

Sergiyenko A.M., Korneychuk V.I.

# Digital Networks Design



**УДК 681.3**

**ББК 32.973-01**

**С 34**

Рецензенти: В.П. Тарасенко, доктор технічних наук, професор, завідувачий кафедрою спеціалізованих комп'ютерних систем НТУУ "КПІ"; О.В.Бузовський, доктор технічних наук, професор, кафедра обчислювальної техніки НТУУ "КПІ".

**С34 Сергієнко А.М., Корнійчук В.І.**

**Digital Networks Design (Комп'ютерна схемотехніка).**

**– К.: «Корнійчук», 2007. –80 с.**

**ISBN 966-7599-41-8**

Розглянуті способи побудови типових вузлів, блоків та пристроїв комп'ютерів від найпростіших логічних схем до ядра мікропроцесора. Приведено опис типових вузлів на мові VHDL. Для студентів, аспірантів, викладачів вузів з викладанням на англійській мові, а також для читання спеціалістами в галузі електроніки, вимірювальної і обчислювальної техніки та зв'язку.

**ББК 32.973-01**

**С34 Sergiyenko A.M., Korneychuk V.I.**

**Digital Networks Design – K.: «Корнійчук», 2007. –80 p.**

**ISBN 966-7599-41-8**

Methods of design of base modules and devices for computers are considered, beginning at simplest logic networks and finishing at microprocessor cores. The base modules are described by the VHDL language. For the students, aspirants, high school lecturers, as well as for reading by the specialists at the fields of electronic engineering, measurements, communications and computer engineering.

**ББК 32.973-01**

**ISBN 966-7599-41-8**

**© Сергієнко А.М., Корнейчук В.И., 2007**

Sergiyenko A.M., Korneychuk V.I.

# Digital Networks Design

Сергієнко А.М., Корнійчук  
В.І.Комп'ютерна схемотехніка  
(на англійській мові)

Київ  
2007

# Contents

|   |    |
|---|----|
| 1. Logic network basics                         | 3  |
| 1. Integral circuits                            | 3  |
| 2. Boolean algebra                              | 7  |
| 3. Combinational logic networks                 | 11 |
| 4. Triggers                                     | 14 |
| 5. Decoders                                     | 17 |
| 6. Multiplexers                                 | 19 |
| 7. Encoders                                     | 20 |
| 8. Shifters                                     | 21 |
| 9. Binary adders                                | 21 |
| 10. Registers                                   | 23 |
| 11. Counters                                    | 25 |
| 12. Programmable logic devices                  | 27 |
| 2. Memory units                                 | 34 |
| 1. General properties                           | 34 |
| 2. Fast memory units                            | 35 |
| 3. Register file                                | 36 |
| 4. Stack memory                                 | 37 |
| 5. Cache memory                                 | 37 |
| 6. Memory integral circuits                     | 39 |
| 3. Networks for arithmetic and logic operations | 43 |
| 1. Arithmetic and logic units                   | 43 |
| 2. Datapath                                     | 45 |
| 3. Binary multipliers                           | 47 |
| 4. Binary dividers                              | 50 |
| 5. Hardware pipelining                          | 51 |
| 4. Control networks                             | 53 |
| 1. CPU control unit                             | 53 |
| 2. CPU instruction set                          | 54 |
| 3. Control unit structure                       | 55 |
| 4. Instruction fetch networks                   | 56 |
| 5. Finite state machines                        | 58 |
| 6. Microprogram controllers                     | 62 |
| 7. Example of CPU design                        | 64 |
| 5. Interfaces                                   | 69 |
| 1 Common busses                                 | 69 |
| 2 AMBA interface in SOC design                  | 71 |
| List of abbreviations                           | 78 |
| Bibliography                                    | 79 |

# 1. Logic network basics

## 1.1 Integral circuits

Integral circuits (ICs) form the basis both of the electronic industry and of all its product applications. The transistor is the atom element of ICs. A single two input logic element (LE), named a **logic gate**, can be formed by three or four **complementary metal-oxide-semiconductor** (CMOS) transistors. The digital IC volume is usually measured by the number of equivalent two input logic gates. Due to this number, all the ICs are divided to small scale integration ICs and to **large scale integration** (LSI) ICs. The small scale integration circuits have the volume of up to hundreds of gates. The standard logic ICs, **programmable logic arrays** (PLAs) and different buffer ICs belong to these circuits.

A set of LSI circuits consists of microprocessors, microcontrollers, memory ICs, **application specific integral circuits** (ASICs), and **application specific standard products** (ASSPs). The modern LSI circuits can contain up to tenths millions of gates. Besides, memory ICs have the volume of up to billions of bits.

In the seventies, the most popular logic ICs were circuits of the series SN74, which consisted of up to thousand of different transistor-transistor logic (TTL) circuits. In USSR the analogous ICs were circuits of the series K155. Some analogous representatives of this series are widely used now, but they are implemented by the modern CMOS transistor technology. In most cases, these ICs are buffers, registered buffers, multiplexers, invertors, simplest logic networks (LNs).

PLAs were designed at the end of seventies on the base of the **read only memory** (ROM) technology. The PLA consists of a set of logic cells. Each of them is a multi-input logic element with a trigger on its output. The PLA programming means the forming bridges between the data sources and the logic element inputs. In PLAs the metal-nitrogenium-oxid-semiconductor (MNOS) transistors play the role of bridges, and they are programmed as the similar bridges in the flash memory. Usually the number of logic cells and the data source number do not exceed ten and fifty, respectively. Now PLAs are widely used as, so called, glue logic circuits, because they "glue" the LSI circuits together in the system.

The **microprocessor** is the main operational unit of the computer. Its functionality is undefined not only by its production but also during its use. It depends on user programs and operational systems. The **microcontroller** has the similar properties as the microprocessor has, but its functionality is usually fixed strictly in the user device where it is built in. This means that it performs a single program, which is usually not exchanged during all its living time. The **digital signal processing** (DSP) processors form a separate subset of microcontrollers. Microprocessors are described in the 4-th chapter of this book.

The memory ICs are divided into **random access memories** (RAMs) and ROMs. These ICs are described in the 2-nd chapter.

ASIC has its name because of its functionality which is fixed during the design and manufacturing processes. Therefore, ASIC implements a single but complex function. ASIC examples are modem circuits, hard disc controllers, parts of the computer chipset. Because of the increase of circuit design cost, the ASIC manu-

facturing is worth of the profit when their stocks have more than million of chips.

ASSPs form a wide set of different devices. Their functionality is less than one of microcontrollers but it is enough to be adapted to a set of different applications. They are application specific memory ICs, like graphic adapter RAM, flash ROM. The microcontrollers with the specific set of peripheral units, for example, MP3 player, MPEG encoder, belong to the ASSP set as well. The system which contains microprocessor, application specific processors, memory units, peripheral units, etc., coupled in a single chip, is named as the **system-on-the-chip** (SOC). From the designer point of view, the ASSP is an ASIC which is developed to satisfy the adaptation possibilities. **Complex programmable logic devices** (CPLDs) and **field programmable gate arrays** (FPGAs) form the specific subsets of ASSPs.

The CPLD structure consists of up to tenths of PLAs placed in a single chip, and connected through a programmable switch array. Its logic volume is usually less than ten thousands of equivalent gates (i.e. two input logic elements).

The FPGA was invented in eighties as the alternative to the CPLD. The FPGA represents the array of 2-6 input logic elements, triggers (registers), and wire parts, which are connected together by a set of bridges. These bridges are formed by the field effect transistors (FETs), controlled by the special programming triggers. The routes of the FPGA netlist are programmed by the exchange of the electric field in the FET gates, and this is the root of the FPGA name. Before the FPGA operation, the programming bit stream, named **configuration**, is automatically loaded into FPGA from the outer ROM. This process is named as FPGA configuring. Modern FPGAs contain RAM units, hardware multipliers, fast speed interfaces, microprocessor cores and other units. Their logic volume reaches ten millions of equivalent gates. Sometimes FPGA is a part of another complex ASSP.

The CPLD and FPGA project designing is cost effective and has the small design period. Therefore, it is the alternative to ASIC when the series of production of the specific device does not succeed hundred thousands of units. As a result, the number of new FPGA and CPLD projects increases, and the number of ASIC projects decreases every year.

Due to the standards, accepted by countries of former USSR, ICs and their parts are drawn in the schematic diagrams as rectangles, named the **network symbols**. This symbol has three fields as is shown in Fig.1.1 (a). The left field signs the input marks (xx, xy), the right field signs the marks of outputs (xz), and the IC function name is placed in the middle (xxx). In the symbol of the simple IC the left and/or right field can be absent. Due to standards of western countries, the symbol fields are not separated by lines, and additional input-output fields can be placed in upper and bottom sides of the symbol.

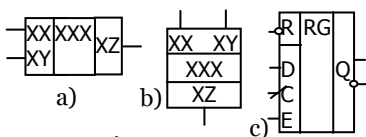


Fig.1.1

If the IC pin serves both as input and as output, then its name can be placed in the left or right field. In such situation the symbol <> (bidirectional) is attached to this name. When it is needed, the IC symbol can be rotated clockwise to 90° (Fig.1.1 (b)).

Some name characters have the common meaning for many ICs, and they are described in the Table 1.1. Fig.1.1(c) illustrates the register with inverse reset input, direct and inverse output, and loading by the rising edge of the clock signal when the enable signal is high.

Table 1.1

|                     |  |                |   |
|---------------------|--|----------------|---|
| A                   | - address bus                              | G              | - generator   |
| ALU                 | - arithmetic-logic unit                    | MUX            | - multiplexor   |
| B                   | - unidirectional or bidirectional bus      | MPU            | - multiplier unit   |
| C                   | - clock input                              | Mn             | - modulo n arithmetic unit                                      |
| CI,CO               | - carry input, carry output                | Q              | - data output, can be bidirectional                             |
| CD                  | - coder                                    | $\bar{Q}$ , nQ | - negated data output   |
| CPU                 | - central processing unit                  | R              | - reset input   |
| CT,CTn              | - counter, counter modulo n                | RG             | - register  |
| D                   | - data input, can be bidirectional         | S              | - set input   |
| DC                  | - decoder                                  | SHU            | - shifter unit  |
| E                   | - enable input                             | SM             | - summator, adder, subtractor                                   |
| F                   | - control input, for function coding       | T              | - latch, 1-bit latch register                                   |
| GND                 | - signal of the logic '0' (ground)         | TT             | - trigger, flip-flop, 1-bit register                            |
| o                   | - negated input or output (see Fig.1.1(c)) | X/Y            | - logic network, transfers X to Y                               |
| /, $\triangleright$ | - rising edge clock input                  | 1              | - repeater, OR function (Fig.1.3(a))                            |
| $\nabla$            | - falling edge clock input                 | &              | - AND function (Fig.1.3(c))                                     |
| $\diamond$          | - mark of open emitter (drain) output      | ==             | - equality function   |
| $\diamond$          | - mark of open collector (source) output   | =1, $\oplus$   | - Exclusive OR function   |
| $\diamond$          | - tristate output                          | #              | - digital function  |
| $\times$            | - not logical input or output              | <n>            | - mark of the grouped component (component is repeated n times) |

**Logic elements** (LEs) form a **component basis** of ICs, which is proper to the given IC technology. The IC is designed on the base of the component library, which is formed by the LEs of different kinds, with different input number, delay  $t$ , power consumption and space on the chip surface. The quality, consumer properties of ICs depend on its technology, component basis, delay  $t$ , maximum clock period  $t_c$ , supply voltage  $V$ , power consumption  $P$ , logic level voltages  $L_0$  and  $L_1$ , its output buffer loading characteristics, and others parameters.

In the logic electric circuits bits a 0 and a 1 are represented by two **logic voltage levels**  $V_0$  and  $V_1$ . Sometimes they are represented by low and high current level, or positive and negative current. In any case, the low level is marked as L, and high level – as H. When  $V_0 = L$  and  $V_1 = H$  the IC is named as one with the **positive logic**, and when  $V_0 = H$  and  $V_1 = L$  then it is the **negative logic** IC. Below we will consider the positive logic ICs. The working voltage range (usually from 0 to  $V$ ) is divided into three ranges:  $V-H_t$ ,  $L_t-0$ , and  $H_t-L_t$ , which are illustrated by Fig.1.2. If the signal magnitude is in the range  $V-H_t$  ( $L_t-0$ ) then LE

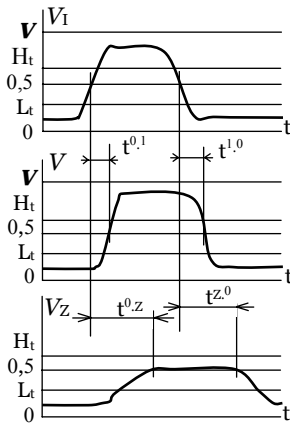


Fig.1.2

accepts this signal as the bit 1 (0). And if the signal magnitude stays in the threshold  $H_t$ – $L_t$  then LE can recognize it both as 0 and as 1, and LE operates unstable. Therefore, such signal magnitude is allowed only in the moment of LE switching.

Some buffer circuits have the operation mode in which their outputs have the high impedance. Then the signal on their outputs is allowed to be in the threshold  $H_t$ – $L_t$  for a long time, because this operation is provided to build common busses. For this reason the state, when signal is in range  $H_t$ – $L_t$ , is named as **third state** (Z-state), and these buffers and busses are named as **tristate** ones.

Modern ICs often have two or more **supply voltages**. One of them is usually equal to  $V=3,3$  volts, and is used for feeding the input and output buffers. Another voltages are much less (down to

$V=1$  volt), and are used to supply the inner circuits. Thus, inner and outer voltage logic levels are different ones. The outer voltage logic levels in most of cases obey the standard TTL logic levels, i.e. level  $H_t$  is 2,4 volts, and level  $L_t$  is 0,8 volts.

The IC speed is derived from the **propagation delays**  $t$  of its LEs and other components. In general, these delays depend on the route of the signal propagation, and on the capacitance at the LE outputs. This capacitance in CMOS circuits is proportional to the number of LE inputs, which are attached to this LE output. This number is named as the LE **fanout**.

Sometimes the delay  $t^{0,1}$  of propagation of transition from L to H is different from the delay  $t^{1,0}$  for transition from H to L. As a rule, these delays are shorter than propagation delay of generating the transition from Z to H or from Z to L and visa versa at the LE output (see the **waveform**  $V_z$  in Fig.1.2). Besides, all the delays become shorter with temperature and power voltage increase. As a result, each LE has the delay function which depends on the fanout, transition form, voltage and temperature. In the **computer-aided design** (CAD)-tools the LE libraries usually contain such delay functions, which are taken into account during the logic synthesis. In the modern ICs, the propagation delays in wires can be higher then delays of LE switching. Therefore, by the design of such ICs the capacitance and inductance of the wires, which cause the delay, are taken into account as well. In simple calculations the LE delay can be considered as that that is equal to a constant for all the similar LEs, and is equal to the maximum delay over all possible delays.

The **power consumption** of modern ICs is caused, in general, by the switching processes in them. Some power consumption is forced by the current leakage, but it is much less for CMOS circuits. The **switching** is the effect of charging and discharging of the LE capacitances, and it can be estimated by the formula:



$$P = C_L V^2 E_S f_C / 2, \quad (1)$$

where  $C_L$  is the physical capacitance at the LE output,  $E_S$  is the average number of output transitions per clock cycle (the switching activity, it is typically 20% in most designs), and  $f_C$  is the clock frequency. Due to the formula (1), reduction of any of its factors will result in a lower power consumption of the IC. The reduction of the supply voltage  $V$  is the most attractive, because it is in a quadratic relation to power. But this has a negative impact on the speed of the design because the voltage reduction increases the delay  $t$  of LEs.  $C_L$  is the LE loading, which is proportional to LE fanout and wire lengths.  $C_L$  decreasing is more effective because it decreases the LE delay, and thus increases the design throughput.  $f_C$  is derived from the maximum delay in the chains of LEs which route the logic signals from the source trigger to the destination trigger, and therefore, it increases by the minimizing such chains.

The IC design strategy is directed to minimize the LE number, its fanout, its input number, and its number in logic chains, considering the given component library. Therefore, the IC logic synthesis is complex task with respect to a set of contradiction goals like hardware and power minimization, and speed maximization. This task is implemented now automatically in many modern CAD-systems. But to achieve success, the designer has to know excellently the rules and laws of the logic synthesis to be able to direct this process. Moreover, often the excellent IC projects are designed by hand, because the automatic results occur to be bad. In the following chapters we will look into the logic design processes.

## 1.2 Boolean algebra

The LE is the circuit which operation can be described by the simple combinational logic function or the **Boolean function** (BF). This function can have only two meanings, or significances: 0 and 1 or *false* and *true*. The BF arguments also have only two such meanings.

The simplest BF  $Y=f(X)$  is given by its significances by  $X=0$  and  $X=1$ . In general, there are four such functions, which are given in the **truth Table 1.2**. BFs  $f_0$  and  $f_3$  are constants 0 and 1. Significances of  $f_1$  are equal to  $X$ , therefore LE which implements  $f_1$  is named as **buffer**. Its graphical symbol is illustrated by Fig.1.3 (a). Here in the left side is the symbol due to former Soviet Union countries standards, or to IEEE standard.

In the right side is the symbol, which is adopted in western countries. BF  $f_2$  exchanges 0 to 1, and 1 to 0. Such transform is named as **inversion**, marked as  $\bar{X}$ , and spelled as not X. LE which implements  $\bar{X}$  is named as an **invertor**, or a NOT gate (see Fig.1.3 (b)). To define the two argument BF  $f(X_1, X_2)$  one has to give its significances on four argument sets  $(X_1, X_2)$ , which is illustrated by the Table 1.3. Such a task can be implemented by one of 16 ways. And 16 different

Table 1.2.

| $X$ | $f_0$ | $f_1$ | $f_2$ | $f_3$ |
|-----|-------|-------|-------|-------|
| 0   | 0     | 0     | 1     | 1     |
| 1   | 0     | 1     | 0     | 1     |

Table 1.3

| $X_1$ | $X_2$ | $X_1 \cdot X_2$ | $X_1 \vee X_2$ | $=1$ |
|-------|-------|-----------------|----------------|------|
| 0     | 0     | 0               | 0              | 0    |
| 0     | 1     | 0               | 1              | 1    |
| 1     | 0     | 0               | 1              | 1    |
| 1     | 1     | 1               | 1              | 0    |

BFs can be distinguished. But practically, the following six BFs are used:  $X_1 \cdot X_2$  (AND function, or **conjunction** or '&'),  $X_1 \vee X_2$  (OR, **disjunction**, or '1'), **Exclusive OR** (shortly EXOR, or '=1'), and their negations:  $\overline{X_1 \cdot X_2}$  (Not AND, shortly NAND),  $\overline{X_1 \vee X_2}$  (Not OR, shortly NOR) and Exclusive Not OR (shortly XNOR, or '=' that means equality). LEs or gates, which implement these BFs, have the proper names (see Fig.1.3(c-h)).

In most of cases of analysis or synthesis of logic networks the functions AND, OR, and NOT are used. These functions form, so called, **Boolean algebra**. Besides, AND, OR functions can have large number of arguments. From the Table 1.3, one can derive the following Boolean algebra equalities and identities:

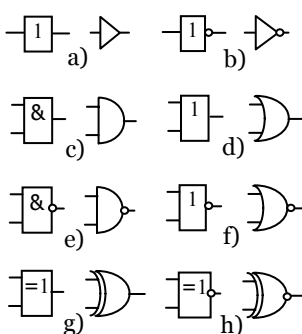


Fig.1.3

$$\begin{aligned}
 X \vee 0 &= X, & X \vee 1 &= 1, & X \vee X &= X, \\
 X \cdot 0 &= 0, & X \cdot 1 &= X, & X \cdot X &= X, \\
 X \vee Y &= Y \vee X, & X \cdot Y &= Y \cdot X, \\
 (X \vee Y) \vee Z &= X \vee (Y \vee Z), & (X \cdot Y) \cdot Z &= X \cdot (Y \cdot Z), \\
 X \vee \overline{X} \cdot Y &= X \vee Y, & X \cdot \overline{Y} \vee X \cdot Y &= X, \\
 X \cdot (Y \vee Z) &= (X \cdot Y) \vee (X \cdot Z), & \overline{X \vee Y} &= \overline{X} \cdot \overline{Y}, \\
 X &= \overline{\overline{X}}, & X \vee \overline{X} &= 1, & X \cdot \overline{X} &= 0, & \overline{0} &= 1, & \overline{1} &= 0, \\
 \overline{X \cdot Y} &= \overline{X} \vee \overline{Y}, & X \vee Y &= \overline{\overline{X} \cdot \overline{Y}}.
 \end{aligned}
 \tag{2}$$

Here and below the higher priority of AND (point) operation is considered.

