Any part may be omitted.

12.1.2 Specification lines

Specification lines provide additional information for the assembler. We will use the following specifications:

.align *n* legger inn ekstra byte til adressen har *n* 0-bit nederst.

.bss switches to the BSS segment for uninitialized data.

.byte reserves one byte of data with a given initial value.

.data switches to the data segment.

.fill reserves the specified number of bytes as uninitialized data.

.globl specifies that a name is to be known globally (and not just within the file).

	Intel	AT&T
Constants (decimal)	4	$4
Constants (hex)	123h	$0x123
Registers	eax	%eax
Sequence	*res, op, op, ...*	*op, op, ..., res*
Size	mov	movl
Type specification	mov ax, WORD PTR v	
Indexing	[eax+1]	1(%eax)

Table 12.1: The major differences between AT&T and Intel assembler notation

.long reserves four byte of data with a given initial value.

.text switches to the program code segment.

.word reserves two byte of data with a given initial value.

12.1.3 Comments

The character "#" will make the rest of the line a comment. Blank lines are ignored.

12.1.4 Alternative notation

In this course, we will use the as/gcc assembler/compiler which adhere to the socalled **AT&T-notation**. Other assemblers, in particular those from Intel and Microsoft, follow the **Intel-notation**. Table 12.1 shows the major differences.

12.2 The assembler

The **assembler** translates assembly code into machine code. We will use the GNU assembler as but accessed through the GNU C compiler gcc:

```
$ gcc -m32 -o myprog myprog.c myfunc.s
```

(The -m32 option specifies that we are treating the processor as a 32-bit one.)

12.2.1 Assembling under Linux

gcc is available as part of every Linux distribution.

12.2.2 Assembling under Windows

gcc for Windows is available as part of the CygWin program package; it may be retrieved from the Ifi-DVD at http://www.cygwin.com/.

Note that CygWin uses a slightly different convention for global names: the name "*xxx*" in C is known as "*_xxx*" in the assembly language. Fortunately,