The number of different BFs is derived from the argument number  $n$ , and is equal to  $2^{2^n}$ , where  $2^n$  is the number of different argument sets. When  $n=3$  we can get 256 BFs. But it is not necessary to build a set of 256 LEs to select from it the needed function of 3 arguments. It is enough to have a set of gates of AND, OR, NOT-type. The fact is that any BF is represented by the superposition of these functions using the following equation, named as a **sum-of-product form**:

$$f(X_1, X_2, \dots, X_n) = \bigvee f(\alpha_1, \alpha_2, \dots, \alpha_n) \cdot X_1^{\alpha_1} \cdot X_2^{\alpha_2} \dots X_n^{\alpha_n}, \tag{3}$$

where the OR function is given on all the sets  $(\alpha_1, \alpha_2, \dots, \alpha_n)$ , and

$$X_i^{\alpha_i} = \begin{cases} X_i, & \text{when } \alpha_i = 1; \\ \overline{X_i}, & \text{when } \alpha_i = 0; \end{cases} \quad i=(1, \dots, n).$$

Really,

$$X^\alpha = \begin{cases} 1 & \text{when } X = \alpha; \\ 0 & \text{when } X \neq \alpha. \end{cases}$$

As a result, the function, named the term, is equal to  $X_1^{\alpha_1} \cdot X_2^{\alpha_2} \dots X_n^{\alpha_n} = 1$  only if  $X_i = \alpha_i$  for all values of  $i$ . And in this situation

$$f(\alpha_1, \alpha_2, \dots, \alpha_n) = 0 \vee 0 \dots f(\alpha_1, \alpha_2, \dots, \alpha_n) \cdot 1 \vee 0 \dots 0 \vee 0,$$

i.e. the left part of the equation (3) is equal to the right one.

Consider an example. There are three lighting switches in the room. The goal is to design the logic network which provides switching on and off by a single switch not to touch the other switches. Consider one state of the switch is zero (0), and another one is one (1). Because there are three switches, the network must implement the logic function of three arguments. Let lighting is off when all the switches are in the state 0, i.e. the switch state is 000. Then a single exchange of any switch forces lighting on. Therefore, BF has to be equal to 1 on the set of states 001, 010, and 100. Any exchange of these states forces lighting off. Finally, when switches are in the states 011, 101, 110 then BF has to be a 0, i.e. the light is off. The next exchange of any switch makes lighting on, which gives  $f(1,1,1)=1$ . The meanings of the BF  $f$  are shown in the Table 1.4.

Then we represent the derived BF in the form (3):

$$\begin{aligned} f(X_1, X_2, X_3) &= f(0,0,0) \cdot X_1^0 X_2^0 X_3^0 \vee f(0,0,1) \cdot X_1^0 X_2^0 X_3^1 \vee f(0,1,0) \cdot X_1^0 X_2^1 X_3^0 \vee f(0,1,1) \cdot X_1^0 X_2^1 X_3^1 \vee \\ &\vee f(1,0,0) \cdot X_1^1 X_2^0 X_3^0 \vee f(1,0,1) \cdot X_1^1 X_2^0 X_3^1 \vee f(0,0,0) \cdot X_1^0 X_2^0 X_3^0 \vee f(1,1,1) \cdot X_1^1 X_2^1 X_3^1 = \\ &= 0 \cdot \bar{X}_1 \bar{X}_2 \bar{X}_3 \vee 1 \cdot \bar{X}_1 \bar{X}_2 X_3 \vee 1 \cdot \bar{X}_1 X_2 \bar{X}_3 \vee 0 \cdot \bar{X}_1 X_2 X_3 \vee 1 \cdot X_1 \bar{X}_2 \bar{X}_3 \vee 0 \cdot X_1 \bar{X}_2 X_3 \vee \\ &\vee 0 \cdot X_1 X_2 \bar{X}_3 \vee 1 \cdot X_1 X_2 X_3 = \bar{X}_1 \cdot \bar{X}_2 \cdot X_3 \vee \bar{X}_1 \cdot X_2 \cdot \bar{X}_3 \vee X_1 \cdot \bar{X}_2 \cdot \bar{X}_3 \vee X_1 \cdot X_2 \cdot X_3. \end{aligned}$$

The network, which implements this BF, is shown in Fig.1.4 (a). It consists of three NOT gates for negating the input signals, four AND gates, and a single OR gate. This network can be simplified by the use of the Exclusive OR gates. The analysis of the Table 1.4 shows that this BF is the sum modulo 2 of input variables. Therefore, this function can be implemented by two 2-input gates, as it is shown in Fig.1.4 (b).

The sum-of-product form (3), named as AND/OR form, is not unique method of deriving BFs. Really, using the relations  $X = \bar{\bar{X}}$ , and  $X \vee Y = \bar{X} \cdot \bar{Y}$  for the previous example, we can get the following forms

$$\begin{aligned} f(X_1, X_2, X_3) &= \bar{X}_1 \bar{X}_2 X_3 \cdot \bar{X}_1 X_2 \bar{X}_3 \cdot X_1 \bar{X}_2 \bar{X}_3 \cdot X_1 X_2 X_3 = \quad (\text{AND-NOT/AND-NOT}) \\ &= \bar{X}_1 \vee X_2 \vee \bar{X}_3 \cdot \bar{X}_1 \vee \bar{X}_2 \vee X_3 \cdot \bar{X}_1 \vee X_2 \vee X_3 \cdot \bar{X}_1 \vee \bar{X}_2 \vee \bar{X}_3 = \quad (\text{OR/AND-NOT}) \quad (4) \\ &= \bar{X}_1 \vee X_2 \vee \bar{X}_3 \vee \bar{X}_1 \vee \bar{X}_2 \vee X_3 \vee \bar{X}_1 \vee X_2 \vee X_3 \vee \bar{X}_1 \vee \bar{X}_2 \vee \bar{X}_3, \quad (\text{OR-NOT/OR}) \end{aligned}$$

Both the function and its inversion can be represented by the form AND/OR. This gives another four forms:

Table 1.4

| $X_1$ | $X_2$ | $X_3$ | $f$ |
|-------|-------|-------|-----|
| 0     | 0     | 0     | 0   |
| 0     | 0     | 1     | 1   |
| 0     | 1     | 0     | 1   |
| 0     | 1     | 1     | 0   |
| 1     | 0     | 0     | 1   |
| 1     | 0     | 1     | 0   |
| 1     | 1     | 0     | 0   |
| 1     | 1     | 1     | 1   |

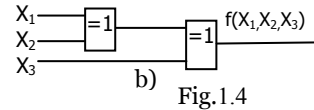
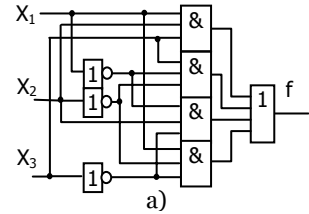


Fig.1.4

$$\begin{aligned}
f(X_1, X_2, X_3) &= \overline{\overline{X_1} \overline{X_2} \overline{X_3}} \vee \overline{\overline{X_1} X_2 X_3} \vee \overline{X_1 \overline{X_2} X_3} \vee \overline{X_1 X_2 \overline{X_3}} = & (\text{AND/OR-NOT}) \\
&= \overline{\overline{X_1} \overline{X_2} \overline{X_3}} \cdot \overline{\overline{X_1} X_2 X_3} \cdot \overline{X_1 \overline{X_2} X_3} \cdot \overline{X_1 X_2 \overline{X_3}} = & (\text{AND-NOT/AND}) \\
&= (X_1 \vee X_2 \vee X_3) \cdot (X_1 \vee \overline{X_2} \vee \overline{X_3}) \cdot (\overline{X_1} \vee X_2 \vee \overline{X_3}) \cdot (\overline{X_1} \vee \overline{X_2} \vee X_3) = & (\text{OR/AND}) \quad (5) \\
&= \overline{\overline{X_1} \vee \overline{X_2} \vee \overline{X_3}} \cdot \overline{X_1 \vee \overline{X_2} \vee \overline{X_3}} \cdot \overline{\overline{X_1} \vee X_2 \vee \overline{X_3}} \cdot \overline{\overline{X_1} \vee \overline{X_2} \vee X_3} = & (\text{OR-NOT/OR-NOT})
\end{aligned}$$

Each of the relations (3), (4), (5) is called as the **normal form** of the BF representation. They can be useful by designing of LNs, based on the concrete gate library.

Many transformations of BF can be usefully interpreted when BF is graphically represented. In the geometrical sense a set  $(X_1^{a_1}, X_2^{a_2}, \dots, X_n^{a_n})$  can be represented by the vector, which forms a point in the  $n$ -dimensional space. All  $2^n$  combinations of vectors form the nodes of the  $n$ -dimensional cube. Marking the nodes, where BF is equal to 1, we derive the graphical representation of BF. These marked nodes represent the terms of the equation (3). In Fig.1.5 (a) the 3-dimensional cube of the function (4) is drawn. The number of variables  $X_i$ , which exchange its coordinate when traversing from one node to another one, is named as the distance between these nodes. Looking at Fig.1.5, one can prove that the distance between all couples of the nodes is equal to 2. A single node of the  $n$ -dimensional

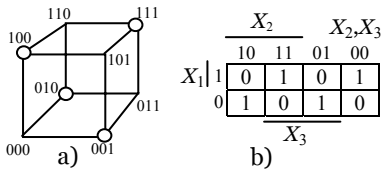


Fig.1.5

cube, which BF is equal to one, is named as a **0-cube**. Two 0-cubes, which are connected by an edge, form the 1-cube. This means that in the 1-cube the distance between two nodes is equal to a 1, and two respective terms are different in a single variable. Four nodes, which form the square plane, belong to the 2-cube, and visa versa. Often the cubes are named as **prime implicants**, because they represent the OR function of terms (implication), which can be reduced.

BFs are often represented graphically by **Karnaugh maps (KM)** or **Veitch diagrams (VD)**. KM is built by unfolding the  $n$ -dimensional cube to the plane. The cube nodes are represented by squares of the KM, which coordinates are equal to the coordinates of the cube nodes. To simplify the representation, the rows and columns of KM, where the coordinate is equal to 1, are marked by the bold line. KM of BF (4) is shown in Fig.1.5 (b). Due to the fact, that selecting the  $k$ -cubes minimizes BF, and these cubes are easily found in KM, KM is often used in the LN synthesis. Such a process is shown below.

### 1.3 Combinational logic networks

The operation algorithm for any **digital network** with  $n$  inputs and  $m$  outputs can be described by  $m$  Boolean equations  $Y_i = f_i(X_1, X_2, \dots, X_n, Z_1, Z_2, \dots, Z_k)$ , ( $i=1, \dots, m$ ), where  $Y_i, X_j$  are output and input variables ( $j=1, \dots, n$ ),  $Z_i$  are variables which represent the inner state of the network ( $j=1, \dots, n$ ). This is the state of some

memory elements, which usually are triggers. Such a network can be represented by two parts (see Fig.1.6). One of them consists of triggers (T). Another one contains LEs, which are connected into the **combinational logic network** (LN). Both parts interact through variables  $Z_i$ , which characterize the trigger states, and variables  $D_i$ , which are the trigger **stimulating functions**.

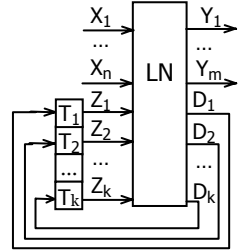


Fig.1.6

The main feature of LN consists in the following. When all the transition processes are finished in the inner LEs, then the output signals depend only on the input signals, and the inner signal states are not needed to derive these output signals. Therefore, equivalent LNs can have different inner structure. The goal of the logic synthesis is to minimize the LN complexity, and maximize its speed, selecting its optimum inner structure.

To develop LN, based on the given LE library, BF has to be represented by the superposition of LE functions. These LE functions are named as **operators**, and their superposition is the **operator representation** of BF. Such representation process is named as the mapping of BF into LN. As the base BF for the logic synthesis, the minimum normal form is selected. The **minimum normal form** is one of eight BFs (3), (4), (5), which has the minimum number of input signal symbols and their negations.

Consider the example of the logic synthesis of the carry network of an adder (Fig.1.7 (a)). The carry function  $C_0$  is equal to a 1 if two or more arguments are equal to a 1. KM of this BF is shown in Fig.1.7 (b).

The 1-cube of  $n$  variables, which was mentioned in the previous chapter, has the property, that it can be represented by the AND function of  $n-1$  variables. Three 1-cubes are selected in the KM in Fig.1.7 (b). One of them is  $((C, X, Y), (C, X, \bar{Y}))$ . Due to the fact that the distance between terms in the cube is equal to one, the conjunction of them is  $CXY \vee CX\bar{Y} = CX(Y \vee \bar{Y}) = CX \cdot 1 = CX$ . As a result, the "gluing" of terms occurs, and BF is reduced.

By BF minimizing, its KM is fully covered by  $0-, 1-, \dots, k-$  cubes, as it is shown in Fig.1.7 (b). The conjunctions of terms, representing those (prime implicants) are reduced as shown above. The resulting BF is the conjunction of all reduced prime implicants. In our example it is  $C_0 = CX \vee CY \vee XY$ . This is the first minimum normal form. Another 7 minimum normal forms can be derived as in (4), (5):

$$\begin{aligned} C_0 &= \overline{CX} \cdot \overline{CY} \cdot \overline{XY} = (\overline{C} \vee \bar{X})(\bar{C} \vee \bar{Y})(\bar{X} \vee \bar{Y}) = \overline{C} \vee \bar{X} \vee \bar{C} \vee \bar{Y} \vee \bar{X} \vee \bar{Y} = \overline{C\bar{X}} \vee \overline{C\bar{Y}} \vee \overline{X\bar{Y}} = \\ &= \overline{C \vee X} \vee \overline{C \vee Y} \vee \overline{X \vee Y} = (\overline{C \vee X}) \cdot (\overline{C \vee Y}) \cdot (\overline{X \vee Y}) = \overline{C\bar{X}} \cdot \overline{C\bar{Y}} \cdot \overline{X\bar{Y}}. \end{aligned}$$

It should be mentioned that four last forms are derived when 1-cubes are selected for the inversed terms, or for zeroed squares of the KM.

Consider the LE library that consists of only 2-input AND-NOT gates, or shortly, 2NAND gates. Then we select the AND-NOT/AND-NOT minimum normal form. We have to design the 3-input LN of the outer stage. Such a LN can

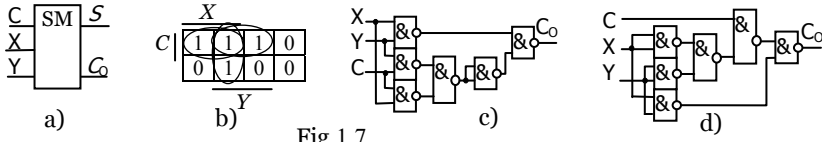


Fig.1.7

be implemented due to the formula:  $\overline{\overline{XYZ}} = \overline{XYZ}$ . The resulting operator representation is  $C_0 = \overline{CX \cdot CY \cdot XY}$ , and it is implemented on 6 gates (Fig.1.7(c)).

To minimize the LN some variables in the formula can be put out of brackets.

For example,  $CX \vee CY \vee XY = C(X \vee Y) \vee XY = \overline{C \overline{X} \cdot \overline{Y}} \vee XY = \overline{C \overline{X} \cdot \overline{Y} \cdot \overline{XY}}$ . The resulting LN is shown in Fig.1.7 (d).

BF can be optimized by the method of decomposing by a variable. This method is based on the equation  $f(X_1, X_2, \dots, X_n) = \overline{X_1} \cdot f(0, X_2, \dots, X_n) \vee X_1 \cdot f(1, X_2, \dots, X_n)$ . Here BFs with variable 0 and 1 are derived from respective halves of KM. In our example  $C_0 = \overline{C} \cdot C_0(0, X, Y) \vee C \cdot C_0(1, X, Y) = \overline{C} XY \vee C XY \vee C(X \vee Y) = (\overline{C} \vee C) XY \vee C(X \vee Y) = XY \vee C(X \vee Y) = XY \vee C \overline{X} \cdot \overline{Y} = \overline{XY \cdot C \overline{X} \cdot \overline{Y}}$ , i.e. we have derived the same LN as is shown in Fig.1.7 (d).

The optimization process is finished by the selection of better LN due to a set of criteria. As the simplest complexity criterion, the Quine complexity can be selected which is equal to the amount of all the gate inputs. This criterion was true for small scale ICs when their cost was proportional to their pin number. Now in the ASIC design each LE from the library has its area, which it occupies on the chip surface. Then the LN complexity is equal to the sum of all the LE areas. When LN is configured in FPGA then all LEs are mapped into 4-input look-up tables (LUTs) or logic cells (LCs). Then the LN complexity is equal to the number of used LUTs or LCs. In any case, there is the common practice to measure the complexity in the number of 2-input equivalent gates.

Comparing derived LNs in Fig.1.7 (c,d), we can see that their complexities are equal to 6 gates. But the complexity of the second LN is something less, because it contains more NOT gates. This LN can be used as the subnetwork of some complex LN, in which the variable sources and its invertors can be common for the whole LN. In this situation these NOT gates stay on the LN inputs. Therefore, they would not be considered.

The LN speed can be estimated as the number of gate stages in it, which is equal to the number of gates in the longest path from any input to any output of LN. Because the delay of CMOS circuits usually does not depend on the logic signal levels then we can not consider the input signal levels in the speed estimation. Comparing LNs in Fig.1.7 (c,d) shows that both of them have the delay of 4 gates. For real ICs the delay of a single gate is equal to 0,1 ns. Therefore, the delay of synthesized LNs is equal to 0,4 ns.

Due to the formula (1), the LN power consumption by clock frequency  $f_c$ , and voltage  $V$  can be calculated on the base of gate fanouts and of average switching

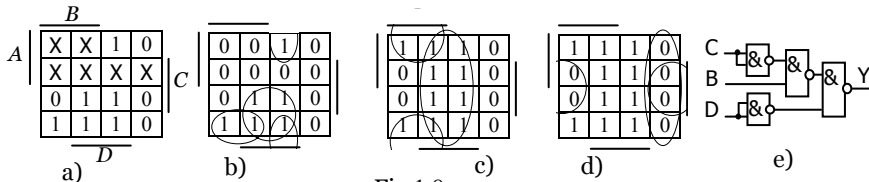


Fig.1.8

activity  $E_s$ . When  $E_s$  is estimated to be 50%, then the power consumption is proportional to the sum of input numbers of all gates. For examples in Fig.1.7(c,d) these figures are equal to each other. More precisely the power consumption is calculated by simulation of the LN model with the real input data sets. Then the figure  $E_s$  is calculated precisely for each gate.

In many cases, BF is not defined on some subsets of input data. Such BF is called as **partially defined** BF. For example, in Fig.1.6, the trigger stimulating functions  $D_i$  would not be defined for the states which never occur in the network. They are called as prohibited states. Taking in consideration this situation, LN with minimized hardware can be derived. Consider the design of the LN for encoding of binary decimal code (BCD) to 7-segment LED display code. Such display has to view digits from 0 to 9, and another 6 possible combinations do not occur. The KM of BF for coding the third LED switching is shown in Fig.1.8 (a). Here unused subsets are marked by X sign and are "**don't care**" conditions.

The "don't care" conditions can be specified as either a 0 or a 1. Consider all of them are equal to a 0. Then we can select the proper cubes (Fig.1.8 (b)), and derive the following minimum normal form:  $Y = \overline{A}D \vee \overline{A}BC \vee \overline{B}C\overline{D}$ . Note that KM is the unfolded 4-dimensional cube, and therefore, on KM the cut ellipses cover a single 1-cube. It can be proven that the distance between its terms is equal to 1. Here four squares, covered by the circle, form the 2-cube, which is reduced to  $\overline{A}D$ . We can assign the undefined states more speculative, to get the cubes of higher order (Fig.1.8(c)). The derived BF is  $Y = D \vee \overline{B}C$ , and is much simpler than the previous one. Its operator representation in the 2NAND operators is  $Y = \overline{D} \cdot \overline{B}C$ . In such a manner, the inverse BF can be derived from the KM in Fig.1.8 (d):  $Y = \overline{B} \cdot \overline{D} \vee C \cdot \overline{D} = \overline{D}(\overline{B} \vee C) = \overline{D} \cdot \overline{B}C$ , i.e. we get the same representation. The resulting LN is shown in Fig.1.8 (e).

In the CAD tools for the logic synthesis the BF optimization and its mapping into LN is made automatically without the designer interference. But many CAD tools can optimize complex BFs of more than 8-10 variables not optimally. It is explained by the fact that the logic optimization is the heavy combinatorial process, and for the affordable period of time the optimum solution could not be found. In this situation, the hand-made BF optimization may give better results. Besides, only selected CAD tools provide optimization of partially defined BFs (with "don't cares"). Therefore, the experienced designer must be able to optimize complex BFs by hand.

### 1.4 Triggers

A **trigger** is a logic element that can hold one of  $N$  stable states. The most widely used trigger has  $N=2$  states, and is named as binary trigger. The trigger is said to store the figure 0 (or 1) if it is in the state zero (one). The triggers are distinguished as latches and flip-flops. A **latch** is a trigger that can follow data variations and transfer them to an output line. It is characterized by two main properties: — it is transparent in that the output  $Q^t$  follows changes at the least part of the time  $t$ ; — the storage is achieved using a bistable circuit, in which either  $Q=0$  or  $Q=1$  can be held.

Table 1.5

| $\bar{R}$ | $\bar{S}$ | $Q^{t+1}$ | $\bar{Q}^{t+1}$ |
|-----------|-----------|-----------|-----------------|
| 0         | 0         | ?         | ?               |
| 0         | 1         | 0         | 1               |
| 1         | 0         | 1         | 0               |
| 1         | 1         | $Q^t$     | $\bar{Q}^t$     |

An **SR latch** has two inputs that are labeled  $S$  and  $R$ . This is associated with the quite general terminology "set" and "reset", that means that we force  $Q$  to a value 1 or 0. An SR latch can be built using two cross-coupled NAND gates, as shown in Fig.1.9 (a), and its symbol is shown in the Fig.1.9 (b). The algorithm of this latch is conveniently represented

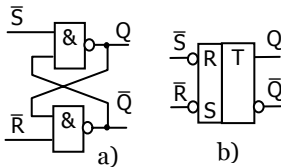


Fig.1.9

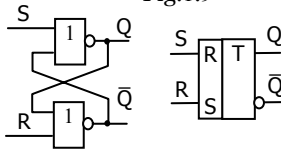


Fig.1.10

by the Table 1.5, which is named as the function table. Here the symbol  $Q^{t+1}$  means the latch state at the moment of time  $t+1$ , i.e. after the switching process is finished, which is caused by the input signals, that were active at the moment  $t$ . From the latch equations  $Q = \bar{Q} \cdot \bar{S}$  and  $\bar{Q} = \bar{Q} \cdot \bar{R}$  (see the Table 1.5) the following equalities are derived:

$Q = \bar{Q} \cdot 1 = \bar{Q}$  and  $\bar{Q} = \bar{Q} \cdot 1 = \bar{Q}$ , which are true for any  $Q$ . Therefore, when  $\bar{R} = \bar{S} = 1$  the state  $Q^{t+1}$  is fully derived from the previous state  $Q^t$ . The signal 0 at the output  $Q$  forces the signal 1 at the output  $\bar{Q}$ , which respectively keeps the signal  $Q=0$  when it enters the input of NAND gate (see Fig.1.9). When

$\bar{R}=0$  and  $\bar{S}=1$  then the output signals are  $Q=0$  and  $\bar{Q}=1$ , which stay stable after input signal exchange to  $\bar{R}=\bar{S}=1$ , because mentioned signals keep themselves. But when  $\bar{R}=\bar{S}=0$  the output signals are  $Q=1$  and  $\bar{Q}=1$ , which after the event  $\bar{R}=\bar{S}=1$  are switched into one of possible states  $Q=0$  and  $\bar{Q}=1$ , or  $Q=1$  and  $\bar{Q}=0$ . The signals  $\bar{R}=\bar{S}=0$  cannot be simultaneously, because they force the indefinite operation.

The similar latch is built on the NOR gates, which is distinguished from the previous one by the direct inputs  $R$  and  $S$  (Fig.1.10).

In a complex digital system the designer must carefully control the flow of data to insure that the proper information is available to each block when it is needed. The common way to control the data movement within a network is to synchronize the system operation using a well-defined reference such as a clock signal. A **clock** is a control signal that periodically makes a transition from a 0 to a 1 and then back to a 0. The clock is usually denoted as  $C$  or  $CLK$ . Using a clock



signal to control the operation of a trigger provides us with the ability to dictate the times when data values can be stored in the device. This allows for the design of complex digital networks in which the data is moved in a synchronous manner.

The network diagram of a clocked SR latch and its symbol are shown in Fig.1.11. Comparing this with Fig.1.9, we see that  $C$  is ANDed with both the  $R$  and  $S$  inputs. When  $C=1$  the latch is operated as SR latch, accepting the information on the inputs  $R$  and  $S$ . When  $C=0$  the latch falls in the storing mode, in which the  $R$  and  $S$  inputs do not infer its state.

Two inputs  $R$  and  $S$  make the latch control complex, and afford two interconnection wires. This makes the disadvantage of the SR latch, which is absent in the D latch. A clocked **D latch** may be created in the same manner, as illustrated in Fig.1.12. As with the clocked SR latch above, the input is only active when  $C=1$ . The clocked D latch is often called a transparent latch due to its behavior during this time. Clocked latches are useful in synchronizing the data flow through a complex system. They also give more meaning to the name "latch" as they can be visualized as circuits that "latch on to" data when  $C=0$ .

The latches are never used as the triggers in the digital networks with the feedback like in Fig.1.6. For instance (see Fig.1.6), in some situation for the signal  $Z_1=0$ , LN generates the signal  $D_1=1$ , and for the signal  $Z_1=1$ , LN gives  $D_1=0$ . Then when latch  $T1$  is opened by the clock, the high frequency oscillations occur due to the feedback chain, which traverses through the latch and LN.

To prevent such a situation, the two staged triggers are used, named **flip-flops** (FFs). The simplest way to design the FF is cascading two clocked SR latches as it is shown in Fig.1.13 (a). The first latch is designated as the master circuit and is responsible for securing the input data  $R$  or  $S$ . The second latch acts as the slave. It is used to hold the value of the data that it receives from the master. The master and the slave circuits are controlled by opposite phases of the clock  $C$ . Since the master latch has  $C$  applied to it, it accepts inputs when  $C=1$ . The slave, on the other hand, uses  $\bar{C}$  for timing, so that it allows for changes in the inputs when  $C=0$ .

The value, that is transferred to the slave circuit (and hence to the output  $Q$ ) is the value, that is in the master latch, when the clock makes a transition from  $C=1$  to  $C=0$ . For this reason, this master-slave configuration is classified as being a falling edge-sensitive device. And such a trigger is the edge-triggered FF. In the FF symbol on the Fig.1.13 (b) such edge sensitive input is designated as  $\downarrow$ . The rising edge sensitive input is designated as  $\uparrow$ . Alternative designations for rising edge and falling edge

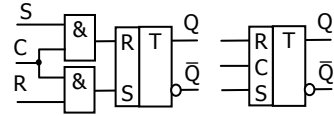


Fig.1.11

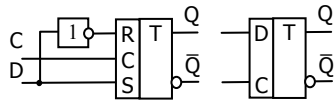


Fig.1.12

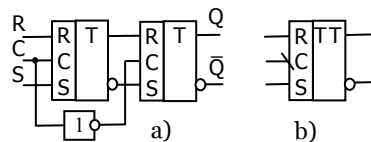


Fig.1.13

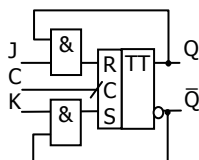


Fig.1.14

clock inputs are  $\triangleright$  and  $\triangleleft$ , respectively. Two letters T in the symbol associate with two latches in FF.

When the master latch is of D-type, then we derive the **D flip-flop**. The **JK flip-flop** is distinguished by the feedback in the master-slave circuit (see Fig.1.14). It behaves as the original SR flip-flop but when  $J=K=1$  then FF exchanges its state to the opposite one, i.e.  $Q^{t+1} = \bar{Q}^t$ . JK-type FF used to be dominant in designs that were based on small scale

integration ICs, but can be useful now in some special networks.

The **toggle flip-flop** is a circuit that has a single input  $T$ . The operation of this FF is exactly as implied by its name: the output toggles whenever  $T$  changes from 0 to 1. When toggle FF has the edge sensitive clock input then its output toggles with each clock rising edge when  $T=1$ . Such FF is derived from JK-type FF when inputs  $J$  and  $K$  are coupled together. T flip-flop is a relatively special LE that does not have the versatility of FFs discussed above but may be useful, for example, in the counters.

FFs as well as latches are never designed as the networks of gates, because of unpredictable behavior of the derived circuits. This is explained by the fact that in modern circuits the delays in wires can supersede the gate delays. As a result, for example, the prohibited condition  $R=S=1$  can occur in unexpected moments. Latches and FFs are usually designed as the transistor circuits when the proper technology gate library is formed. By this process, the complex problem of signal races both in gates and in wires between them is solved.

Table 1.6

| C          | J | K | $Q^t$ | D           | $Q^{t+1}$   |
|------------|---|---|-------|-------------|-------------|
| 0          | X | X | X     | X           | $Q^t$       |
| $\uparrow$ | 0 | 0 | $Q^t$ | $Q^t$       | $Q^t$       |
| $\uparrow$ | 0 | 1 | X     | 0           | 0           |
| $\uparrow$ | 1 | 0 | X     | 1           | 1           |
| $\uparrow$ | 1 | 1 | $Q^t$ | $\bar{Q}^t$ | $\bar{Q}^t$ |

DFFs are most widely used. Additionally, they usually have  $S$  or  $R$  input, or both of them for asynchronous set or reset to the initial state. Very often DFFs have the enable input  $CE$ , which enables the FF clock sensitivity. Another FF types are designed on DFFs as on the component. Consider the design of JKFF. Its function table is the Table 1.6. Here the arrow means the clock rising edge, the letter X means the "don't care" state. The analysis of this table shows

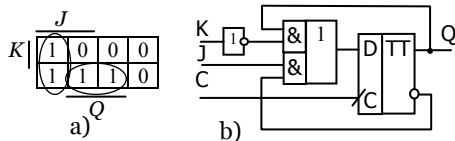


Fig.1.15

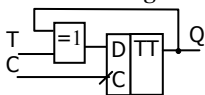


Fig.1.16

that LN is needed which is attached to the input D of the FF. The respective KM is shown in Fig.1.15 (a), and the resulting FF network is illustrated by Fig.1.15 (b). Fig.1.16 illustrates the T-type FF based on D-trigger. In the following chapters the main components of logic networks are described, which are based on logic gates and triggers.

### 1.5 Decoders

The combinational network, which implements a set of BFs:

$$Q_j = A_1^{a_1^j} A_2^{a_2^j} \dots A_n^{a_n^j}, \quad (6)$$

is named as **decoder** (DC), where  $j = 0, \dots, 2^n - 1$ ,  $A_i$  is the input variable, ( $i=1, \dots, n$ ),  $a^i$  is the  $i$ -th digit of the binary representation of  $j$ .

From (6) one can see that to develop the DC network the AND gates are needed. Consider the  $p$ -input AND gates, and the variables  $A_i$  and  $\bar{A}_i$  are generated out of DC. If  $p \geq n$  then the DC design is simple: this DC consists of  $2^n$  gates (see Fig.1.17), each of them implements one BF (6). If  $p < n$  then function  $Q_j$  has to be formed by dividing  $A_i$  to the sets of up to  $p$  variables. LN in which the conjunctions are implemented in parallel has the highest speed. In Fig.1.18 (a) the DC network is drawn, which has the maximum speed, and which implements the 16-input AND function on the 3-input gates. Its delay is equal to  $3t$ , where  $t$  is the gate delay. The number of gates is equal to 8. In general,

$$t(n, p) = [\log_p n]t, \quad L(n, p) = [(n - 1)/(p - 1)], \quad (7)$$

where  $t(n, p)$  and  $L(n, p)$  are time delay and gate number of the LN, respectively,  $[x]$  means the nearest higher natural number of  $x$ . The whole DC network for  $2^n$  outputs contains  $2^n[(n - 1)/(p - 1)]$  of  $p$ -input gates. But such LN has large hardware volume. Firstly, it contains up to  $2^n$  conjunction gates, in this example, say  $A_1 A_2 A_3$ . Really, only eight such conjunctions are needed. Therefore, it would be better to take off the unnecessary gates. In Fig.1.18(a) the figures above the gate symbols show how many gates are really needed. Secondly, it can occur that some gate inputs in the last stages are not engaged (Fig.1.18 (a)). It would be better to divide the input variables into the sets to minimize the free gate inputs, as it is shown in Fig.1.18 (b).

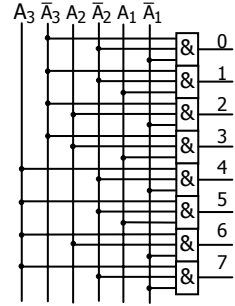


Fig.1.17

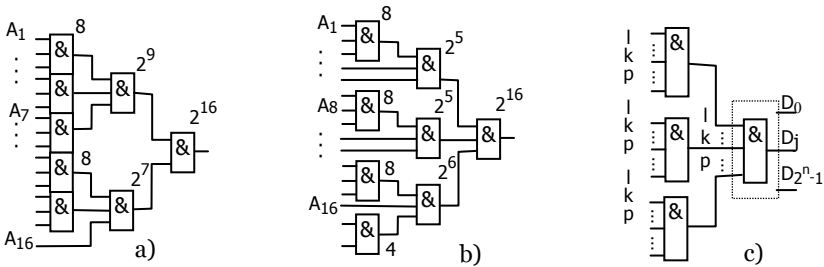


Fig.1.18

The optimum solution of DC network (according to both speed and hardware volume) has the structure, represented in Fig.1.18(c). Its complexity is derived from the formula

$$C(n, p) = 2^n + \sum_{i=1}^p C(n_i, p), \quad (8)$$

where  $n_i$  is the natural number, such that  $\sum_{i=1}^p n_i = n$ . The formula (8) achieves the minimum, when  $n_i$  has the value that is nearest to the value of  $n/p$ . Getting the optimum division of input digits to  $p$  groups, a set of decoders was designed. Their parameters are shown in the Table 1.6.

Table 1.6

| $n$      | 3 | 4     | 5     | 6     | 7     | 8     | 9     | 10         | 11                |
|----------|---|-------|-------|-------|-------|-------|-------|------------|-------------------|
| $n_i$    | 3 | 2,1,1 | 2,2,1 | 2,2,2 | 3,2,2 | 3,3,2 | 3,3,3 | 3,3(2,2,1) | 3,(2,1,1),(2,1,1) |
| $C(n,3)$ | 8 | 20    | 40    | 76    | 144   | 276   | 536   | 1060       | 2096              |

Consider  $n_i \approx n/p$  then the equation (8) can be unfolded:

$$C(n, p) \approx 2^n + \sum_{i=1}^p C(n/p, p) = 2^n + pC(n/p, p) = 2^n + p2^{n/p} + p \sum_{i=1}^p C(n/p^2, p) \dots$$

Then we derive the resulting equation:

$$C(n, p) \approx 2^n + p2^{n/p} + p^2 2^{n/p^2} + p^3 2^{n/p^3} + \dots \quad (9)$$

Consider  $n=11$  and  $p=3$ , then due to three first items of the formula (9)  $C(11,3) = 2107$ , i.e. it gives rather good estimation, comparing to the Table 1.6. If the DC output number is represented as  $M=2^n$  then the equation (9) is represented in another form

$$C(n, p) \approx M + p\sqrt[p]{M} + p^2 \sqrt[p^2]{M} + \dots \quad (10)$$

In many cases, it is useful to estimate the DC complexity as the sum of LE inputs. Such estimation is equal to the formula (10) multiplied by  $p$ :

$$C(n, p) \approx pM + p^2 \cdot \sqrt[p]{M} + \dots \quad (11)$$

The analysis of the formulas (7) and (11) shows that DC on 2-input LEs has the minimum hardware volume and maximum time delay, and DC on  $n$ -input LEs has the large hardware volume and the small time delay.

At present, small DCs with  $M < 100$  are implemented on PLA and CPLD. When designing ASIC, DC network is usually got from the library or it is generated by the special subprogram, or is synthesized from the behavioral description. The LEs of FPGA usually have the limited number of inputs (mostly 4). Therefore, to develop DCs in FPGA the designer must take into consideration the methods of DC network building.

## 1.5 Multiplexers

The combinational network which has up to  $2^n$  data inputs, one data output, and  $n$ -bit wide control input, which selects one of the data input, is usually named as an **multiplexor**. Consider the  $2^2 = 4$  – input multiplexor. Then its BF is

$$Y = E \cdot (D_0 \cdot \bar{X}_1 \cdot \bar{X}_0 \vee D_1 \cdot \bar{X}_1 \cdot X_0 \vee D_2 \cdot X_1 \cdot \bar{X}_0 \vee D_3 \cdot X_1 \cdot X_0), \quad (12)$$

where  $E$  is the enable signal,  $D_i$  is the input data,  $X_1, X_0$  are digits of the selected data position. Comparing this equation and equation (6), one can find that multiplexor is the combination of DC, and AND/OR networks, which is illustrated by Fig.1.19.

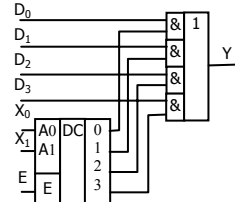


Fig.1.19

The standard 2- and 4- input multiplexers are usually in the project libraries. The large multiplexers are designed on the base of them as the multiplexor trees. The example of the 6-input multiplexor, based on 2-input multiplexers, is shown in Fig.1.20.

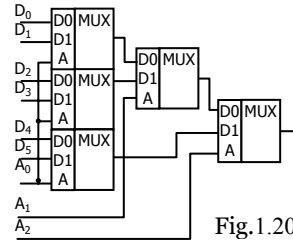


Fig.1.20

In different projects the multiplexers are widely used to provide the sharing of the computational resources among different data sources. For example, they are inserted at the inputs of ALUs to put the input dates from different directions.

In ASICs the tri-state busses are implemented rarely because of their high cost and low reliability. Therefore, the common busses are made on the multiplexor basis. In this situation the output of the  $n$ -input multiplexor is connected to all the bus destinations (for example, processor units – PUs). And the output of the  $j$ -th source is connected to the  $j$ -th input of the multiplexor, where  $j \leq n$  (see Fig.1.21). The bus address to the multiplexor and the enable signals to destinations are formed by the arbiter network, which is not shown in Fig.1.21. As the simplest case of the common bus, consider the registered memory in which all the register outputs are connected together through the common multiplexor.

In more complex situations, any source can be connected to any destination. Then up to  $n \times n$  multiplexers are needed, as it is shown in Fig.1.22.

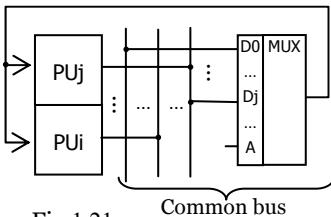


Fig.1.21

Common bus

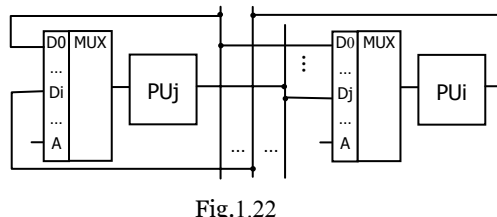


Fig.1.22

Table 1.7

| $D_i$ | $A_3$ | $A_2$ | $A_1$ | $A_0$ |
|-------|-------|-------|-------|-------|
| $D_0$ | 0     | 0     | 0     | 0     |
| $D_1$ | 0     | 0     | 0     | 1     |
| $D_2$ | 0     | 0     | 1     | 0     |
| $D_3$ | 0     | 0     | 1     | 1     |
| $D_4$ | 0     | 1     | 0     | 0     |
| $D_5$ | 0     | 1     | 0     | 1     |
| $D_6$ | 0     | 1     | 1     | 0     |
| $D_7$ | 0     | 1     | 1     | 1     |
| $D_8$ | 1     | 0     | 0     | 0     |
| $D_9$ | 1     | 0     | 0     | 1     |

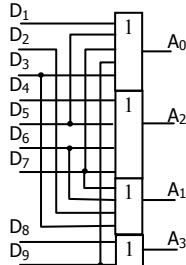


Fig.1.23

Table 1.8

| $D_9 - D_0$ | $A_3$ | $A_2$ | $A_1$ | $A_0$ | $P$ |
|-------------|-------|-------|-------|-------|-----|
| 0000000001  | 0     | 0     | 0     | 0     | 1   |
| 000000001x  | 0     | 0     | 0     | 1     | 1   |
| 00000001xx  | 0     | 0     | 1     | 0     | 1   |
| 0000001xxx  | 0     | 0     | 1     | 1     | 1   |
| 000001xxxx  | 0     | 1     | 0     | 0     | 1   |
| 00001xxxxx  | 0     | 1     | 0     | 1     | 1   |
| 0001xxxxxx  | 0     | 1     | 1     | 0     | 1   |
| 001xxxxxxx  | 0     | 1     | 1     | 1     | 1   |
| 01xxxxxxx   | 1     | 0     | 0     | 0     | 1   |
| 1xxxxxxx    | 1     | 0     | 0     | 1     | 1   |
| 0000000000  | x     | x     | x     | x     | 0   |

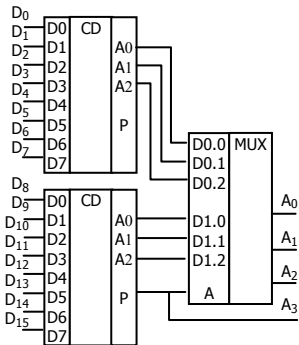


Fig.1.24

## 1.6 Encoders

The **encoder** is a combinational network, which transforms the input data into the position of the most significant bit of this data. The example of the encoder is LN, which transforms the signals from the decimal button array to the **binary-decimal code** (BCD)  $A_3A_2A_1A_0$ , which represent the pushed button from 0 to 9 (see the Table 1.7). One of the possible LNs of such an encoder is shown in Fig.1.23. But when two buttons are pressed simultaneously then LN generates the incorrect code. For example, if  $D_5=D_6=1$  then  $A_3A_2A_1A_0=0111$ , that represents the signal  $D_7=1$ . Besides, it is impossible to recognize the pressed button, for example, the button 0.

To remove these disadvantages it is necessary to synthesize the **priority encoder**. Such encoder always forms the code of a single pressed button, for instance, more significant one. The Table 1.8 is the truth table of such an encoder. When any button is pressed then the output bit  $P=1$ . The proper Boolean equations are

$$A_3 = D_9 \vee D_8, A_2 = (\overline{D_9 \vee D_8})(D_7 \vee D_6 \vee D_5 \vee D_4)$$

$$A_1 = (\overline{D_9 \vee D_8})((D_7 \vee D_6) \vee (\overline{D_5 \vee D_4})(D_3 \vee D_2)),$$

$$A_0 = D_9 \vee \overline{D_8}(D_7 \vee \overline{D_6}(D_5 \vee \overline{D_4}(D_3 \vee D_2D_1))),$$

$$P = D_0 \vee D_1 \vee A_3 \vee A_2 \vee A_1.$$

To build many input priority encoders the hierarchical LN is used, which consists of small encoders. Consider we have the encoder unit, which is built due the previous equations, but the inputs  $D_8, D_9$  are not used (zeroed). Then one of possible 16-input encoders is shown in Fig.1.24.

Here the multiplexor selects the group of bits, which is generated by the activated coder, which has the higher priority. Due to these principles, the priority encoders are built, which are used, for example, as interrupt encoders. Another example is LN which finds the number of zero bits before the

most significant bit in the mantissa during its normalization.