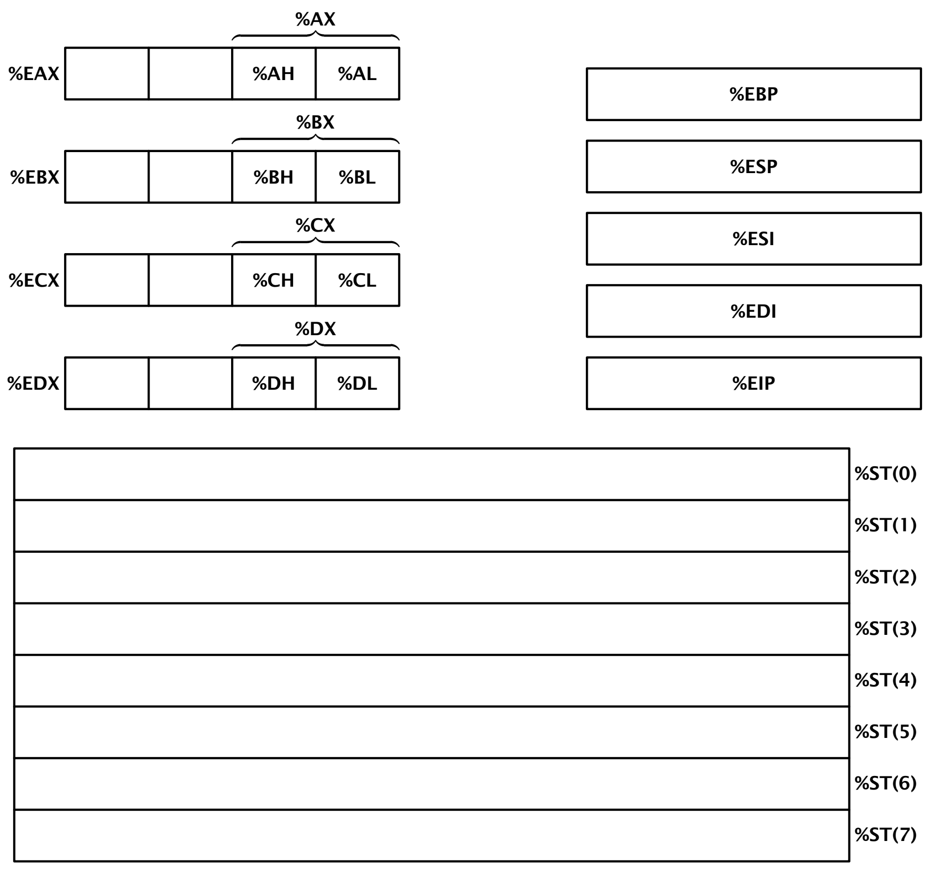


Figure 12.1: The most important x86/x87 registers

it is possible to comply with both the Linux and CygWin conventions by defining every global name twice, as in

```
        .globl  funcname
        .globl  _funcname
funcname:
_funcname:
           :
```

12.3 Registers

The most commonly used registers are shown in Figure 12.1.

12.4 Instruction set

Tables 12.2 and 12.3 list the subset of x86 instructions used in this course, and table 12.4 gives a similar list for the x87 floating-point instructions. The following notation is used in these tables:

$\{a\}$ is an address, usually in the form of a label name.

{bwl} is an instruction suffix indicating the size of the operation:

–b byte

–w word (= 2 bytes)
–l long (= 4 bytes)

$\{c\}$ is a constant, given as

- a decimal number (as in "$123")
- a hex number (as in "$0x1af")

$\{cmr\}$ is any of $\{c\}$, $\{m\}$ or $\{r\}$

$\{cr\}$ is either $\{c\}$ or $\{r\}$

$\{m\}$ is a memory reference, given as

- a label (i.e., a name declared somewhere)
- a number in decimal or hex notation (but no $ sign!)
- an indirect reference (as in "(%ESI)" or "4(%ESP)")
- an indexed reference (as in "8(%ESI,%ECX,4)" in which the memory address is 8 + %ESI + 4%ECX)

$\{mr\}$ is either a $\{m\}$ or a $\{r\}$

$\{r\}$ is a register (as in "%EAX")