### 1.7 Shifters

The **shifter** is a combinational network, which transfers the input word to the output with the shift of its bits. The Table 1.9 shows how the logic right shift of the 4-bit word  $X$  is implemented in the shifter, giving the output word  $Y$ .

The shift bit number is given by the word  $A_1A_0$ . Due to the logic shift, the bit, which is shifted in, is zero. In the case of the arithmetic shift, the left shifted bit would be the sign bit, here  $X_3$ . It is useful to build the shifters on the base of the multiplexor. Such shifter, which implements this algorithm, represented by the Table 1.9, is illustrated by Fig.1.25.

According to this principle, the shifter up to  $n-1$  digits is based on the  $n$ -input multiplexers. But when  $n$  is larger than 6–8 then the hardware volume is too high. In this situation the multistage shifters are formed.

One can design the shifter to 0, 4, 8, and 12 bits based on 4-input multiplexers. Then the complex shifter consists of such shifter, named as U1, and the usual shifter to 0,1,2,3 bits, named as U2. Such a shifter is illustrated by Fig.1.26. Here the bit shift number is given by the code  $A_3A_2A_1A_0$ . The most significant bits  $A_3A_2$  control the shifter U2, and digits  $A_1A_0$  control the shifter U1. Consider we have to shift the code  $X$  to 13=1101 bits. Then the shifter U1 shifts it to 1 bit, and the shifter U2 shifts it to 12 bits.

Table 1.9

| $A_1A_0$ | $Y_3$ | $Y_2$ | $Y_1$ | $Y_0$ |
|----------|-------|-------|-------|-------|
| 0 0      | $X_3$ | $X_2$ | $X_1$ | $X_0$ |
| 0 1      | 0     | $X_3$ | $X_2$ | $X_1$ |
| 1 0      | 0     | 0     | $X_3$ | $X_2$ |
| 1 1      | 0     | 0     | 0     | $X_3$ |

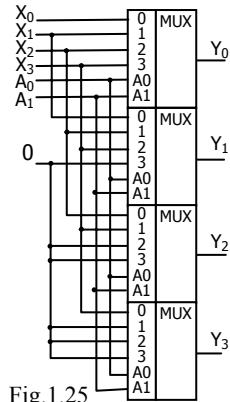


Fig.1.25

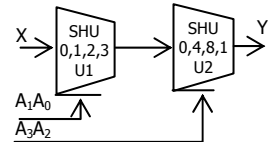


Fig.1.26

### 1.8 Binary adders

Binary **adders** or **summaters** (shortly SM) are used for addition of binary integer numbers

$$Q = B + D, \text{ where } B = B_{n-1}2^{n-1} + \dots + B_12 + B_0, D = D_{n-1}2^{n-1} + \dots + D_12 + D_0.$$

The combinational binary adders, which consist of one bit adders (**full adders**), are mostly used. The Table 1.10 is the truth table of such full adder. Here  $Q_i$  is the sum of  $i$ -th bits  $B_i$  and  $D_i$ ,  $C_i$  is the **carry** bit to the  $i$ -th bit. KM for the output  $C_{i+1}$  is illustrated by the Fig.1.7(b) for  $X = B_i$ ,  $Y = D_i$ . And the proper Boolean equation is  $C_{i+1} = B_i D_i \vee B_i C_i \vee D_i C_i$ . KM for the output  $Q_i$  is illustrated by the Fig.1.5(b) for  $X_1 = B_i$ ,  $X_2 = D_i$ ,  $X_3 = C_i$ . Therefore, this BF could not be simplified, and is equal to  $Q_i = B_i D_i C_i \vee B_i \bar{D}_i \bar{C}_i \vee \bar{B}_i D_i \bar{C}_i \vee \bar{B}_i \bar{D}_i C_i$ . Additionally, BF  $\bar{C}_{i+1}$  is needed to simplify the circuit of the  $(i+1)$ -th stage of the adder. The respective network occupies 10 gates.





The subtraction function is usually performed using the 2-s complement approach. **2-s complement** of  $n$  bit binary number  $D \neq 0$  is defined as

$$\{D\}_2 = 2^n - D = (2^n - 1 - D) + 1 = (1 \dots 11 - D) + 1.$$

The number in brackets is **1-s complement** of  $D$ , which is formed by inversion of all the bits of the number  $D$ . The subtraction is calculated as  $B - D = B + \{D\}_2$ . Therefore, to implement the subtraction, the second number has to be inverted, and a 1 is added to the sum.

In applications both addition and subtraction are required. A single adder/subtractor unit can be built using the ripple-carry adder discussed above. This unit is shown in Fig.1.30. Inspecting the logic diagram shows that 2 basic modifications have been made to the original adder in Fig.1.28. First, each  $D_i$  input line has an XOR gate in its path. Second, a new control bit  $F$  has been added to the circuit, which is connected to each XOR and also to the carry-in bit of the adder. The circuit acts as an adder when  $F=0$ , and as subtractor when  $F=1$ . In the second situation XOR gates work as invertors and carry-in bit is a 1, which is added to the sum.

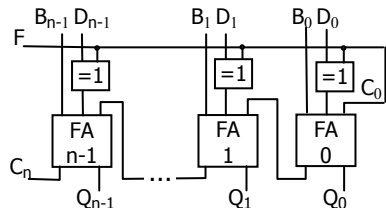


Fig.1.30

### 1.9 Registers

**Register (RG)** is a set of triggers, which have a common control network. A single trigger can be considered as the one bit wide register. Therefore, triggers often are named as registers as well. RG is used for data storing and implementation of some operations with them. The bit wise logic operation with the word  $Q$  in RG and the word  $D$  at its inputs is such operation. Another operations are different shift operations and data output.

RG is considered to be a logic network, which consists of a trigger set (plain register) and logic network (LN) which implements the output functions and functions of trigger stimulating (see Fig.1.6). The RG design consists in selection of the trigger type and in LN synthesis.

When the computing system with RGs is designed, one has to take in considerations the properties of the clock propagation system. Each modern LSI circuit has one or small number of clock propagation trees. Such a tree provides the stable clock signal to each trigger with the minimum clock skew. The **clock skew** is the delay between the edge of the clock signal, which enters the trigger, and the base moment of time. Only when the clock skew is zero, the minimum clock period is estimated as the maximum delay from output of one trigger to the input of another one plus the trigger delay. Thank to this feature, the proving of the design correctness is assured for the circuit with thousands or millions of triggers. Considering this feature of the clock propagation system, in most of cases RGs are implemented on flip-flops.

Table 1.12

| C | R | S | W | D     | $Q^{t+1}$ |
|---|---|---|---|-------|-----------|
| 0 | X | X | X | X     | $Q^t$     |
| ↑ | 0 | 0 | 0 | $Q^t$ | $Q^t$     |
| ↑ | 1 | 0 | X | 0     | 0         |
| ↑ | 0 | 1 | X | 1     | 1         |
| ↑ | 0 | 0 | 1 | DI    | DI        |
| ↑ | 1 | 1 | X | X     | X         |

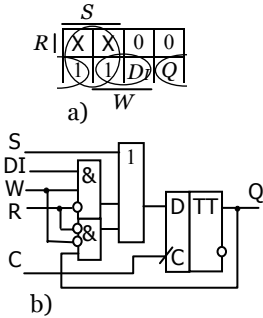


Fig.1.31

and right shift ( $SR$ ) to a single bit. When the left shift operation, the  $i$ -th output is connected to the  $(i+1)$ -th input of RG. The input data  $DL$  is shifted in the first bit. When the right shift, the input data  $DR$  is loaded into the  $(n-1)$ -th bit,  $i$ -th output is connected to the  $(i-1)$ -th input of RG. Note, that  $DL=Q_{i-2}$ ,  $DR=Q_{i+2}$  when  $n=3$  (see Fig.1.32). When shifts are not done, i.e. when  $\overline{SR} \vee \overline{SL} = 1$ , then the  $i$ -th input and output of RG is connected together to provide the information storing.

As was shown in the previous chapter, the subtraction is calculated through the 2-s complement of the data. For this purpose, in some operational units the data registers are used which outputs the direct or inverse code depending on the addition or subtraction operation. A single bit of such RG is shown in Fig.1.33. This is an example of RG with data output operation. The input  $E$  enables the register data storing.

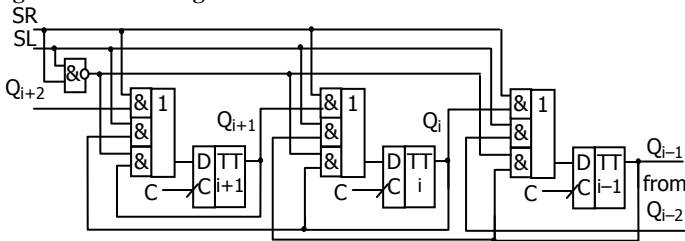


Fig.1.32

To provide the minimum clock skew it is not recommended to attach any logic circuit to the clock input of the trigger. Besides, when the races of the control signals occur, such circuit can generate glitches. The **glitch** is a short, needle formed impulse, which forces the incorrect trigger switching. Instead of controlling the clock signal, the enable input of the trigger is usually used.

Consider the design of RG with the functions of set, reset, and write (control signals  $S$ ,  $R$ ,  $W$ , respectively) based on the D-triggers. The table 1.12 is its truth table, and respective WD is shown in Fig.1.31(a). Here the signals  $S$ ,  $R$  have higher priority. The resulting stimulating function for the  $i$ -th digit of RG is the following:

$$D = S \vee \overline{R} \cdot W \cdot D \vee \overline{R} \cdot \overline{W} \cdot Q.$$

Fig.1.31 (b) illustrates the  $i$ -th bit of the register.

The shift operations in RG can be implemented as left and right shifts to 1, 2,... bits. Fig.1.32. represents a part of the **shift** RG with operations of left shift ( $SL$ ),

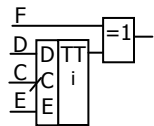


Fig.1.33

### 1.10 Counters

**Counter (CTR)** is a logic unit which implements the increment or decrement operation to 1, 2, etc., and data storing. As any logic circuit, it can be designed as a finite state machine (FSM). But there are some methods of the CTR synthesis, which use its operation specific.

The CTR is a network with FFs. Therefore, the CTR synthesis must obey the rules of the FF synchronization, which were mentioned in the previous section. For example, two decades ago the method of sequential counters was widely used. Such counters are based on asynchronous T-triggers, which are connected sequentially in a chain. In this situation, each trigger is considered to have its own clock signal, which feeds its T-input. When such a CTR is implemented in the LSI circuit, then its design is complicated, and its operation can be unstable.

To prevent this situation, the synchronous T-type FFs are used. A set of such FFs form the specific RG, which bits can be negated synchronously by the clock edge. Consider the count impulse  $C_1$  comes to the T input of the first FF, and its stimulating signal is  $T_0 = C_1$ . Then its output  $Q_0$  exchanges its state every impulse  $C_1$ , i.e. every second impulse  $Q_0 = 1$ . The output of the second FF has to be exchanged every second impulse  $C_1$ , i.e. its stimulating signal has to be  $T_1 = Q_0 C_1$ . The resulting stimulating functions of CTR triggers are the following:

$$T_0 = C_1, T_1 = Q_0 C_1, T_2 = Q_1 Q_0 C_1, \dots, T_i = Q_{i-1} Q_{i-2} \dots Q_0 C_1.$$

The respective 4-bit CTR network is shown in Fig. 1.34. Here the signal on the FF input T enables the operation +1 modulo 2. Such CTR has the name of parallel carry CTR because the carry signal (here  $T_i$ ) to the  $i$ -th bit is formed in parallel by the AND gate.

The usual  $n$ -bit wide counter repeats its states after  $2^n$  input impulses. Such CTR is said to have the period (modulo) equal to  $2^n$ . Often the CTR counting period has to be not equal to  $2^n$ . Such CTRs are designed by excluding the spare states. This is usually achieved by jumping round such states.

Consider the CTR with a period of  $k=5$  cycles. To design such a CTR  $n=\lceil \log_2 k \rceil = 3$  FFs are needed. The states of the CTR with the period of 8 cycles are represented by the table 1.13. Let three last states are to be excluded. Then to go round these states the following states are selected: the state before the first excluded state  $A=(a_2, a_1, a_0) = (1, 0, 0)$ , the first excluded state  $B=(b_2, b_1, b_0) = (1, 0, 1)$ , the last excluded state but the next one  $C=(c_2, c_1, c_0) = (0, 0, 0)$ .

In the usual counter after the state  $A$  goes the state  $B$ , but in this CTR the jump from  $A$  to  $C$  has to be done. Then by the analysis of bits  $a_i, b_i, c_i$  in the  $i$ -th digit the correction of the

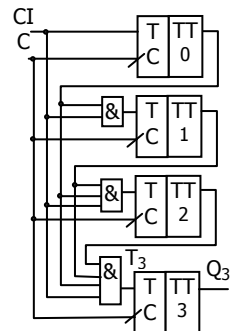


Fig. 1.34

Table 1.13

| $Q_2$ | $Q_1$ | $Q_0$ | State |
|-------|-------|-------|-------|
| 0     | 0     | 0     | $C$   |
| 0     | 0     | 1     |       |
| 0     | 1     | 0     |       |
| 0     | 1     | 1     |       |
| 1     | 0     | 0     | $A$   |
| 1     | 0     | 1     | $B$   |
| 1     | 1     | 0     |       |
| 1     | 1     | 1     |       |

stimulating function of the  $i$ -th FF is found. The three situations are distinguished, when  $b_i = c_i$ ;  $b_i \neq c_i = a_i$ ; and  $b_i \neq c_i \neq a_i$ .

In the first situation the correction of the stimulating function is not needed, because jumps  $a_i \rightarrow b_i$  and  $b_i \rightarrow c_i$  are equal to each other because of  $c_i = b_i$ . In the second one the stimulating function provides the storing mode, due to  $c_i = a_i$ . And in the third one it implements the inversion, because of  $c_i \neq a_i$ , i.e.  $c_i = \bar{a}_i$ .

When  $C$  is all zero state, then such correction is implemented by forming the function  $F = Q_n^{a_n-1} \dots Q_1^{a_1} Q_0^{a_0}$ , which is equal to a 1 only for the set  $(a_2, a_1, a_0)$ . Consider we have the T-type FFs. Then the stimulating functions are corrected

$$\text{due to the rule: } T'_i = \begin{cases} T_i & \text{when } b_i = c_i; \\ T_i \vee F & \text{when } b_i \neq c_i \neq a_i; \\ T_i \cdot \bar{F} & \text{when } b_i \neq c_i = a_i, \end{cases}$$

where  $T'_i$  is a new stimulating function of the trigger.

For this example  $F = Q_2 \bar{Q}_1 \bar{Q}_0$ . The KM of this function, which considers the "don't care" sets, is shown in Fig. 1.35. From this KM the resulting function is  $F = Q_2$ , and  $T'_0 = T_0 \bar{Q}_2$ ,  $T'_1 = T_1$ ,  $T'_2 = T_0 \vee Q_2$ . Because  $T_0 = 1$ ,  $T_1 = Q_0$ ,  $T_2 = Q_1 Q_0$ , then the resulting functions are

$$T'_0 = \bar{Q}_2, T'_1 = Q_0, T'_2 = Q_1 Q_0 \vee Q_2.$$

The network of derived CTR is illustrated by Fig. 1.36. Its waveforms are shown in Fig. 1.37, when the count enable signal is  $C_1 = 1$ . Before its operation CTR has to be set in one of permitted states, for example, in the zeroed state using the R-inputs of its triggers. The count enable signal  $C_1$  feeds the clock enable inputs of the triggers.

|       |       |   |   |
|-------|-------|---|---|
| $Q_0$ | $Q_1$ | 0 | 1 |
| 0     | 0     | 0 | X |
| 0     | 1     | X | 1 |
| 1     | 0     | 0 | X |
| 1     | 1     | 0 | 0 |

Fig.1.35

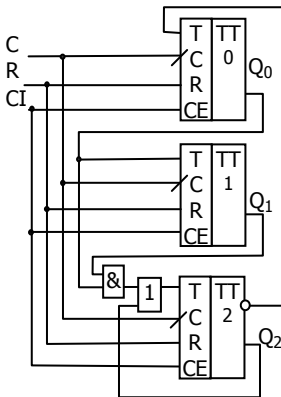


Fig.1.36

Often the CTR is designed, which is based on the couple of adder and RG. The adder adds the increment to the RG content, and the sum is stored in RG each clock impulse when the clock enable (i.e. count enable) signal is active. Here the various increments can be used. Moreover, using the subtraction operation, CTR with the decrement is implemented. In the last situation, to form the 2-s complement the negated outputs of FFs are used.

In the digital network engineering also the **ring counters** are used, which are based on shift RGs with the feedback. Consider the 3-bit shift right RG. In general, the design of the ring counter consists in the synthesis of the feedback LN (see Fig.1.38). The

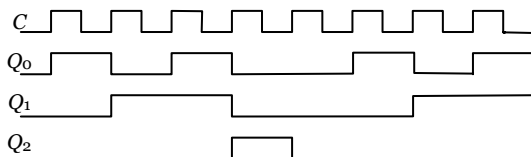


Fig.1.37

well known ring CTR is the **Johnson**-type CTR. Its LN is a single NOT-gate, which connects the output  $Q_0$  to the input  $D_1$ . Then with each clock impulse the state of RG is exchanged, as it is shown in the Table 1.14. The period of state exchange is equal to 6. To prevent this CTR of beginning its operation in the prohibited states 101 or 010, it has to be reset by the signal  $R$ .

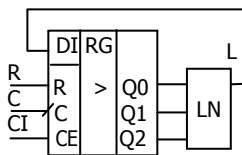


Fig. 1.38

| Cycle | $Q_2 Q_1 Q_0$ |
|-------|---------------|
| 0     | 0 0 0         |
| 1     | 1 0 0         |
| 2     | 1 1 0         |
| 3     | 1 1 1         |
| 4     | 0 1 1         |
| 5     | 0 0 1         |
| 6     | 0 0 0         |

Consider the synthesis of the ring CTR with the period of 5 clock cycles. Let CTR is in the state 000. Then depending on the output  $L$  (0 or 1) after the right shift CTR goes to the state 000 or 100. These branches and another ones are represented by the state diagram, which nodes and edges represent states and branches, respectively (see Fig. 1.39). Five nodes are selected in the diagram, which are connected into a ring by the respective branch edges. They are marked in bold in Fig. 1.39. To provide these branches LN has to output the values 1, 1, 0, 0, and 0 on the sets 000, 100, 110, 011, and 001. Another values are "don't care" ones. These values are represented by KM in Fig. 1.40. The

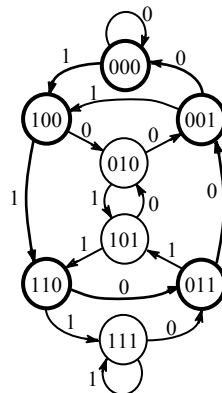


Fig. 1.39

### 1.11 Programmable logic devices

As it was mentioned above, PLAs, CPLDs and FPGAs are widely used by many designers as the base of the customer made logic networks. In this chapter we discuss the ways of developing the LNs based on these devices. Many companies supply the designers by different CAD tools to develop such networks. Usually the company, which produces such devices, provides the proper CAD tool. Such a tool is easily installed in PC, and has the friendly user interface.

There are two different approaches to develop the program for PLAs. First of them is based on drawing the schematic network using the library of standard logic components in the proper graphical editor. The second one is based on the behavioral description of the network by some **hardware description language** (HDL). The second approach has many advantages like fast description and debugging of large projects, the description is independent on the network basis and technology, and it is standardized, and can be accepted by any CAD tool. The most widely used HDL languages are Verilog and VHDL. Therefore all the projects of ASICs and most of projects of CPLDs and FPGAs are designed on HDL. Below we will try to describe many LNs using VHDL.

One logic cell (LC) of PLA can implement any logic function in the form (3) or its inversion. Such form can contain up to  $M$  terms, each of them has no more

|       |       |   |   |   |
|-------|-------|---|---|---|
|       | $Q_1$ |   |   |   |
| $Q_0$ | 0     | X | X | 1 |
|       | X     | 0 | 0 | 1 |
|       | $Q_2$ |   |   |   |

Fig. 1.40

than  $N$  variables and its inversions, and the number of different input variables is no more than  $K$ . For the modern CPLD devices like Altera MAX7000, Xilinx X9500 these parameters are:  $M \leq 5$ ,  $N \leq 52$ ,  $K \leq 52$ . To increase  $M$  the logic expander cells are used. The connection of LC and logic expander is equivalent to connection of two LCs.

For example, LN in Fig.1.1, (a) is described in VHDL by the following entity and its architecture

```
entity SWITCH3 is
port(X1,X2,X3:in bit;
      F: out bit);
end SWITCH3;
architecture BOOL of SWITCH3 is begin
    f<=(not X1 and not X2 and X3) or (not X1 and X2 and not X3) or
      (X1 and not X2 and not X3) or (X1 and X2 and X3);
end BOOL;
```

The entity declares the interface of LN, which shows how to connect this LN in the network of the higher hierarchy level. The architecture is the description of LN behavior due to the designer's algorithm. In this example, such behavior is represented by the operator (statement) of parallel signal assignment, namely, by the Boolean equation. This operator can be freely exchanged by the following one

$$f \leq X1 \text{ xor } X2 \text{ xor } X3 \text{ after } 5 \text{ ns};$$

which represents LN in Fig.1.1 (b). Therefore, a single entity can have different architectures depending on its behavioral description.

In examples the words in bold represent the language reserved words. In the statements they are logic operators of the language. The identifiers represent the signals. The signal in VHDL plays different roles simultaneously. It is the object, which has some value (0 or 1, true or false, integer, etc). It makes signaling (the signal exchange starts the execution the parallel operators in which it is used as an argument), and its time history can be stored and reproduced as the waveform. When the VHDL model is running, the parallel operator is executed any time when any its argument is exchanged, and just in this time (or after delay, which is given by the **after** clause) the result is exchanged.

In the FPGA as LC the look-up table (LUT) is used. LUT represents one bit ROM with the  $k$ -bit address. In the  $i$ -th cell of this ROM the value  $f(\alpha_1, \alpha_2, \dots, \alpha_n)$  is stored, where  $\alpha_1, \alpha_2, \dots, \alpha_n$  are digits of the binary representation of  $i$  (see the eq. (3)). In the modern FPGAs the 3, 4, 5 and 6 – input LUTs are used. When the complex Boolean function is synthesized then it is decomposed in the subfunctions, which are mapped into a set of LUTs. In Fig.1.41 the LUT is represented, which implements the Boolean function from the previous example.

The VHDL model is compiled into the gate level description by the synthesis compiler. By this process, the Boolean equations are usually optimized automatically. If the target is PLA or CPLD then this description is transferred directly into the programming bit stream. In another situation, using the implementation CAD

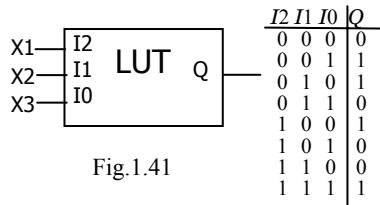


Fig.1.41

tools, this description is mapped into the LC set, which is placed into the target chip, and the proper wire route set is found. The resulting configuration file can be stored in the configuring EPROM or be directed in FPGA.

Consider some VHDL description examples for LNs of different kind. The decoder in Fig.1.17 can be described by the following parallel statement

```
with A select
D<="00000001" when "000", "00000010" when "001",
  "00000100" when "010", "00001000" when "011",
  "00010000" when "100", "00100000" when "101",
  "01000000" when "110", "10000000" when others;
```

Here the signal A is the input address, which selects one of alternative codes to assign to the signal D. Therefore this statement is called as the selective parallel assignment statement. Both signal A and signal D is, so called, bit vector. The digits of D represent the proper decoder outputs. This signal has to be declared as port in the entity as the following

```
D : out bit_vector(7 downto 0);
```

Here the word **downto** shows that the bits in this vector are numbered in the descending direction, from D(7) down to D(0). The decoder can be used as part of larger description, then the signal D is not outputted to the outer space through the port, and is consumed in inner subnetworks. Then it has to be declared as the signal in architecture body before the word **begin** as the following

```
signal D : bit_vector(7 downto 0);
```

The multiplexor can be described by the Boolean equation like (12), or by the following two parallel assignments

```
with X select
T<=D0 when "00", D1 when "01", D2 when "10", D3 when others;
Y<= T and E;
```

Here T is the intermediate signal, the bit vector X represents the address of the multiplexor input, and E is the enable signal.

The priority encoder shown in Fig.1.23 can be represented by the following when-else statement

```
A <= "1001" when D9='1' else "1000" when D8='1' else "0111" when D7='1' else
      "0110" when D6='1' else "0101" when D5='1' else "0100" when D4='1' else
      "0011" when D3='1' else "0010" when D2='1' else "0001" when D1='1' else
      "0000" when D0='1' else "1111";
```

The operator '=' compares bits and returns *true* when they are equal to each other. The **when**-conditions are proven sequentially. The signal  $D_9$  is proven as the first one, then  $D_8$  is proven, etc. Therefore,  $D_9$  has the higher priority level.

VHDL proposes effective tools to describe shifters. For example, the right logic shift operation to  $A$  bits is described by the following operator

```
Y <= X srl A;
```

Here  $X$  is bit vector, and  $A$  has to be declared as integer.

Consider the example of the full comparer design. Such a network compares the bit vectors  $A$  and  $B$ , representing positive integer values. It outputs signal  $A_E=1$  when they are equal, signal  $A_P=1$  when  $A>B$ , and signal  $A_N=1$  when  $A<B$ . The following VHDL program describes such a LN.

```
entity COMPARATOR is
generic (n:integer:=8);           -- bit width of A and B
port (A,B: in bit_vector(n downto 1); -- input dates
      AE, AP, AN: out bit);       -- equal, positive difference, negative difference
end COMPARATOR;
architecture LOG is
  signal aei,api:bit;             -- intermediate results
begin
  aei<='1' when A=B else '0';
  api<='1' when A>B else '0';
  AE<=aei; AP<=api;
  AN<= not aei and not api;
end LOG;
```

The words after two hyphens mean the comments. Identifiers  $aei$ ,  $api$  are the inner network signals, and are declared in the architecture as signals. The operators '=' and '>' compare the bit vectors and return the true value when they are equal or left operand is higher than right one, respectively.

The **generic** clause shows that the network can be adjusted by the generic constant  $n$ . The integer  $n$  means the bit width of input data. For this property this project is multipurpose one, and can be adjusted to different bit widths in the projects where it is used as the component. Moreover, during its instantiating, some outputs can be left open. And in this situation, the synthesis compiler will remove the unnecessary networks, which are directed to the open outputs.

The trigger behavior depends on its state in the previous moment of time. Therefore, the behavioral description of triggers and networks based on them is more complex than description of combinational circuits. For this purpose the process statement is used. This parallel statement represents a small program, which operators are implemented sequentially. There are input and output sig-



nals of the process statement, and inner variables may be held. A subset of input signals forms the sensitivity list. When any signal from the sensitivity list is exchanged, then the process starts to run, and after implementation of its operators it stops. At this moment of time all the output signals get its new value.

The latch in Fig.1.11 is described by the following process statement.

```
process(C,S,R) begin
    if C='1' and R='1' then
        Q<='0';
    end if;
    if C='1' and S='1' then
        Q<='1';
    end if;
end process;
```

Here the list in the brackets is the sensitivity list; the sequential **if**-operators realize the logic behavior of this latch. When  $C = 1$  then the **if**-operators are implemented sequentially and output signal  $Q$  accepts the value depending on  $R$  and  $S$ , i.e. the latch is transparent. When  $C = 0$ , then the signal  $Q$  does not assigned, i.e. it stores its previous value. When  $R = S = C = 1$ , the resulting value is  $Q = 1$ , i.e. it is not undefined value, as for real latches. There are more complex and precise VHDL models of such latches.

It is worth to be mentioned, that when the signal assignment is not implemented in some process running, then such a process describes some latch. In another situation, the process describes some combinational circuit. For example, the multiplexor in Fig.1.19 is described by the following process statement

```
process(E,D0,D1,D2,D3,X1,X0) begin
    if E='1' then
        if X1='0' and X0='0' then Y<=D0;
        if X1='0' and X0='1' then Y<=D1;
        if X1='1' and X0='0' then Y<=D2;
        else Y<=D3;
        end if;
    else Y<='0';
    end if;
end process;
```

The signal  $Y$  is assigned by some value during any process running, and it is not a latch. Consider that the last **else** clause is absent, and then it is the multiplexor with the D latch at its output, which is controlled by the signals  $X1$ ,  $X0$ .

To model the edge sensitive trigger the attribute `'event` is used, which returns the true value, when the rising or falling edge of the proper signal occurs. Below is the model of the D flip-flop with the clock enable, and asynchronous reset inputs.

```
entity DFFE is port(C:in bit; -- clock
    D,R,CE: in bit; -- data, reset, clock enable
    Q: out bit);
```

```

end entity;
architecture beh of DFFE is begin
  process(C,R) begin
    if R='1' then
      Q<='0';          -- reset
    elsif C='1' and C'event then
      if CE='1' then   -- clock enable is separated from the
        clock condition
          Q<= D;      -- data loading
        end if;
      end if;
    end process;
  end beh;

```

Consider the design of the 8-bit shift register based on this FF entity, which is functionally equal to one in Fig.1.32. The respective VHDL description is

```

entity RGS is port(C:in bit; -- clock
  R: in bit; -- reset
  SR,SL: in bit; -- shift right, shift left
  DI: in bit; -- input data
  Q: out bit_vector(7 downto 0));
end entity;
architecture beh of RGS is
  component DFFE is          -- component declaration
    port(C:in bit;          -- clock
      D,R,CE: in bit; -- data, reset, clock enable
      Q: out bit);
  end component;
  signal D,Y: bit_vector(7 downto 0); -- intermediate data
  signal E: bit;                -- FF enable
begin
  D(0)<=DI when SL='1' else '0'; -- multiplexor of LSB
  D(7)<=DI when SR='1' else '0'; -- multiplexor of MSB
  MUX:for i=1 to 6 generate     -- multiplexers for register inputs
    D(i)<=Y(i-1) when SL='1' else
      Y(i+1) when SR='1' else '0';
  end generate;
  E<=SR or SL;                 -- logic of the register clock enable
  RG:for i=0 to 7 generate      -- 8-bit register
    U_FF:DFFE(C=>C,R=>R,CE=>E,D=>D(i),Q=>Y(i));
  end generate;
  Q<=Y; -- register output
end beh;

```

In the declarative part of the architecture description the FF component, and intermediate signals are declared. In the behavior description part the trigger entity is instantiated in the network by the component instantiating operator.

This instantiation with the label RG is implemented 8 times with the different index  $i$  by the operator **generate**. Here the named binding of ports and signals is used. The respective associative binding is

```
U_FF:DFFE(C,R,E,D(i),Y(i));
```

The named binding is preferable, because it provides less errors, and good readability of the description. Moreover, in such a binding the order of ports in the list can be variable. The operator **generate**, which is labeled by MUX, expands 6 times the operator, which describes the multiplexor at the  $i$ -th trigger input. Due to the increment and decrement of indexes in assignments, for example,  $D(i) \leq Y(i-1)$ , shifts right and left are implemented.

The counters can be described in VHDL as the set of FFs with the respective LN. But the language provides more effective tools to do this. Consider the counter, which is similar to one in Fig.1.36. This counter is described as:

```
library IEEE;
use IEEE.numeric_bit.all;
entity CT5 is port(R,CI: in bit; -- clock, reset, count enable
                  Y: out bit_vector(2 downto 0));
end entity;
architecture beh of CT5 is
  signal Q: unsigned(2 downto 0); -- state of the counter
begin
  process(C,R) begin
    if R='1' then
      Q<="000"; -- reset
    elsif C='1' and C'event then
      if CI='1' then -- count enable
        if Q= 4 then -- state A
          Q<="000"; -- state C
        else
          Q<=Q+1; -- direct counting
        end if;
      end if;
    end if;
  end process;
  Y<=bit_vector(Q);
end beh;
```

In the first rows the library IEEE, and its package numeric\_bit are attached to the project. This package defines a set of types and functions, which are useful to operate with bit vectors as with the integer numbers. The subtype means unsigned that the respective bit vector is considered as the positive integer number without a sign. The state of the counter is declared as unsigned, and therefore it provides the increment operation  $Q+1$  and comparing with integer  $Q = 4$ . As a result, the counter behavior is described rather shortly and clear. To output the counter state to the output port the near type conversion  $Y \leq \text{bit\_vector}(Q)$ ; is used.

## 2. Memory units

### 2.1 *General properties*

All the computing systems require the ability to "remember" the values of binary variables. This is accomplished by using memory cells to store the variable and then recall it as needed. These cells are found in a register, or as large arrays, named **memory units** (MUs), that can store millions of bits of data.

Digital systems employ different types of MUs whose characteristics vary with the application. One classification scheme is based on which operations are provided by the cell design. A read/write memory is one where the user may store values, hold them for an indefinite period of time, and read them out as needed. It is usually called as **random-access memory** or RAM. In a **read-only memory** (ROM), the information is permanently stored in the device before it is used in the electronic system. A user may read the information out of a ROM but is not permitted to change the data. A variation of this is the **programmable ROM** (PROM) where the user may store the desired data, but the write procedure requires a special electronics setup and is performed in a few times.

The MU parameters depend on each other. The MU volume increasing forces its cost and delay increase. Therefore, in modern computers MUs are used, which have different volume, speed, and they form the hierarchical system. The main data storage in the computer is a RAM. This RAM is based on dynamic (**DRAM**) or static (**SRAM**) memory ICs. The memory IC cost decreases approximately in 30% per year. DRAM is usually in 5 times cheaper than SRAM of similar volume. Its energy consumption is less approximately in 4 times as well. But usually the SRAM speed is in 2-3 times higher than one of DRAM. Modern synchronous DRAM (**SDRAM**) combines in itself the high volume DRAM cell array and high speed SRAM buffer. Due to the pipelined burst mode it provides the average access time less than 5-7 ns.

There is a tendency to increase the RAM speed in two times per 5 years. But due to the Moore's law, the twofold increase of the CPU speed occurs in two years. This forces the increase of the margin between the CPU and RAM speeds. The compromise solution was found, which consists in the use of small volume but fast speed intermediate MU, where the frequently used data were stored. This MU is named as the **fast memory** (FM). The representative of FM is cache-RAM, which stores recently used pages of the virtual memory. Because the FM speed depends not only on its technology, but on the distance to CPU, the most effective way is to place FM near CPU in a single chip. The volume of such a FM is limited by the chip technology, and is less than ca. 256 kbytes.

To increase the FM effectiveness, it is arranged as hierarchical FM. FM, which is integrated in the CPU chip, forms a first level FM. FM, which is placed near CPU, forms a second level FM. Often a third level FM is used. When CPU needs new data and it doesn't find them in a first level FM, then this data is rewritten from a second level FM to a first level FM, if any. Otherwise, this data is found in a third level memory. To increase the data transfer speed, FMs of different levels usually operate in parallel. When a portion of data is read from the first level FM then another portion is rewritten simultaneously from the second level FM.

Usually instructions and data are placed in different areas of the memory. After reading, they are loaded into different parts of CPU: instructions come to the control unit, and data enter ALU. Therefore, often FM is divided into two parts of data FM and instruction FM. And such a division allows for reading both data and instructions in parallel.

The large data arrays are stored in the non-volatile outer MU (**NVRAM**), which have large volume ( $10^{12} - 10^{16}$  bytes) but relatively small speed. They often are based on magnetic discs, and sometimes on tapes. Many recent NVRAMs in portable devices like the memory sticks are based on flash EEPROM. The hard disc drivers (**HDD**), named winchesters, are the most popular NVRAMs. Their access time is 1–10 ms, and data transfer speed is 2–40 Mbytes/s. To adjust the FM speed and speed of NVRAM it is arranged by the buffer memory of middle volume and middle speed, named HDD cache.

The system of all MUs, which is used by the CPU, is named as a computer memory. The multilevel computer memory can be considered as a virtual memory. Each level of such a memory is arranged by a special control unit, which provides automatic data transfer between memory levels. This control unit usually uses some strategy, which minimizes the average data access time. By the proper access strategy the virtual memory behaves as MU with the volume of NVRAM ( $\sim 10^{13}$  bytes) and the access time of FM ( $\sim 10$  ns).

## 2.2 *Fast memory units*

The access time of FM is much less than one of the usual RAM. FM is added to CPU to minimize the stream of accesses to RAM, or the average RAM access time. It increases the CPU speed, because it depends on the MU speed. Besides, the data access in FM, and operations in ALU can be implemented in parallel.

FMs with direct, associative, pipeline and stack addressing modes are distinguished. By the **direct addressing**, the FM cells have the addresses from 0 to  $m-1$ . The cell addresses are placed directly in the address field of the instruction. When  $m$  is a small number, FM is usually called as a **register file**. To define the register file address a small bit number is distinguished, for example, 4 by  $m=16$ . In the direct addressing mode, the programmer or compiler has to optimize the cell loading to increase the CPU speed, and this is a complex task.

In the **associative addressing mode**, the operand has not an address but a tag or a set of tags. When the tag is input in the associative FM, then it outputs one or several words with equal tags, or nothing, when tags mismatch. This addressing mode is used in the cache memories, which are discussed below.

Two processing units often are connected through the buffer FM, with the **pipeline addressing mode**. Such FM is considered as a set of registers, which are connected in a pipeline. The source unit pushes the data in FM, and the destination unit reads the pulled data in the very order, in which the data have entered FM. This mode is named as first in – first out (**FIFO**). Hence, such a FM is often called as FIFO. This FM is usually used when the source unit outputs the data with unstable time intervals, for example, in the communication systems.

In the **stack addressing mode**, FM is considered as a register stack. I.e. the user has access only to a stack top, and can push the data to it or pull the data from it. Hence, the read data is the data, which was pushed in the stack last time, and this mode is called as least recently used (**LRU**).

### 2.3 Register file

The CPU provides a way to store data words that can be easily used as inputs into its arithmetic and logic unit (ALU). These storage locations are made up of several registers that are wired into the datapath in a convenient manner. Such a group of registers is called as register file.

The register file can have a single input and output as the usual RAM has. But to increase the datapath throughput, the register file has at least one writing channel and one or two reading channels, which are associated with ALU operands. For example, the three-port FM provides loading of two operands of the instruction in ALU and storing the result in a single clock cycle, which provides the three fold increase of the speed. It should be mentioned that in FM with two write ports the conflict situation can occur. When the writing to two channels is provided, and equal addresses are used simultaneously, then the written datum has the undefined value.

Consider the register file of 16 registers with two reading channels *A* and *B*, and writing channel *Q*. Its structure diagram is shown on Fig.2.1. By the addresses *AB* and *AD* the multiplexers MUXB and MUXD select one of 16 registers for reading. To write the data by the address *AQ* = *i*, and writing enable signal *WE*, one of the decoder DC outputs enables the clock signal *CLK* to the *i*-th register.

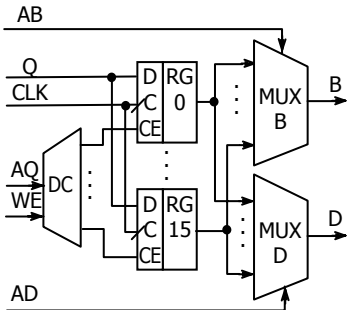


Fig.2.1

Such FM as well as RAM can be modeled in VHDL, and then synthesized in ASIC as a set of triggers, which are addressed by the proper decoders and multiplexers. But the FM description is more clear, when instead of `bit_vector` type the array type is used. Consider the mentioned above register file with the data width of 8 bits. Its VHDL description looks like the following.

```
Library IEEE; use IEEE.numeric_bit;
entity FM16 is port(CLK,WE: in bit; -- clock and writing enable
  AB,AD,AQ: in bit_vector(3 downto 0); -- 4-bit addresses
  Q: in bit_vector(7 downto 0); -- input data
  B,D: out bit_vector(7 downto 0)); -- output data
end FM16;
architecture BEH of FM16 is
  type TRAM16 is array (0 to 15) of bit_vector(7 downto 0); --register array type
  signal RAM:TRAM16:=(others=>"00000000");-- array of 16 cells is initialized by zeros
begin
```

```

RAM_16:process(CLK,WE,ADDR) -- process, which describes RAM
  variable addrb,addrd,addrq:natural; -- intermediate address variables
begin
  addrb:=To_integer(Unsigned(AB));-- bit vector is transferred to a natural value
  addrd:=To_integer(Unsigned(AD));-- bit vector is transferred to a natural value
  addrq:=To_integer(Unsigned(AQ));-- bit vector is transferred to a natural value
  if Rising_edge(CLK) then --clock rising edge finding
    if WE='1' then
      RAM(addrq)<= DI; -- data writing by the rising edge of the clock when WE=1
    end if;
  end if;
  B<= RAM(addrb);      -- asynchronous data reading by the address AB
  D<= RAM(addrd);      -- asynchronous data reading by the address AD
end process;
end BEH;

```

In such a manner the RAM of any volume can be modeled and then synthesized. Also the ROM can be modeled as RAM with specific cell initialization and without the writing property. But usually the ROM is given as the array of constants.

## 2.4 Stack memory

The stack FM can be implemented on the base of the register file or RAM. Then the operand address is derived from the previous address by the increment or decrement to a 1. I.e. the addressing is implemented by a counter or by the array moving up or down to a single cell. The FM structure with the counter addressing is illustrated by Fig.2.2. When writing (WR) of a word to the memory array M, a 1 is added to the CTR content. When reading (RD), CTR is decremented to a 1. The next word is written to a cell with an address, which is in a 1 higher than the previous cell address, and the reading is implemented in the reversed order. Signals "FM is empty" (OF<), "FM is full" (OF>) are formed in the zeroed and in  $m$ -th states of CTR. The error signals ER< and ER> indicate the reading of empty FM and FM overflow.

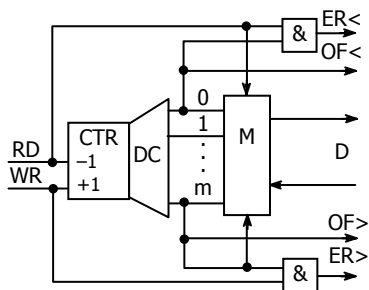


Fig.2.2

## 2.5 Cache memory

The automatic exchange of data between FM and RAM is achieved in the associative addressing mode. The FM structure with such a mode is shown on Fig.2.3. It contains information MU (M), associative MU (CAM), control unit CU, address register RGA, and data register RGD. Each data stored in M has a **tag**, which is the data address, and which is stored in CAM. When addressing FM, in

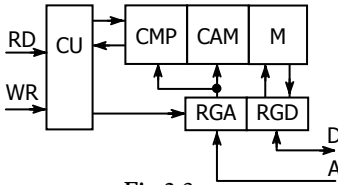


Fig.2.3

RGA the address is written, and signals of reading RD and writing WR are entered in CU. The address is compared with the addresses in CAM by a set of comparators CMP. If the address is equal to some in CAM then the access is said to **hit**. In this situation the proper datum is read from M to RGD or written from RGD to M. If the

address is not equal to any in CAM, the access is said to **miss**. Then CU organizes the access to the outer RAM. The datum read from RAM is written to RGD and to the empty cell of M, and its address is written to the respective cell of CAM. The next reading of this datum is fulfilled from FM but not from RAM. When the writing operation is implemented then the datum is written both in M and in outer RAM.