$\{sl\}$ is one of

–l a double
–s a float

Instruction	**Explanation**	**Effect**	C	S	Z
Data movement					
lea{bwl} $\{cmr\}$,$\{mr\}$	Copy address	$\{mr\} \leftarrow$ Adr($\{cmr\}$)			
mov{bwl} $\{cmr\}$,$\{mr\}$	Copy data	$\{mr\} \leftarrow \{cmr\}$			
pop{wl} $\{r\}$	Pop value	$\{r\} \leftarrow$ pop			
push{wl} $\{cr\}$	Push value	push $\{cr\}$			
Block operations					
cld	Clear D-flag	$D \leftarrow 0$			
cmpsb	Compare byte	(%EDI) − (%ESI); %ESI ← %ESI ± 1; %EDI ← %EDI ± 1	✔	✔	✔
movsb	Move byte	(%EDI) ← (%ESI); %ESI ← %ESI ± 1; %EDI ← %EDI ± 1			
rep ⟨**instr**⟩	Repeat	Repeat ⟨instr⟩ %ECX times			
repnz ⟨**instr**⟩	Repeat until zero	Repeat ⟨instr⟩ %ECX times while Z			
repz ⟨**instr**⟩	Repeat while zero	Repeat ⟨instr⟩ %ECX times while $\bar{Z}$			
scasb	Scan byte	%AL − (%EDI); %EDI ← %EDI ± 1	✔	✔	✔
std	Set D-flag	$D \leftarrow 1$			
stosb	Store byte	(%EDI) ← %AL; %EDI ← %EDI ± 1			
Arithmetic					
adc{bwl} $\{cmr\}$,$\{mr\}$	Add with carry	$\{mr\} \leftarrow \{mr\} + \{cmr\} + C$	✔	✔	✔
add{bwl} $\{cmr\}$,$\{mr\}$	Add	$\{mr\} \leftarrow \{mr\} + \{cmr\}$	✔	✔	✔
dec{bwl} $\{mr\}$	Decrement	$\{mr\} \leftarrow \{mr\} - 1$		✔	✔
divb $\{mr\}$	Unsigned divide	%AL ← %AX/$\{mr\}$; %AH ← %AX mod $\{mr\}$	?	?	?
divw $\{mr\}$	Unsigned divide	%AX ← %DX:%AX/$\{mr\}$; %DH ← %DX:%AX mod $\{mr\}$	?	?	?
divl $\{mr\}$	Unsigned divide	%EAX ← %EDX:%EAX/$\{mr\}$; %EDX ← %EDX:%EAX mod $\{mr\}$	?	?	?
idivb $\{mr\}$	Signed divide	%AL ← %AX/$\{mr\}$; %AH ← %AX mod $\{mr\}$	?	?	?
idivw $\{mr\}$	Signed divide	%AX ← %DX:%AX/$\{mr\}$; %DH ← %DX:%AX mod $\{mr\}$	?	?	?
idivl $\{mr\}$	Signed divide	%EAX ← %EDX:%EAX/$\{mr\}$; %EDX ← %EDX:%EAX mod $\{mr\}$	?	?	?
imulb $\{mr\}$	Signed multiply	%AX ← %AL × $\{mr\}$	✔	?	?
imulw $\{mr\}$	Signed multiply	%DX:%AX ← %AX × $\{mr\}$	✔	?	?
imull $\{mr\}$	Signed multiply	%EDX:%EAX ← %EAX × $\{mr\}$	✔	?	?
imul{wl} $\{cmr\}$,$\{mr\}$	Signed multiply	$\{mr\} \leftarrow \{mr\} \times \{cmr\}$	✔	?	?
inc{bwl} $\{mr\}$	Increment	$\{mr\} \leftarrow \{mr\} + 1$		✔	✔
mulb $\{mr\}$	Unsigned multiply	%AX ← %AL × $\{mr\}$	✔	?	?
mulw $\{mr\}$	Unsigned multiply	%DX:%AX ← %AX × $\{mr\}$	✔	?	?
mull $\{mr\}$	Unsigned multiply	%EDX:%EAX ← %EAX × $\{mr\}$	✔	?	?
neg{bwl} $\{mr\}$	Negate	$\{mr\} \leftarrow -\{mr\}$	✔	✔	✔
sub{bwl} $\{cmr\}$,$\{mr\}$	Subtract	$\{mr\} \leftarrow \{mr\} - \{cmr\}$	✔	✔	✔
Masking					
and{bwl} $\{cmr\}$,$\{mr\}$	Bit-wise AND	$\{mr\} \leftarrow \{mr\} \wedge \{cmr\}$	0	✔	✔
not{bwl} $\{mr\}$	Bit-wise invert	$\{mr\} \leftarrow \overline{\{mr\}}$			
or{bwl} $\{cmr\}$,$\{mr\}$	Bit-wise OR	$\{mr\} \leftarrow \{mr\} \vee \{cmr\}$	0	✔	✔
xor{bwl} $\{cmr\}$,$\{mr\}$	Bit-wise XOR	$\{mr\} \leftarrow \{mr\} \oplus \{cmr\}$	0	✔	✔

Table 12.2: A subset of the x86 instructions (part 1)