When FM is in operation, all the cells of FM are full. Therefore, to access a new address one of the cells has to be released. It is natural to **release** a cell that is accessed less times. To distinguish such a cell CU contains a network, which selects such a cell considering some strategy. The simplest strategy is the following. Consider triggers  $T_i$ , which are tagged to  $i$ -th cells. At the beginning,  $T_i = 0$  for all  $i$ . If access to the  $i$ -th cell hits, then  $T_i = 1$ . This means that at least one access to this cell has occurred. If access to FM misses, then the first selected cell is flushed, which  $T_i$  contains a 0. When at this moment  $T_i = 1$  for all  $i$ , then  $T_i = 0$  for all  $i$ , except one, which is selected for this access. To simplify the assignment of the released cell, the randomized selection of the cell or the selection due to the round robin rule is implemented.

The disadvantage of the associative FM consists in its hardware complexity when its address volume is high. Its hardware effectiveness is increased when the address-associative mode is used. Consider the example of the 8 kbyte cache RAM of the i486 CPU. It contains 32-bit address bus and 4-byte data bus. One associative cell contains four 16-byte rows of data and a 91-bit tag (see Fig.2.4). The memory unit M contains only 128 associative cells.

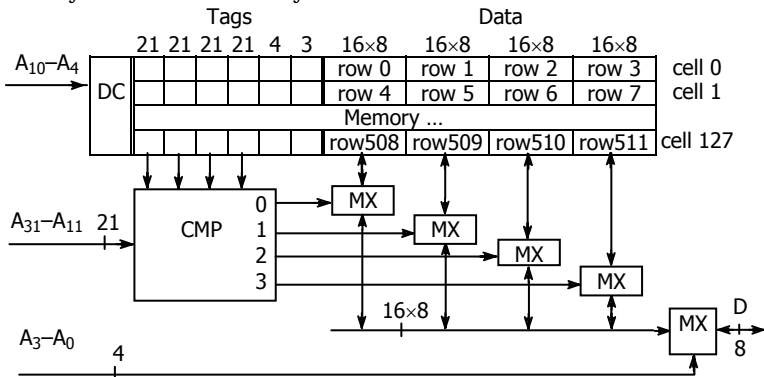


Fig.2.4



The tag consists of four 21-bit address fields, 4 bits of the row correctness code and 3 bits, which are used to find out the cell for releasing. 4 lowest address bits select a single byte of a row. 7 middle address bits select the associative cell. And the rest of them are the address code, which is compared to the proper 21-bit tag field. When the access occurs, 7-bit address field selects a single associative cell; four 21-bit comparators compare highest address bits with the proper tag fields. If the comparing succeeds, and the data correctness bit is a 1, then the datum is accessed. Otherwise, in the selected cell a row is released, to which the access did not occur for a long time, and the datum from the outer RAM is written in it. Simultaneously the respective tag field is corrected. We see that such a cache RAM is based on the usual RAMs and on only four-address comparators.

Many microprocessor cache RAMs have the similar structure. They are distinguished by MU volume, associative cell number, row length, etc. When the access is missed, not a single byte is exchanged but a whole data row. Therefore, if the address sequence is a randomized, then the average access time can decrease in many times. But in most cases the address sequence is the incremental one, for example, the program sequence. Therefore, the next instructions occur in the same row, and cache misses happen more rarely.

## 2.6 *Memory integral circuits*

The MU can be a part of CPU or system on the chip. Here the memory cell is usually implemented as a trigger. As a result, such a MU could not have large volume (more than  $\sim 10^5$  of bytes), and it consumes large power. This is the reason that large volume MUs are usually manufactured as the separate ICs. Due to the specific technology, special design of memory cells, read-write amplifiers, decoders and multiplexers, such MUs have the minimized power consumption, high speed and the volume up to  $\sim 10^8$  of bytes. The memory ICs are divided to SRAM, DRAM and EEPROM due to their technology and properties.

RAM stands for random-access memory, which means that any word in it can be accessed in the same amount of time as any other word. The term static RAM (SRAM) means that once data is stored in the RAM, the data remains there until the power is turned off. This is in contrast with a DRAM, which requires that the memory be refreshed periodically to prevent the data loss.

SRAMs are available that can store up to  $\sim 10^7$  bytes of data. For illustration, we describe a CMOS SRAM that can store 2 K bytes of data. But the principles illustrated here also apply to large SRAMs. Fig.2.5 shows the block diagram of this SRAM. This MU has 16384 cells, arranged in a  $128 \times 128$  memory matrix M. The 11 address lines are divided into 2 groups. Lines  $A_{10}-A_4$  select one of the 128 rows in the matrix using the row decoder DCR, which outputs are word lines  $WLi$ . Lines  $A_3-A_0$  select 8 columns in the matrix at a time, since there are 8 data lines. The selection is performed by the bidirectional column I/O multiplexor MXCIO. The matrix data outputs go through tristate buffers before connecting to the data I/O pins. These buffers are disabled except when reading from MU.

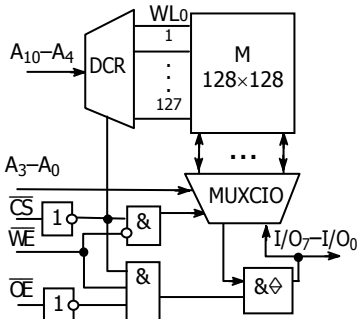


Fig.2.5

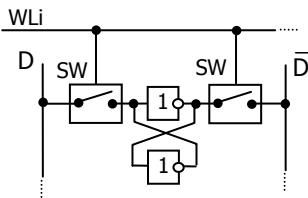


Fig.2.6

Table 2.1

| CS | OE | WE | Mode            | I/O pins |
|----|----|----|-----------------|----------|
| 1  | X  | X  | not selected    | Z        |
| 0  | 1  | 1  | output disabled | Z        |
| 0  | 0  | 1  | read            | data out |
| 0  | X  | 0  | write           | data in  |

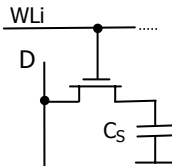


Fig.2.7

A SRAM cell (Fig.2.6) that stores one bit of data is constructed by embedding the cross-coupled inverters in a large network that allows us to set or reset the cell as a latch. The access switches (SW) are controlled by the  $i$ -th word line  $WLi$ , which is decoded by the row decoder. When  $WLi = 0$  the cell is isolated from external influences and holds the bit consuming the smallest leakage power. If  $WLi = 1$  then both switches are closed, which connect the bit line  $D$  and bit-bar line  $\bar{D}$  to opposite sides of the cell. This allows us to writing to, or reading from, the cell. The lines  $D$  and  $\bar{D}$  are connected to MUXCIO. The simplest cell is formed by 4 CMOS transistors of inverters and 2 transistors of switches. This provides the small hardware volume of SRAM. In a dual-port SRAM the switches, bit lines and bit-bar lines are doubled, as well as RDC and CIO are.

The truth table for the RAM (Table 2.1) describes its basic operation. Z in the I/O column means that the output buffers have high impedance outputs, and the data inputs aren't used.

In the read mode, the address lines are decoded to select 8 of the cells, and the data comes out on the I/O pins. In the write mode, input data is routed to the latch inputs in the selected cells when  $WE = 0$ , but writing to the latches in the cells is not completed until either  $WE = 1$  or the chip is deselected.

A DRAM array is similar to an SRAM array in that it allows us to store data using the concept of cell addressing. The difference between the two types of MU is in the internal design of the cells themselves. The circuit schematic for such a cell is shown in Fig.2.7. The cell consists of a single FET transistor and a storage capacitor  $C_s$ . This allows for a very high integration density and makes it possible to create a single chip that has up to  $\sim 10^9$  of cells.

When  $WLi = 1$ , the FET acts as a closed switch allowing a write or read operation. A hold state is obtained by bringing the word line to  $WLi = 0$ , shutting off the direct conduction path between the data line and the storage capacitor. But closed FETs admit a small leakage current that removes charge from the capacitor. This is an explanation that the data bit can only be held for a short period of time ( $< 100$  ms). The data must be periodically updated to insure that it is valid. This is called a **refresh** operation. It is performed by readout of the data,

amplifying it, and then writing it back into the cell. In modern ICs the row selection for reading forces the refresh of the whole row of cells. The refresh circuitry is included on the chip and makes it appear that MU has long-term retention characteristics. Refresh rates are on the order of a few kilohertz. For a single refresh clock one row of cells is refreshed. To refresh the whole DRAM, all the rows have to be traversed. Therefore, a thousand of rows are traversed with the speed of at least a few megahertz.

The difference in row and column addressing is usually utilized in all DRAM devices. Consider the smallest DRAM device of the volume of 65 K bits, which drawing symbol is shown on Fig.2.7. Its truth table (Table 2.2) shows the DRAM mode depending on control signals. The waveforms in Fig.2.8 illustrate its usual operation. *RAS* and *CAS* are acronyms for the row address select and column address select signals. When  $\overline{RAS} = 1$ , the circuit stores the data. When  $\overline{RAS} = 0$ ,  $\overline{CAS} = 1$ , the row address bits, which are set in the address bus, are latched in the inner address register. Simultaneously all the cells of the respective row are refreshed. When  $\overline{CAS} = 0$ , the column

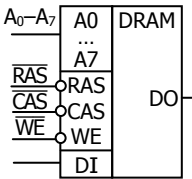


Fig.2.7

Table 2.2

| <i>RAS</i> | <i>CAS</i> | <i>WE</i> | Mode                         |
|------------|------------|-----------|------------------------------|
| 1          | X          | X         | not selected                 |
| 0          | 1          | X         | row address latched, refresh |
| 0          | 0          | 1         | read                         |
| 0          | 0          | 0         | write                        |

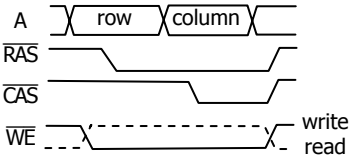


Fig.2.8

address bits, which are set in the address bus, are decoded as well as the row address bits do. Simultaneously the writing or reading operation is implemented depending on the *WE* signal, which is finished when  $\overline{CAS} = 1$ .

As we see, the address in DRAM is loaded in two cycles. Due to this property, the pin number of this IC is much less than this number for SRAM. The access period is often minimized when the row address is latched a time, and the cells of a single row are accessed using only the column address exchange. The disadvantage of such a mode consists in the need of the outer address multiplexor, which forms the row and column address bits. Nevertheless, modern SOC and microprocessors, which access the DRAM devices, usually have the special DRAM controllers, which provide both address control and refresh control, not to say about timing control, IC initialization, and control bit checking. Therefore, the DRAM disadvantages are "invisible" for the user.

Due to the deep address decoding networks, slow column data amplifiers and two-cycle access mode, DRAMs usually had the access time of tenths and hundreds of nanoseconds. Modern DRAMs provide the high-speed access, thanks to a set of improvements. A set of memory banks is placed in IC, which provide the access in parallel. The whole address is the concatenation of row, column and bank addresses. The DRAM ICs have usually bidirectional data bus with the width up to 36 bit. To access the parts of this data (bytes, nibbles), the data mask bits (DQM) are used. To provide the automatic refresh mode, DRAM has the

refresh counter, which is attached to the row address register.

In SDRAMs the network is pipelined, i.e. the input data, address, output data and even the read row data are synchronously stored in the buffer register networks for a clock cycle. Therefore, the read operation latency reaches 2–4 clock cycles. But the clock frequency increases up to hundreds of megahertz.

Besides, if a set of similar operations is performed, for example, reading sequence, then a single access can occur in a single clock period. The **fast page mode** (FPM) or burst mode serves to this process. In FPM, the row address is latched, and the access is performed for different column addresses. To provide the **burst mode**, the column address register is implemented as the fast speed counter. Therefore, when CPU writes or reads a set of data to incremented addresses, the burst mode is usually used.

To control the fast page mode correctly, its timing is programmed in the DRAM controller by four figures. For example, the set 5-1-1-1 shows that to implement the first access 5 clock cycles are need, and to access the second, third and fourth data, it takes a single clock cycle. The DRAM controller has to consider that DRAM operates in different modes (initialization, read, write, read burst, write burst, no operation, refresh, load mode and others), which need the proper timings. The *RAS*, *CAS*, *WE*, *DQM* signals are considered as a command word and are stored by a clock edge. In the load mode, the address bits are considered to be the control word, which controls burst length, burst type, access latency, etc.

CMOS technologies provide for a large variety of ROM circuits to be manufactured. Although most of them allow user data to be entered. **EPROM** is an acronym for Erasable-Programmable ROM. Programming in it is achieved by a process in which a high voltage is used to transfer charge to a "floating" capacitor of a memory cell. The capacitor is usually implemented as the gate of a MNOS transistor. When the charge is present, then the transistor is open. The charge is trapped on the capacitor and it cannot escape under normal circumstances. In this type of devices, erasure is achieved by placing the device under an ultraviolet light source, and keeping it there for several minutes. Now these devices have been replaced by ones that can be erased electrically.

The electrically erasable EPROM (**EEPROM**,  $E^2$ PROM) has the advantage that the data may be erased using electrical circuitry and does not require that the chip be physically removed from the system. To erase the cell, the capacitor voltage is reversed and the charge moves in the opposite direction. New technology allows us that a large number of cells can be erased at a time. These devices are called **flash EPROMs**, with "flash" referring to the speed at which the array may be erased.

The ferroelectric RAM (**FRAM**) absorbs in itself the advantages of DRAM (high volume, fast speed) and NVRAM (data storing after power off). Its cell circuit resembles the one of DRAM. The datum in it is stored not as a charge but as a polarization sign of the cell capacitor. For this purpose, the capacitor is made of the ferroelectric insulator. When reading, if the capacitor has changed its polarization, then a 1 is considered to be read. But at this process, the polarization is reversed, i.e. a 0 occurs in it. Therefore, the reading process is finished by storing the read data back.

# 3. Networks for arithmetic and logic operations

## 3.1 Arithmetic and logic units

In the first chapter, we discussed the design of networks for arithmetic and logic operations like adder, subtractor, AND gates, shifter, etc. In the network for arithmetic and logic operations of CPU or application-specific processor these units are usually combined in a single unit, named as an **arithmetic and logic unit** (ALU). Usually the ALU has two  $n$ -bit input data busses  $A$ ,  $B$  and an  $n$ -bit result bus  $D$ . The  $k$ -bit control word  $F$  can set one of  $2^k$  arithmetic or logic operations. Additional input  $CI$  serves as the carry bit input. One or more flag outputs are used for signaling carry output  $C$ , zero result  $Z$ , negated result  $N$ , or overflow  $V$ .

The simplest way to design an ALU is to merge the outputs of the separate units (adder, AND gate array, etc) by a multiplexor, which is controlled by the word  $F$ . Such a network has potentially the maximum speed. But its hardware volume is too high. Another approach consists in the synthesis of the multifunctional LN, which contains  $n$  equal stages, may be, except first and last one. And each stage, named a bit slice, represents the Boolean function of operand bits  $A_i$ ,  $B_i$ ,  $C_i$ , and control word  $F$ . Sometimes the additional inputs and outputs are needed, for example, to provide the shift operations.

Consider the design of a simple 8-bit ALU, which we name as a multipurpose summator (**LSM**). It implements two arithmetic functions – addition and subtraction – and three logic bit-wise functions – a 1 output, AND and Exclusive OR. The output flags are carry  $CO$ , negative result (sign)  $N$  and zeroed result  $Z$ . Let the design is based on 4-input PLA cells. It can be implemented on 4-input LUTs as well.

Firstly, the truth table for the outputs of the  $i$ -th bit slice of LSM is composed (see Table 3.1). In three sets of the control code  $F$  the function  $D_i$  is undefined and it is marked by "X". The subtraction is implemented as an addition with negated operand plus a 1 (see the chapter 1.8).

The idea of the LSM synthesis consists in the following. The output function can be decomposed and be represented by the intermediate results  $X_i$ ,  $Y_i$  and  $C_{i+1}$ . The resulting network is searched as the network in Fig.3.1. The output of the unit LNO implements the addition modulo 2 of  $X_i$ ,  $Y_i$  and  $C_i$ , i.e.  $A_i \oplus B_i \oplus C_i$ . Units LNX, LNY, and carry circuit LNC generate the proper operands  $X_i$ ,  $Y_i$ ,  $C_{i+1}$ , depending on the code  $F$ . The modulo 2 func-

Table 3.1

| $F_0F_1$ | $F_2 = 0$                         |  | $F_2 = 1$        |
|----------|-----------------------------------|--|------------------|
|          | $D_i$                             | $C_{i+1}$                                    | $D_i$            |
| 0 0      | 1                                 | X  | X                |
| 0 1      | X                                 | X  | $A_iB_i$         |
| 1 0      | $A_i \oplus \bar{B}_i \oplus C_i$ | $A_i\bar{B}_i \vee A_iC_i \vee C_i\bar{B}_i$ | X                |
| 1 1      | $A_i \oplus B_i \oplus C_i$       | $A_iB_i \vee A_iC_i \vee C_iB_i$             | $A_i \oplus B_i$ |

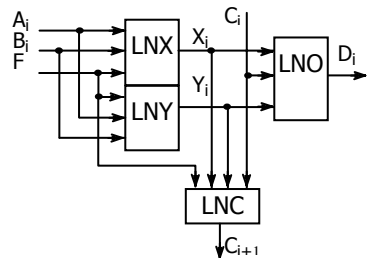


Fig.3.1

tion of 3 bits is (see the chapter 1.2):

Table 3.2

| $F_0F_1$ | $F_2 = 0$ |             |                                  | $F_2 = 1$ |                |           |
|----------|-----------|-------------|----------------------------------|-----------|----------------|-----------|
|          | $X_i$     | $Y_i$       | $C_{i+1}$                        | $X_i$     | $Y_i$          | $C_{i+1}$ |
| 0 0      | $A_i$     | $\bar{A}_i$ | 0                                | $A_i$     | X              | X         |
| 0 1      | $A_i$     | X           | X                                | $A_i$     | $A_i\bar{B}_i$ | 0         |
| 1 0      | $A_i$     | $\bar{B}_i$ | $X_iY_i \vee X_iC_i \vee C_iY_i$ | $A_i$     | X              | X         |

|       |   |          |       |             |       |             |                |          |             |
|-------|---|----------|-------|-------------|-------|-------------|----------------|----------|-------------|
| $F_0$ | X | $B_i$    | $B_i$ | $\bar{B}_i$ | $F_0$ | $\bar{B}_i$ | $B_i$          | $B_i$    | $\bar{B}_i$ |
|       | X | $A_iB_i$ | X     | $\bar{A}_i$ |       | $\bar{A}_i$ | $A_i\bar{B}_i$ | $A_iB_i$ | $\bar{A}_i$ |
|       |   | $F_1$    |       |             |       | $F_1$       |                |          |             |

a)                      b)

Fig.3.2

$$D_i = \bar{X}_iY_i\bar{C}_i \vee X_i\bar{Y}_i\bar{C}_i \vee \bar{X}_i\bar{Y}_iC_i \vee X_iY_iC_i.$$

From this function we can derive the Exclusive OR function  $X_i \oplus Y_i$ , when  $C_i = 0$ . Now we can build the truth tables of the functions  $X_i$ ,  $Y_i$ , and  $C_{i+1}$  (Table 3.2). The "don't care" states of  $C_{i+1}$  by  $F_2 = 1$  can be defined as zeros. Then  $C_{i+1} = \bar{F}_2(X_iY_i \vee X_iC_i \vee C_iY_i)$ , which is a function of 4 arguments. The value  $C_{i+1} = 0$  by  $F = 000$  is formed when  $C_i = 0$  and  $Y_i = 0$ .

It is useful to do without LNX

unit, i.e. this unit outputs always  $X_i = A_i$ , and when  $C_i = 0$  we have  $D_i = A_i \oplus Y_i$ . When arithmetic operations are done, the function  $Y_i$  is equal to  $B_i$  for addition and to  $\bar{B}_i$  for subtraction. To derive the function  $Y_i$  for AND and a 1 operations, we have to solve the following equations:  $A_i \oplus Y_i = A_iB_i$  and  $A_i \oplus Y_i = 1$ . The solution of the first equation is  $Y_i = A_i\bar{B}_i$ , because  $A_i \oplus A_i\bar{B}_i = A_i \oplus A_i(1 \oplus B_i) = A_i \oplus A_i \oplus A_iB_i = A_iB_i$ . The solution of the second equation is  $Y_i = \bar{A}_i$  because  $A_i \oplus \bar{A}_i = 1$ . These solutions are put in the Table 3.2. The Boolean function  $Y_i$  is represented by KM, which is shown in Fig.3.2 (a). Fig.3.2 (b) illustrates this KM, but with assigned "don't cares". The resulting function is  $Y_i = \bar{F}_0F_1A_i\bar{B}_i \vee F_0F_1B_i \vee F_0\bar{F}_1B_i \vee F_0\bar{F}_1\bar{A}_i$ .