Instruction	Explanation	Effect	C	S	Z
Extensions					
cbw	Extend byte→word	8-bit %AL is extended to 16-bit %AX			
cwd	Extend word$rightarrow$double	16-bit %AX is extended to 32-bit %DX:%AX			
cwde	Extend double→ext	Extends 16-bit %AX to 32-bit %EAX			
cdq	Extend ext→quad	Extends 32-bit %EAX to 64-bit %EDX:%EAX			
Shifting					
rcl{bwl} $\{c\}$,$\{mr\}$	Left C-rotate	$\{mr\} \leftarrow \langle\{mr\},C\rangle \circlearrowright^{\{c\}}$	✔		
rcr{bwl} $\{c\}$,$\{mr\}$	Right C-rotate	$\{mr\} \leftarrow \langle\{mr\},C\rangle \circlearrowleft^{\{c\}}$	✔		
rol{bwl} $\{c\}$,$\{mr\}$	Left rotate	$\{mr\} \leftarrow \{mr\} \circlearrowright^{\{c\}}$	✔		
ror{bwl} $\{c\}$,$\{mr\}$	Right rotate	$\{mr\} \leftarrow \{mr\} \circlearrowleft^{\{c\}}$	✔		
sal{bwl} $\{c\}$,$\{mr\}$	Left shift	$\{mr\} \leftarrow \{mr\} \overset{\{c\}}{\Leftarrow} 0$	✔	✔	✔
sar{bwl} $\{c\}$,$\{mr\}$	Right arithmetic shift	$\{mr\} \leftarrow S \overset{\{c\}}{\Rightarrow} \{mr\}$	✔	✔	✔
shr{bwl} $\{c\}$,$\{mr\}$	Right logical shift	$\{mr\} \leftarrow 0 \overset{\{c\}}{\Rightarrow} \{mr\}$	✔	✔	✔
Testing					
bt{wl} $\{c\}$,$\{mr\}$	Bit-test	bit $\{c\}$ of $\{mr\}$	✔		
btc{wl} $\{c\}$,$\{mr\}$	Bit-change	bit $\{c\}$ of $\{mr\}$ $\leftarrow\overline{(\text{bit } \{c\} \text{ of } \{mr\})}$	✔		
btr{wl} $\{c\}$,$\{mr\}$	Bit-clear	bit $\{c\}$ of $\{mr\}$ $\leftarrow 0$	✔		
bts{wl} $\{c\}$,$\{mr\}$	Bit-set	bit $\{c\}$ of $\{mr\}$ $\leftarrow 1$	✔		
cmp{bwl} $\{cmr\}_1$,$\{cmr\}_2$	Compare values	$\{cmr\}_2 - \{cmr\}_1$	✔	✔	✔
test{bwl} $\{cmr\}_1$,$\{cmr\}_2$	Test bits	$\{cmr\}_2 \wedge \{cmr\}_1$	✔	✔	✔
Jumps					
call $\{a\}$	Call	push %EIP; %EIP $\leftarrow \{a\}$			
ja $\{a\}$	Jump on unsigned >	if $\bar{Z} \wedge \bar{C}$: %EIP $\leftarrow \{a\}$			
jae $\{a\}$	Jump on unsigned ≥	if $\bar{C}$: %EIP $\leftarrow \{a\}$			
jb $\{a\}$	Jump on unsigned <	if C: %EIP $\leftarrow \{a\}$			
jbe $\{a\}$	Jump on unsigned ≤	if $Z \vee C$: %EIP $\leftarrow \{a\}$			
jc $\{a\}$	Jump on carry	if C: %EIP $\leftarrow \{a\}$			
je $\{a\}$	Jump on =	if Z: %EIP $\leftarrow \{a\}$			
jmp $\{a\}$	Jump	%EIP $\leftarrow \{a\}$			
jg $\{a\}$	Jump on >	if $\bar{Z} \wedge S = O$: %EIP $\leftarrow \{a\}$			
jge $\{a\}$	Jump on ≥	if $S = O$: %EIP $\leftarrow \{a\}$			
jl $\{a\}$	Jump on <	if $S \neq O$: %EIP $\leftarrow \{a\}$			
jle $\{a\}$	Jump on ≤	if $Z \vee S \neq O$: %EIP $\leftarrow \{a\}$			
jnc $\{a\}$	Jump on non-carry	if $\bar{C}$: %EIP $\leftarrow \{a\}$			
jne $\{a\}$	Jump on ≠	if $\bar{Z}$: %EIP $\leftarrow \{a\}$			
jns $\{a\}$	Jump on non-negative	if $\bar{S}$: %EIP $\leftarrow \{a\}$			
jnz $\{a\}$	Jump on non-zero	if $\bar{Z}$: %EIP $\leftarrow \{a\}$			
js $\{a\}$	Jump on negative	if S: %EIP $\leftarrow \{a\}$			
jz $\{a\}$	Jump on zero	if Z: %EIP $\leftarrow \{a\}$			
loop $\{a\}$	Loop	%ECX $\leftarrow$%ECX-1; if %ECX $\neq$ 0: %EIP $\leftarrow \{a\}$			
ret	Return	%EIP $\leftarrow$ pop			
Miscellaneous					
rdtsc	Fetch cycles	%EDX:%EAX $\leftarrow$ $\langle$number of cycles$\rangle$			