The function  $C_0$  has to be a 1, when subtraction, and a 0, when another functions. If it is equal to the input signal  $CI$ , then it can be given the proper value by outer networks. Carry flag is  $CO = C_8$ , sign flag is  $N = D_7$  and zero flag is  $Z = (D_7 \vee D_6 \vee D_5 \vee D_4) \vee (D_3 \vee D_2 \vee D_1 \vee D_0)$ . The synthesized ALU network is shown in Fig.3.3. Its hardware volume is equal to 27 PLA cells or LUTs.

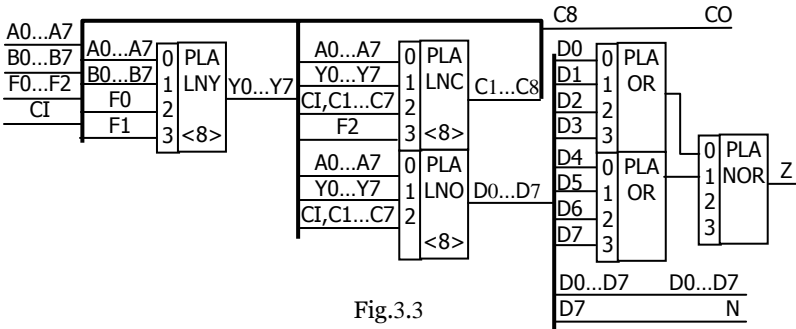


Fig.3.3

The speed of LSM is found from the longest path. It contains seven PLA cells, which form  $C_8$ , one cell to calculate  $D_8$ , and two cells to derive  $Z$ . The amount delay is equal to 10 delays of PLA cells.

This LSM can be easily described by VHDL as the following

```
C<=1 when CI='1' else 0;
with F select
    T<="011111111" when "000",      -- a 1
    RESIZE(A,9) + not B + C when "010",-- subtraction
    RESIZE(A,9) + B + C when "011",   -- addition
    "0"&(A and B) when "101",         -- AND
    "0"&(A xor B) when others;        -- XOR
D<=T(7 downto 0);
CO<=T(8);
N<=T(7);
Z<=not(T(7) or T(6) or T(5) or T(4) or T(3) or T(2) or T(1) or T(0));
```

Here signals  $A, B, D, T$  are of type signed, which is declared in the IEEE.Numeric\_Bit package. They support both logic and arithmetic operations, and synthesis of respective logic networks. The temporary signal  $T$  is the bit vector of the length 9. Its most significant bit (MSB) serves only to derive the carry bit  $CO$  after addition. To provide the vector width match by the signal  $T$  assignment, the resize function RESIZE and concatenation (function '&') with zero bit are used. The assignment depending on the code  $F$  is implemented as the selective parallel signal assignment (**with...select**). The "don't care" meanings in it are assigned in the **when others** clause.

This program piece can be inferred by the synthesis compiler as the adder-subtractor and the logic circuit, which are coupled by a multiplexor. Hence, the hardware volume of the derived network is much higher than one of the network, shown in Fig. 3.3. This example shows the advantage of the manual synthesis.

### 3.2 Datapath

The main CPU units are the control unit and the datapath. The control unit provides fetching and decoding the instructions, and outputting the respective control signals. The **datapath** circuits provide the logic for every instruction that can be performed by CPU. In general, the datapath network can be broken down into three main groups of circuits: the register file, the ALU and the local MU. The register file is a group of general-purpose registers, which are used to store data words for use in the current chain of calculations. ALU provides all of the arithmetic and logic functions. The local MU serves as a cache MU. It is included in CPU to provide fast read and write operations that will not slow down the CPU operation. The register file and cache MU were discussed in chapters 2.2, 2.4.

Consider the simplest datapath, which contains only register file (FM) and ALU. The block diagram of the datapath, which utilizes the three-channel FM, is shown in Fig.3.4. The block diagrams of its FM and LSM are shown in Fig.2.1, and

Fig.3.3, respectively. The access to such a FM needs three addresses  $AB$ ,  $AD$ , and  $AQ$ . The datapath is connected to the outer space (I/O ports, RAM) through the busses  $DI$  and  $DO$ . Through the bus  $DI$  the data is loaded into FM, and through the bus  $DO$  the result is outputted using the load and store instructions.

To achieve the algorithmic completeness, the ALU needs the shift operation implementation. The left shift can be implemented by addition of the same operand ( $B+B=2B$ ). To shift right a shifter  $SHU>$  to a single bit is needed. If it is synthesized as one in the chapter 1.7, then it is represented by the two-input multiplexor, which is controlled by the bit  $F>$ . The shifter has the shifted in bit input  $QI$  and shifted out bit output  $QO$ .

When the two-channel FM is used, then only two address codes are needed. Then the register-accumulator (AC) is of demand, which stores the second operand. Two methods of AC switching are possible: at the LSM input (Fig.3.5) and at the LSM output (Fig.3.6). The second method is preferable, because both PLA and FPGA cells have the structure, in which the logic circuit result is stored in a trigger. Therefore, AC can be mapped together with the logic circuits of  $SHU>$  and LSM, and this minimizes both hardware volume and signal delays.

The simplest operations like addition, shift, logic operations are performed for a single clock cycle. And the complex operations like multiplication, division are calculated by the subprograms. In the RISC processors these subprograms are formed by the processor instructions, and in the CISC processors they are usually microprograms. In the second situation, the signals  $AB$ ,  $AD$ ,  $AQ$ ,  $WE$ ,  $F$ ,  $CI$ ,  $F>$ ,  $QI$  usually form the control fields of the microinstruction.

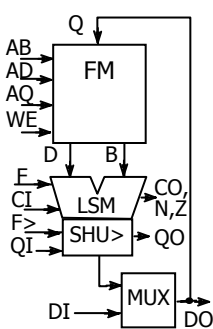


Fig.3.4

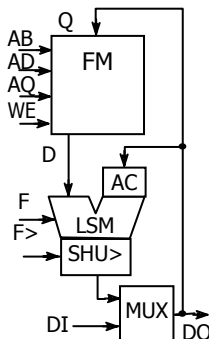


Fig.3.5

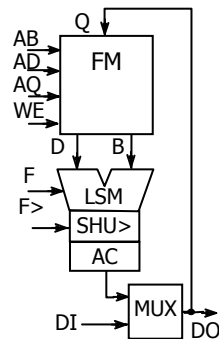


Fig.3.6



### 3.3 Binary multipliers

The multiplication is the frequently used operation in all the computers. Consider the multiplication of natural values, i.e. unsigned integers  $X$  and  $Y$ :

$$P = YX = Y(x_{n-1}2^{n-1} + \dots + x_12^1 + x_02^0) = 2^{-1}(Y'x_{n-1} + \dots + 2^{-1}(Y'x_1 + 2^{-1}(Y'x_0 + 0)) \dots),$$

where  $Y' = Y2^n$ . That means that the multiplication affords up to  $n$  additions and shifts, and it can be implemented as  $n$  iterations of the cycle  $\Pi_{i+1} = 2^{-1}(\Pi_i + Yx_i)$  by the initial conditions  $\Pi_i = 0$ ,  $i = 0$ . Such an algorithm is named as multiplication, beginning at the **least significant bit** (LSB) of the multiplier and with the shift right of the sum of **partial products**. To implement this algorithm in the network, the multiplicand register RGY, AND network for the bit multiplication  $Yx_i$ , adder SM for addition of  $\Pi_i$  and  $Yx_i$ ,  $2n+1$  bit shift register RGP of partial products  $\Pi_i$ , and shift register of the multiplier RGX are needed. The block diagram of such a multiply unit (MPU) is shown in Fig.3.7.

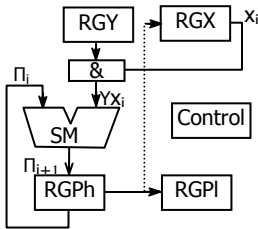


Fig.3.7

Table 3.3

| Cycle | RGX  | $x_i$ | RGY  | RGPh  | RGPl | Operation  |
|-------|------|-------|------|-------|------|------------|
| 0     | 1011 | 1     | 1100 | 00000 | 0000 | initialize |
| 1     | 1011 | 1     |      | 01100 | 0000 | +Y         |
| 2     | 0101 | 1     |      | 00110 | 0000 | shift      |
| 3     | 0101 | 1     |      | 10010 | 0000 | +Y         |
| 4     | 0010 | 0     |      | 01001 | 0000 | shift      |
| 5     | 0010 | 0     |      | 01001 | 0000 | +0         |
| 6     | 0001 | 1     |      | 00100 | 1000 | shift      |
| 7     | 0001 | 1     |      | 10000 | 1000 | +Y         |
| 8     | 0000 | 0     |      | 01000 | 0100 | shift      |
| 9     | 0000 | 0     |      | 01000 | 0100 | end        |

The register RGP consists of the higher  $n+1$ -bit part RGPh and lower part RGPl.

Because RGX and RGPl shift the data in the same direction, and RGX becomes empty in the operation process, then these registers can be combined in a single register RGX (see the dotted line in Fig.3.7). Let us see an example of 4-bit multiplication of 1100 to 1011. The state chart of this process is shown in the table 3.3. It is obviously, that the addition can be combined with the shift in a single cycle. Then RGPh is an usual  $n$ -bit register, the sum is transferred from SM to RGPh with the shift right, and the sum LSB is shifted in RGPl.

This multiplier unit (MPU) structure and its behavior can be considered as the multiplication **scheme**, i.e. the computational model, which represents the multiplication algorithm. Such a scheme can be implemented as a subprogram in the computer. For example, it can be programmed in the datapath, considered in the chapter 3.2. Then RGX, RGY, RGPh are mapped in 3 registers of FM, LSM

makes addition, SHU> shifts multiplier and partial products, bit  $x_i$  is the shifted out bit from SHU>, and it serves as the branch condition.

There are another three multiplication schemes, which are different in beginning at LSB or beginning at most significant bits (**MSB**) of the multiplier and in the shift of partial products or shift of multiplicand. This type of MPU is referred to as a **serial-parallel** MPU, since the multiplier bits are processed serially, but the addition takes place in parallel. Such kind of MPUs, and multiplication schemes were widely used in computers in sixties and seventies. But now they are implemented only in the simplest controllers and some application specific processors. Instead, the **parallel** MPUs are most popular ones, described below.

Consider the 4-bit unsigned multiplication. On the bit level it can be represented as

$$\begin{aligned}
 P = YX = & 2^3x_0y_3 + 2^2x_0y_2 + 2^1x_0y_1 + 2^0x_0y_0 + \\
 & + 2^4x_1y_3 + 2^3x_1y_2 + 2^2x_1y_1 + 2^0x_1y_0 + \\
 & + 2^5x_2y_3 + 2^4x_2y_2 + 2^3x_2y_1 + 2^2x_2y_0 + \\
 & + 2^6x_3y_3 + 2^5x_3y_2 + 2^4x_3y_1 + 2^3x_3y_0 \\
 \hline
 & 2^7p_7 + 2^6p_6 + 2^5p_5 + 2^4p_4 + 2^3p_3 + 2^2p_2 + 2^1p_1 + 2^0p_0
 \end{aligned}$$

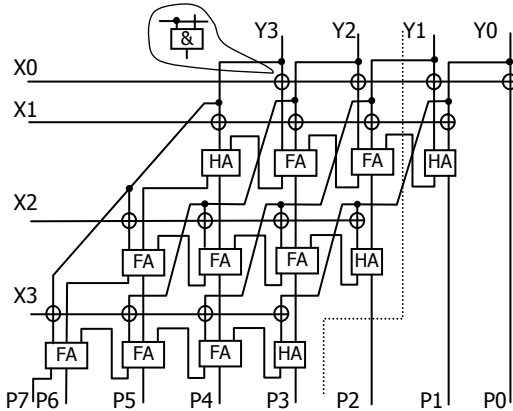


Fig.3.8

It is easily to understand, that each row of this formula except the last one represents a partial product of the multiplicand  $Y$  to the bit of the multiplier  $X$ . Each bit product  $x_iy_j$  can be calculated by a single AND gate. The partial product summation consists in the addition of the bit products with the equal weights (columns) considering the carry bits from the low bits. The network, which calculates this formula, is the parallel MPU, and it is shown in Fig.3.8. To build this MPU,

$n^2$  AND gates,  $n^2 - 2n$  one bit full adders FA and  $n$  half-adders HA are of demand.

The MPU components form an array. For this feature, this kind of MPUs is usually named as the array multiplier. For an  $n \times n$  array multiplier the longest path from input to output goes through  $2n$  adders, and corresponding worst-case multiply time is estimated as  $2nt_A$ , where  $t_A$  is the full adder delay. This means that the array multiplier speed is in 2 times less than the adder speed, considering that the  $n$ -bit adder delay is  $nt_A$ .

After  $n \times n$  multiplication the  $2n$ -bit result occurs. When calculations contain a set of products, then the data width can increase dramatically. To prevent this process, products are usually truncated to  $n$  high bits. In this situation, the array

multiplier can be abridged, that minimizes its hardware volume. In Fig.3.8 the dotted line shows the truncation line, which separates such abridged MPU. To provide the result correctness,  $n+m$  columns of the array are left, where  $m=2, 3, 4$ , depending on  $n$ .

Some algorithms are available for multiplication of signed binary numbers. The straightforward way to carry out such multiplication consists in complementing the multiplier and/or multiplicand if negative, multiplying the two positive binary numbers, and complementing the product if it should be negative. The procedure for multiplying signed 2's complement binary fractions is widely used, which requires only the ability to complement the multiplicand. It is the same as for multiplying positive binary fractions, except that one must be careful to preserve the sign of the partial product at each step. If the sign of the multiplier is negative, the multiplicand should be complemented in the step of multiplying to the sign bit. The hardware is almost identical to that used for unsigned MPU, except a complements unit must be added for the multiplicand. Consider an example of multiplying  $Y = 5/8 = 0.101$  to  $X = -3/8 = 1.101$ . The multiplication steps are shown in the Table 3.4.

Table 3.4

| Data     | Variable  |
|----------|-----------|
| 1.101    | $X$       |
| 0.101    | $Y$       |
| 0.000000 | $\Pi_0$   |
| 0.000101 | $+Y/8$    |
| 0.000101 | $\Pi_1$   |
| 0.0101   | $+Y/2$    |
| 0.011001 | $\Pi_3$   |
| 1.011    | $-Y$      |
| 1.110001 | $P=\Pi_4$ |

In the application specific processors, the hardware multiplier to a constant is widely used. Due to its specific structure, such a MPU provides both high speed and small hardware volume. Usually this MPU is built as an adder tree. It sums the multiplicand, which is shifted to different digit numbers, according to the weights of a 1 in the binary representation of the constant. Therefore, the MPU contains in average  $n/2-1$   $n$ -bit adders. This figure can be decreased to  $n/3-1$ , when both adders and subtractors are used, and when the constant is decomposed to bits 1, 0,  $-1$ . Consider MPU, which multiplies to the fraction  $C=0,110111$ . By subtracting a 1 from LSB, it can be transformed to  $0,11100\bar{1}$ . In such a manner, this result is transformed to  $1,001001$ . Then due to this multiplier representation, the product can be factorized as  $P = XC = X - 2^{-3}(X+2^{-3}X)$ .

The derived block diagram of MPU is shown in Fig.3.9. Here the small black rectangles mean the right shift of the data to the respective bit number. It is easy to prove that the use of such constant decomposition has decreased the adder number from 4 to 2, which shows its high effectiveness.

MPU is easily described by VHDL, because the multiply operator is inferred by the synthesizer as MPU (if the operand types allow the use of this operator). For example, the operator

$$P \leftarrow Y * X;$$

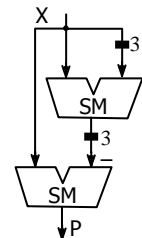


Fig.3.9

represents the MPU, which gives the  $n+m$ -bit product  $P$  from  $n$ -bit multiplicand  $Y$  and  $m$ -bit multiplier  $X$ . If  $X$  and  $Y$  are of unsigned or natural type, then we derive the unsigned MPU, if they are of signed or integer type, then the MPU with

sign will be synthesized. If  $X$  or  $Y$  is a constant, then the compiler will try to synthesize the constant MPU. Some synthesis constraints make the synthesizer to implement the pipelined MPU, which has higher throughput.

### 3.4 Binary dividers

In most of computers, division is implemented as a subprogram, because it is used rarely. In high-end CPUs, some microcontrollers and ASICs the binary dividers are used. Just as binary multiplication can be carried out as a series of add and shift operations, division can be carried out by a series of subtract and shift operations. In one division scheme, the divisor is shifted right, the dividend is motionless before each subtraction. In another one, the dividend is shifted left,

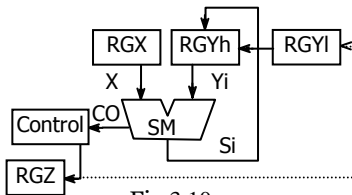


Fig.3.10

and the divisor is fixed. Consider the binary divider example for positive numbers, based on the second scheme. It contains  $2n+1$ -bit dividend register RGY,  $n$ -bit divisor register RGX, quotient register RGZ, subtractor SM, as shown in Fig.3.10. The shift register RGY is divided to the high RGYh and low RGYl parts.

On each step  $i$  the divisor  $X$  is subtracted from the remainder  $Y_i$  in RGYh,  $Y_i - X = S_i$ . If the result overflows, then the carry out bit is  $CO = 1$ . That means that the recent bit of the quotient is a 0. Otherwise, the result bit is a 1 and  $S_i > 0$  is stored in RGYh. After that, RGY is shifted left, and the bit of the quotient is shifted in RGZ. The first step proves that RGYh content is less than the divisor. Otherwise, the quotient overflow occurs. After  $n$  steps in RGYh the remainder remains, and in RGZ the quotient is. To get the remainder at its correct position, the register RGY is not shifted at the last step. Consider  $X = 1101 = 13$ ,  $Y = 10000111 = 135$ . The state chart of the division process is shown on the Table 3.5. We get the correct result  $Z = 1010 = 10$  and remainder  $101 = 5$ , i.e.  $135/13 = 10$  with a remainder 5.

This network hardware can be minimized, when the quotient bits are shifted not in RGZ but in RGYl substituting the low bits of the dividend (see the dotted line in Fig.3.10). The speed can be increased, when shift and subtraction are performed in a single cycle. The shift is done by the additional multiplexor, which selects shifted output of SM or RGYh to load in RGYh depending on signal CO.

Table 3.5

| Cycle | RGYh  | RGYl | CO | RGZ  | Operation  |
|-------|-------|------|----|------|------------|
| 0     | 01000 | 0111 | 0  | 0000 | initialize |
| 1     | 01000 | 0111 | 1  | 0000 | $-X$       |
| 2     | 10000 | 1110 | 0  | 0000 | shift      |
| 3     | 00011 | 1110 | 0  | 0000 | $-X$       |
| 4     | 00111 | 1100 | 0  | 0001 | shift      |
| 5     | 00111 | 1100 | 1  | 0001 | $-X$       |
| 6     | 01111 | 1000 | 0  | 0010 | shift      |
| 7     | 00010 | 1000 | 0  | 0010 | $-X$       |
| 8     | 00101 | 0000 | 0  | 0101 | shift      |
| 9     | 00101 | 0000 | 1  | 0101 | $-X$       |
| 10    | 00101 | 0000 | 0  | 1010 | shift, end |

This multiplexor can be removed, when another division schema is used. In it, the subtraction result (even the negative) is stored in RGYh in each step, and in the next step, to the negative remainder the dividend is added but not subtracted.

The binary divider achieves the maximum speed when it is built as an array LN. Such a divider con-

sists of  $n$  stages; each of them subtracts the divisor from the remainder calculated in the previous stage, and derives a single result bit.

The division of signed data consists in complementing the divisor and/or dividend, if negative, division the two positive binary numbers, and complementing the quotient if it should be negative. However, some algorithms combine the data complementing with the division flow.

The division is simply described in VHDL by the operator slash "/". But the synthesizer infers this operator only as a shifter, when the divisor is equal to  $2^i$ . Therefore, the binary divider should be described carefully as its behavior. The simplest unsigned divider of 8-bit dividend  $Y$  to 4-bit divisor  $X$  deriving 4-bit quotient  $Z$  is described by the following process statement

```
process(X,Y)
  variable yi: unsigned(8 downto 0);
  variable si: unsigned(4 downto 0);
begin
  yi:=Y&'0';
  for i in 0 to 3 loop
    si:=yi(8-i downto 4-i)-X;  yi:=yi;
    if si(4)='1' then
      Z(3-i)<='0';
    else
      Z(3-i)<='1'; yi(8-i downto 4-i):=si;
    end if;
  end loop;
end process;
```

where the **for – loop** statement implements 4 division iterations, each of them derives a single result bit. Note that all the signals are of unsigned type, which is declared in the IEEE.Numeric\_Bit package. When compiling, the **for – loop** statement is unrolled, and each iteration is inferred as a stage of the array divider.

### 3.5 *Hardware pipelining*

**Pipelining** is a special processing method, which makes it possible to execute simultaneously several computations on the same hardware by using its different parts. In this case, the computation is divided into execution steps and different steps are executed by different **pipeline stages**. A new computation is started before the old one finishes its execution. Several computations co-exist in the same pipeline, each of them being in a different execution phase. **Hardware pipelining** and **software pipelining** are distinguished. The first one is usually implemented in the datapath and in the control network of the computer. The second one is arranged at the software level, and it provides the effective utilization of the hardware pipelining, or plans the computations in separate units in the pipeline manner.