Table 12.3: A subset of the x86 instructions (part 2)

Instruction	Explanation	Effect	*C*	*S*	*Z*
Load					
fld1	Float load 1	Push 1.0			
fildl {*m*}	Float int load long	Push long {*m*}			
fildq {*m*}	Float int load quad	Push long long {*m*}			
filds {*m*}	Float int load short	Push short {*m*}			
fldl {*m*}	Float load long	Push double {*m*}			
flds {*m*}	Float load short	Push float {*m*}			
fldz	Float load zero	Push 0.0			
Store					
fistl {*m*}	Float int store long	Store %ST(0) in long {*m*}			
fistpl {*m*}	Float int store and pop long	Pop %ST(0) into long {*m*}			
fistpq {*m*}	Float int store and pop quad	Pop %ST(0) into long long {*m*}			
fistq {*m*}	Float int store quad	Store %ST(0) in long long {*m*}			
fistps {*m*}	Float int store and pop short	Pop %ST(0) into short {*m*}			
fists {*m*}	Float int store short	Store %ST(0) in short {*m*}			
fstl {*m*}	Float store long	Store %ST(0) in double {*m*}			
fstpl {*m*}	Float store and pop long	Pop %ST(0) into double {*m*}			
fstps {*m*}	Float store and pop short	Pop %ST(0) into float {*m*}			
fsts {*m*}	Float store short	Store %ST(0) in float {*m*}			
Arithmetic					
fabs	Float absolute	%ST(0) ←|%ST(0)|			
fadd %ST(*X*)	Float add	%ST(0) ← %ST(0) + %ST(*X*)			
fadd{*sl*} {*m*}	Float add	%ST(0) ← %ST(0)+float/double {*m*}			
faddp {*m*}	Float add and pop	%ST(1) ← %ST(0) + %ST(1); pop			
fchs	Float change sign	%ST(0) ←−%ST(0)			
fdiv %ST(*X*)	Float div	%ST(0) ← %ST(0) ÷ %ST(*X*)			
fdiv{*sl*} {*m*}	Float div	%ST(0) ← %ST(0)÷float/double {*m*}			
fdivp {*m*}	Float reverse div and pop	%ST(1) ← %ST(0) ÷ %ST(1); pop			
fdivrp {*m*}	Float div and pop	%ST(1) ← %ST(1) ÷ %ST(0); pop			
fiadd{*sl*} {*m*}	Float int add	%ST(0) ← %ST(0)+short/long {*m*}			
fidiv{*sl*} {*m*}	Float int div	%ST(0) ← %ST(0)÷short/long {*m*}			
fimul{*sl*} {*m*}	Float int mul	%ST(0) ← %ST(0)×short/long {*m*}			
fisub{*sl*} {*m*}	Float int sub	%ST(0) ← %ST(0)−short/long {*m*}			
fmul %ST(*X*)	Float mul	%ST(0) ← %ST(0) × %ST(*X*)			
fmul{*sl*} {*m*}	Float mul	%ST(0) ← %ST(0)×float/double {*m*}			
fmulp {*m*}	Float mul and pop	%ST(1) ← %ST(0) × %ST(1); pop			
fsqrt	Float square root	%ST(0) ←$\sqrt{\text{\%ST(0)}}$			
fsub %ST(*X*)	Float sub	%ST(0) ← %ST(0) − %ST(*X*)			
fsub{*sl*} {*m*}	Float sub	%ST(0) ← %ST(0)−float/double {*m*}			
fsubp {*m*}	Float reverse sub and pop	%ST(1) ← %ST(0) − %ST(1); pop			
fsubrp {*m*}	Float sub and pop	%ST(1) ← %ST(1) − %ST(0); pop			
fyl2xpl	Float ???	%ST(1) ←%ST(1) × $\log_2$(%ST(0) + 1); pop			
Stack operations					
fld %ST*X*	Float load	Push copy of %ST(*X*)			
fst %ST*X*	Float store	Store copy of %ST(0) in %ST(*X*)			
fstp %ST*X*	Float store and pop	Pop %ST(0) into %ST(*X*)			

Table 12.4: A subset of the x87 floating-point instructions

Appendix A
Questions Catalogue

A.1 Introduction to Digital Electronics

1) What is ‘digital’ electronics?
2) What does ‘binary’ mean?
3) What is Moore’s law?
4) What is the basic building block of digital electronics today?

A.2 Binary Numbers

1) What are binary, decimal and hexadecimal numbers?
2) How are signed numbers encoded as binary numbers? Variants?
3) Explain the two’s complement encoding!
4) name some properties of two’s complement representation!
5) how do you invert a two’s complement encoded number?
6) how do you subtract in two’s complement? Pitfalls?
7) how do you multiply/divide by 2^n in two’s complement?
8) What is an arithmetic right shift?
9) How do you execute a general multiplication of two two’s complement numbers a and b?

A.3 Boolean Algebra

1) What is a Boolean function?
2) How can you describe a Boolean function?
3) Describbe the deMorgan’s theorem!
4) Can you list logic gates and their corresponding Boolean function?
5) Can you show what it means thet the AND and OR opperators are commutative, associative and distributive?
6) Can you simplify an easy Boolean function, e.g., $a \wedge (\overline{b} \oplus c) \vee \overline{a}b \wedge c$
7) How do you set up a Karnaugh map and how do you use it to simplyfy a Boolean expression?
8) How doeas a Karnaugh map look like that will still result in a very long and complicated expression.

9) How do you use a NAND/NOR gate to implement the basic Boolean operators?

10)

A.4 Combinational Logic Crcuits

1) Can you define combinational logic circuits?

2) Can you draw a digital circuit with inverters, AND and OR gates that implement the XNOR function?

3) What's the function of an encoder/decoder, multiplexer/demultiplexer?

4) Can you draw and explain the function of a full-adder?

A.5 Sequential Logic Crcuits

1) What is a flip-flop?

2) What distinguishes a synchronous from an asynchronous flip-flop/latch?

3) What is the characteristic table/function of a D-latch, SR-latch, JK-flip-flop D-flip-flop, T-flip-flop, ...?

4) What is the clock frequency of a clock with a period of 1ns?

5) How do you make a T-flip-flop/D-flip-flop from a JK-flip-flop and a minimal number of logic gates?

6) How do you make a JK-flip-flop from a D-flip-flop and a minimal number of logic gates?

7) What's a state transition graph, what's a finite state machine?

8) Can you draw a state transition graph for a hysteretic controller of an automatic blind that closes if the sun has been out for three consequtive clock cycles and that opens if the sun has been away for three consequtive clock cycles?

9) What's the difference of a Mealy and a Moore type FSM?

10) What are registers?

11) Can you draw and explain a synchronous counter/ripple counter/shift register?

A.6 Von Neumann Architecture

1) What are the main units of a computer according to the Von Neumann reference model?

2) What are possible subcomponents of the execution unit/control unit (memory unit/I/O unit)?

3) What is the Von Neumann bottleneck?

4) What does the abbreviation ALU stand for and what's its task?

5) can you describe the simple 1-bit ALU as depicted in the script?

6) What does DRAM and SRAM stand for, what is their task and what are their differences?
7) can you draw and explain an SRAM/DRAM cell?
8) What is a tri-state buffer?
9) What is a sense-amplifier?
10) What is the task of the control unit (CU)?
11) What is machine code?
12) What is a microarchitecture/microcode?
13) What is a CISC/RISC?
14) What is the task of an I/O controller?
15) What is memory mapped/isolated I/O?
16) What is programmed/polled I/O, interrupt driven I/O and direct memory access?

A.7 Optimizing Hardware Performance

1) What is a memory hierarchy and why is there a hierarchy?
2) What is cache and what is its task?
3) Why does cache work in many cases?
4) What is associateive-, direct mapped- and set-associative cache?
5) How is cache coherency a potential problem, how is it solved?
6) How and when are blocks in the (associative/direct mapped) cache replaced? What's the respective advantages and disadvantages of different strategies?
7) What are look-through and look-aside cache architectures?
8) Explain virtual memory!
9) Depict a memory management unit (managing virtual memory)
10) Explain the principle of pipelining!
11) ...

Index