Consider the datapath, which implements the operation  $S = AX + Y$ . The datapath contains the input and output registers RGA, RGX, RGY, RGS,

multiplier MPU and adder SM (see Fig. 3.11(a)). The minimum clock cycle for this network is equal to the maximum delay between the outputs of the registers RGA, RGX, RGY, and input of the register RGS, and is estimated as

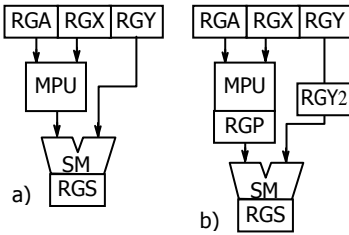


Fig.3.11

$$T_{CLK_1} = t_M + t_S \approx 3t_S.$$

We decompose the operation to two sequential steps:  $P = AX$  and  $S = P + Y$ . The result of the first step is stored in the intermediate register RGP, named as the pipeline register.

The additional pipeline register RGY2 is shown in Fig. 3.11(b). Its minimum clock cycle is estimated as  $T_{CLK_2} = \max(t_M, t_S) = t_M \approx 2t_S$ . This means that the hardware pipelining provides the 1,5 times increase of the clock frequency of this datapath.

When the calculations are planned properly, i.e. the input data loaded in the input registers in each clock cycle, then the pipelining provides the respective throughput increase of this datapath, which asymptotically achieves in our example the figure 1,5. To find out such a calculation plan, usually the **reservation table** is used. It is, in principle, a timing diagram, which shows the flow of data through the data path units. Each row of this table represents a data path unit and each column represents a time step. When the mark is set in a cell of the table, it means that the pipeline stage represented by this row is used during the execution step indicated by the column. A set of marks filling the table represents the execution pattern for a given computation.

Table 3.6

| Step | 1 | 2 | 3 | 4 | 5 |
|------|---|---|---|---|---|
| RGA  | X |   | * |   |   |
| RGX  | X |   | * |   |   |
| RGY  | X | X | * | * |   |
| MPU  | X |   | * |   |   |
| RGP  |   | X |   | * |   |
| SM   |   | X |   | * |   |
| RGS  |   |   | X |   | * |

Table 3.7

| Step | 1 | 2 | 3 | 4 | 5 |
|------|---|---|---|---|---|
| RGA  | X | * |   |   |   |
| RGX  | X | * |   |   |   |
| RGY  | X | * |   |   |   |
| MPU  | X | * |   |   |   |
| RGP  |   | X | * |   |   |
| RGY2 |   | X | * |   |   |
| SM   |   | X | * |   |   |
| RGP  |   |   | X | * |   |

circuits, network hardware reusability, power consumption minimizing. The hardware pipelining is one of the principles of the RISC processor architecture.

## 4. Control networks

### 4.1 CPU control unit

In the previous chapter it was mentioned, that the CPU consists of the control unit (CU) and the datapath, and the problems of the datapath design were enlighten. This chapter is devoted to the problems of the CU design. The von **Neumann model** of a computer is based upon a repeating four-cycle procedure to execute a program.

During the first cycle, named as instruction fetch, the CPU sends a signal to MU, telling it which instruction is needed. MU responds by sending the instruction to the CPU, where it is held in CU.

During the instruction decode cycle, the process is held of interpreting the instruction and determining what needs to be done within the CPU to implement this operation. The derived information is sent to the datapath from the CU.

The third cycle is the instruction execute. After the datapath receives the information from the CU, the instruction may be executed. The datapath receives the necessary input data, either from MU or a local storage within the datapath itself, and outputs the results.

In the storage cycle, the results are stored back in MU.

Every instruction in the program is treated using the same sequence of procedures. CPU repeats the same four cycles so long as the program is running. The main difference among the instructions is handled by changing the function, performed by the datapath network. Some instructions, which perform the program branches, may not control the datapath at all.

All the mentioned above functions except ones implemented by the datapath hardware are to be performed by the CU. These functions can be divided into categories depending on the purpose and target of them. The **fetching** functions control the instruction flow. They are responsible for fetching the next instruction depending on a set of conditions. They are condition branches, subroutine calls, etc. To recognize which of the functions have to be computed, CU implements an instruction decoding function. The **datapath control** functions provide the proper control flow for the arithmetic and logic operations in the datapath. The **addressing** functions provide the proper data addressing depending on the given addressing mode. Some functions handle with the **memory management** including virtual memory support, page mapping and memory caching. **Tasking** functions provide the simultaneous computing by CPU of a set of tasks, for example, in the multiprogramming mode. Here the **context switching** functions take place.

An **exception** is an interruption of the normal flow of instruction processing. There are two situations, in which this occurs. The first, which is called as a **trap**, occurs when CPU recognizes that the execution of an instruction has caused an error of some kind (overflow, fall in the protected address area, etc.). The second, which we call an **interrupt**, occurs when a device external to CPU signals that a certain event should be processed. Usually CPU has hardware support for interrupts and traps, which we concern as the CU function as well.

## 4.2 CPU instruction set

The **instruction** word can be decomposed to operational part, named opcode, and address part. In the **opcode** the code of the function is placed, which implements the CPU when this instruction runs. Obviously, if more bits are used in the opcode, then more distinct instructions can be supported. The cleanest approach is to use a fixed number of opcode bits for all instructions. In practice, a designer will recognize that certain operations are much more common than others, and its opcodes can be shorter. For example, if we have determined 4-bit opcode for the most commonly used instructions then 16 possible bit patterns are available. 15 of these are used for the most common instructions, and 16-th is used to indicate all of the other instructions. Additional bits then attached to the instruction format so that these less common instructions can be distinguished. Due to this principle, the Intel complex instruction set computer (**CISC**) opcodes were selected, which number of opcode bits ranges from 5 to 19.

On the other hand, the reduced instruction set computer (**RISC**) designs strongly favour short opcodes and small number of uniform instruction formats, preferably all of the same size. The regularity of these formats simplifies the instruction decoding mechanism and pipelined instruction implementation.

The information about operand and result addresses, and place of the next instruction stays in the **address part**. Four, three, two, one, and zero address instructions are distinguished depending on the number of addresses, which are given in the instruction. One kind of zero address instructions distinguishes the implicit operand address, for example, a separate register like accumulator. The instructions of another kind carry in itself the, so called, immediate operand.

The address number depends on the addressing spaces and it is usually optimized when the instruction set is selected. One hand, the many address instruction can substitute a set of short instructions, providing the performance increase. The other hand, the modern MU address spaces afford the address length up to four and even eight bytes, and many address instructions take a lot of memory space and decrease the instruction access time. Therefore, the instruction format selection is a complex optimization task. It is usually finished by comparing the time of the testbench program implementation on the CPU models with different instruction sets.

One of the ideas of the RISC computers consists in division of the instructions to ones, which handle only with the datapath, and others, which do not. The first type instructions are three and four address instructions. Here the addresses of the register array are used, which are 4-5-bit width codes (sometimes up to 7). And the second type instructions are zero, one, and two address instructions. In the two-address instruction, first address points to the MU cell, and the second one is the register address. Such instruction provides the data loading to or storing from the register.

The address field can represent either direct address or code, which plays the distinguished role in the address calculation, i.e. for the indirect addressing. CPU can provide a lot of addressing modes. In choosing a set of addressing modes, the issues of instruction complexity versus utility arise as well.



### 4.3 Control unit structure

At the beginning of the chapter, a set of functions of the CU was declared. These functions can be implemented in the separated and relatively independent units. The CU of the common CPU contains the instruction fetch network IFN, address and memory management unit AMMU, interrupt handling unit IHU and finite state machine (FSM) or microprogram controller (MPC) for the control algorithm implementation. Its approximate structure is illustrated by Fig.4.1. The instruction fetch network consists of program counter (PC), instruction register (IRG) and instruction decoder DC. IR contains opcode field OP and address fields A1, A2.

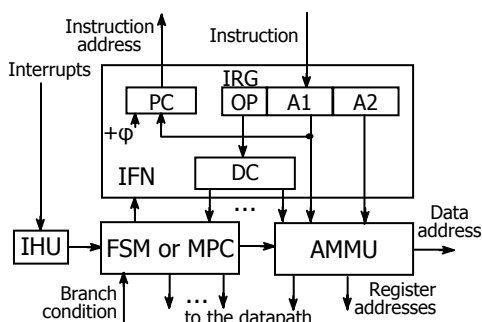


Fig.4.1

IRG holds the binary instruction, which opcode is decoded by the instruction decoder. PC is used to maintain the flow of the program by counting the instructions as they are executed. It contains the memory address for the next instruction that is retrieved and executed by CPU. The instruction address from PC is fed to the program memory, and the corresponding data word is transferred to IRG. If the instruction order is plain, then the

next address is  $PC = PC + \phi$ , where  $\phi$  is the spacing needed to get the next instruction. For example, when the instruction length is 4 bytes then  $\phi=4$ .

Once in the IRG, the instruction is decoded by DC, which causes the instruction to be carried out. When the procedure is finished, the instruction is retired, and the next instruction is fetched from the memory and transferred to the IR. When the branch instruction is decoded then depending on the branch condition, the code from the address field is loaded to the PC. Then the next instruction is selected from the cell with the branch address. When the subroutine call instruction is decoded, then before the call address is entered PC, its content, increased to  $\phi$ , is stored into the address stack as the return address.

The address and memory management unit AMMU performs the data address generation using a set of addressing modes like direct, indexed, based, indirect addressing and its combinations. It provides for efficient memory access by separating the notion of logical and physical memory accesses. It is responsible for the context storing when the context switching occurs.

FSM or MPC generate the main control signal stream both for datapath and for other units of the CU providing the correct implementation of the decoded instruction. FSM is a sequential logic network, which implements some control algorithms. Its state depends on the previous one and on the input signals. Its outputs depend logically on its state and, may be, on input signals. MPC is a kind

of FSM, which state is the state of the microinstruction address counter. Its output signals are the bits of the microinstruction, which is selected by the microinstruction address counter from the microprogram MU. MPC is simple for designing and expanding of its functions, but its speed is comparatively small. FSM is hard to design, but it is the fastest sequential machine with small energy consumption. Because the FSM synthesis is fully implemented by CAD systems, most of CPU is based now on FSM.

The interrupt handling unit fixes the interrupt signals and trap conditions. After that, it provides selection of the signal with the highest priority and call of the respective interrupt subroutine. By this process, the CU provides correct program flow abrupt, storing the CPU state and restoring this state after the return from the interrupt subroutine.

CUs are divided into central and distributed CUs. Above the structure of central CU was described. Such CUs are used in the simple CPUs and microcontrollers. In the distributed CU, the master and slave CUs are distinguished. The master CU decodes the opcode and gives only the general control signals. These signals control the slave CUs, which implement the direct control of the instruction implementation in parts of the datapath, address and memory management unit and others units. For example, the separate slave CU can control the datapath for the division and square root implementation. In the next chapters, the CU parts will be discussed in details.

#### 4.4 Instruction fetch networks

All the CPU instructions are divided into two categories depending on the way, how the next instruction address is formed. The instructions of the first type do not violate the natural order of the instruction flow. After finishing the instruction in the  $j$ -th cell, the CPU starts the implementation of the instruction in the  $j+1$ -th cell. They usually are data moving, arithmetic and logic instructions. The second type instructions make the conditional or unconditional jumps. For example, the instruction JZ A, which is placed in the  $j$ -th cell, checks the condition flag Z. If  $Z = 1$  then a jump to the instruction in the cell A is performed. If  $Z = 0$  then a new instruction is one from the cell  $j + 1$ . The unconditional jump instruction JMP A behaves as the instruction JZ for which Z is always a 1.

The diagram of the subnetwork, which implements these jumps, is shown in Fig.4.2. If the instructions JMP or JZ are computed, and the jump condition satisfied

(the trigger Z is in the state 1) then the next instruction address is formed by writing the address from the instruction in the PC. In other cases the next instruction address is formed by addition a 1 to the PC. The signal ST strobes the PC and it is formed by FSM in the cycle of the PC modification. In the network in Fig.4.2 a single flag Z is analyzed for jumping. In the real CPUs a set of flags are used for this purpose. For example, in the microprocessor I8080 the jump is performed using

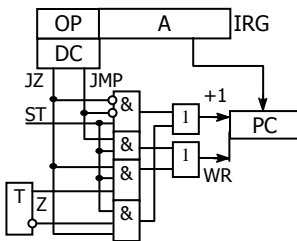


Fig.4.2

four conditions  $Z$ ,  $C$ ,  $N$ , and  $V$ , and their negations (see Table 4.1). The three byte jump instructions contain an opcode byte 11xxx010, and two bytes of the jump address, where xxx is the condition code. One of possible subnetworks for implementation of this jump mechanism is shown on Fig.4.3.

In the previous example, a 1 is added to the PC because the MU cell is considered to have a length of a single instruction. Most of CPUs have instruction sets with the variable length. This length is usually proportional to 8 bits. In this situation, the instruction fetch goes by one, two and more bytes; each of them has its own address. The instruction address is equal to its first byte address. Note that in many CPUs the big-endian byte addressing of the word is used, where the most significant byte is stored in the lowest addressed byte. Then the instruction address is equal to its last byte address. Anyway, the next instruction address is higher to  $\phi$ , where  $\phi$  is the executed instruction length in bytes. As a result, to form the next instruction address the instruction decoder has to generate the code of the instruction length  $\phi$ , which is used as an increment to the PC.

A simple network to do this is shown in Fig.4.4. The least significant bits of the PC are implemented as a register RG with multiplexor and adder SM at its input. The address increment code  $\phi$  from some instruction decoder output is added to the register content. The adder overflow is used as an increment signal to the counter CTR, which form the most significant bits of the PC. When the jump is implemented, then the jump address is loaded to counter and register.

In modern CPUs, the instructions of the length from 1 to 12 bytes are used. The data and instruction bus width is equal to 4, 8 and even 16 bytes. Therefore, the instruction information has to be read as words of respective length of 4, 8, 16 bytes in a buffer, and then the executed instruction has to be selected from this buffer. Such a buffer is named as a prefetching buffer. In this buffer, a set of instructions can be stored except one under execution. Because this buffer has the high speed, it plays the role of a cache memory between CPU and slower MU. It is very important in pipelined and superscalar CPUs. But a bad situation occurs, when the instruction length is higher than one of the buffer, or the address of the long instruction is not aligned to the word bound. Then this instruction is read from the buffer in two steps as higher and lower parts.

Table 4.1

| Instruction mnemonics | Jump condition | xxx |
|-----------------------|----------------|-----|
| JNZ                   | $Z = 0$        | 000 |
| JZ                    | $Z = 1$        | 001 |
| JNC                   | $C = 0$        | 010 |
| JC                    | $C = 1$        | 011 |
| JPO                   | $V = 0$        | 100 |
| JPE                   | $V = 1$        | 101 |
| JP                    | $N = 0$        | 110 |
| JM                    | $N = 1$        | 111 |

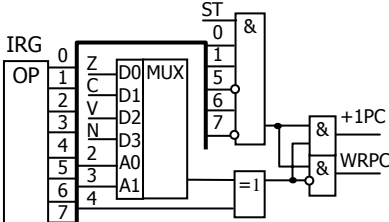


Fig.4.3

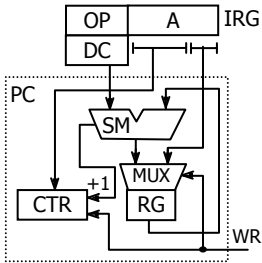


Fig.4.4

### 4.5 Finite state machines

FSM is a sequential logic network, which implements the given control algorithm as the predefined sequence of its states. Its next state  $S_{i+1}$  depends on the previous one  $S_i$  and on the input signals  $X_j$ . Its outputs  $Y_p$  depend logically on its state  $S_i$ . Such a FSM is named as a Moore FSM. When its outputs  $Y_p$  depend both on its state  $S_i$  and on input signals  $X_j$ , then it is named as a Mealy FSM. The FSM algorithm is fully described by its **state graph** (state diagram) or by **FSM chart**. Modern CADs synthesize the FSM from its state graph automatically.

The nodes of the state graph represent the FSM states, and its directed edges represent the branches from one state to another. In the **Moore** state graph the node  $S_k$  is labeled by variables  $Y_p$ , separated by a slash "/", which output a 1, when FSM stays in this state. The edges are labeled by the input labels which are the Boolean functions of the input variables, and which derive the branch conditions. They can be labeled by the output labels that are output variables, when it is the **Mealy** state graph. In Fig.4.5 is an example of the state graph, which has both Moore and Mealy outputs.

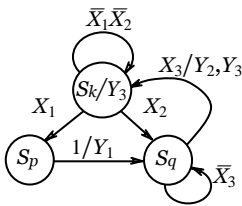


Fig.4.5

If we label an edge  $X_i X_j / Y_p Y_q$ , this means that if inputs  $X_i$  and  $X_j$  are 1 (we don't care what the other input values are), the outputs  $Y_p$  and  $Y_q$  are 1 and other outputs are 0, and we will traverse this edge to go to the next state. For example, in the graph in Fig.4.5 the state  $S_k$  remains the same when  $\bar{X}_1 \bar{X}_2 = 1$  and it is exchanged to the state  $S_k$  when  $X_1 = 1$ , providing the output signal  $Y_1 = 1$ . In order to have a completely specified proper state graph, in which the next state is always uniquely

defined for every input combination, we must place the following constraints on the input labels for every state  $S_k$ :

If  $F_i$  and  $F_j$  are any pair of input labels (Boolean functions) on edges exiting state  $S_k$ , then  $F_i \cdot F_j = 0$ , if  $i \neq j$ .

If  $n$  edges exit state  $S_k$ , and they have input labels  $F_1, F_2, \dots, F_n$ , respectively, then  $F_1 \vee F_2 \vee \dots \vee F_n = 1$ .

The first condition assures us that at most one input label can be a 1 at any given time, and the second condition assures us that at least one input label will be a 1 at any given time. Therefore, exactly one label will be 1, and the next state will be uniquely defined for every input combination. For example in Fig.4.5 conditions are satisfied for all the nodes.

As an alternative to state graphs, a state machine flowchart, or FSM chart is. Just as flowcharts are useful in software design, they are useful in the hardware design of digital systems. The state in it is represented by a state box. It contains a state name, followed by a slash "/" and an optional output list. A state code may be placed outside the box. A decision box is represented by a diamond-shaped symbol with true and false branches. The condition placed in the box is a Boolean expression that is evaluated to determine which branch to take. The conditional output box, which has curved ends, contains a conditional output list.

An FSM chart is constructed from FSM blocks. Each of them contains exactly one state box, together with the decision boxes and conditional output boxes, associated with that state. Block has one entrance path and one or more exit paths. Each block describes the operation during the time that FSM is in one state. A path through a block from entrance to exit is referred to as a link path.

Certain rules must be followed when constructing an FSM block. First, for every valid combination of input variables, there must be exactly one exit path defined. This is necessary since each allowable input combination must lead to a single next state. Second, no internal feedback within a block is allowed.

It is easy to convert a state graph to an equivalent FSM chart. The chart, which is equivalent to one in Fig.4.5, has three blocks – one for each state. The Moore output  $Y_3$  is placed in the state box  $S_k$ , since it does not depend on the input. Some condition nodes  $X_1, X_2$  have a single output. This is explained by the fact that the Mealy outputs  $Y_1, Y_2$  appear in conditional output boxes, since they depend on both the state and input. The resulting FSM chart is shown in Fig.4.6.

The FSM network is searched as the network shown in Fig. 1.6. It contains a set of triggers and LN. Triggers store the FSM state, or other words, they form a state register. The LN consists of two parts. One of them generates signals  $D_i$ , which are the trigger stimulating functions. Another one outputs the resulting signals  $Y_j$ .

First of all, the states  $S_k$  are given the concrete values. There is a set of methods of coding the states. The method selection depends on the number of states, whether the FSM is optimized for speed or hardware volume, or error immunity. The one-hot coding means that for  $n$  state FSM the  $n$ -bit wide state word is selected, in which a 1 stays in the  $k$ -th position, when coding the state  $S_k$ . For example, FSM with the state graph in Fig.4.5 would have the state coding 001, 010, 100. This coding is usually used when the state number is small. It provides usually the highest speed, because the trigger stimulating functions occur to be rather small. When the state graph contains the long chains of nodes, the state register can be implemented as a shift register.

The natural number coding is used in most of cases, especially, when the state graph contains the long chains of nodes. In the example on Fig.4.6, the codes are 00, 01, and 10. In this situation, the state register behaves as a counter. When the states are coded by the Gray codes, then in most of state branches, only a single bit of the code is exchanged. This serves both to LN minimization and to error immunity. In the combined coding, the code word is divided into 2 or more fields, each of them are coded by some coding method. Here the advantages of different methods can be used. For example, when the code word has two  $n$ -bit fields, which have one-hot coding, then such a code word can code up to  $n^2$  states.

Then, the state table is built. This table has the columns of present state, next state and present output. The next state column contains the subcolumns, which

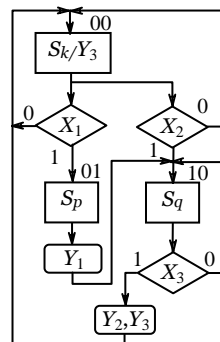


Fig.4.6

are coded by the bits of the input signals. These subcolumns show what next state is for the given value of the input variable. The present output column has the similar form. Table 4.2 is the state table for the FSM chart in Fig.4.6.

From the next state columns of the state table the trigger stimulating functions are derived, as signals that force the D triggers of the state register to be set:

Table 4.2

| Present state<br>$Z_2Z_1$ | Next state, $X_3X_2X_1 =$ |     |     |     |     |     |     |     | Present output, $X_3X_2X_1 =$ |     |     |     |            |     |     |     |
|---------------------------|---------------------------|-----|-----|-----|-----|-----|-----|-----|-------------------------------|-----|-----|-----|------------|-----|-----|-----|
|                           | 000                       | 001 | 010 | 011 | 100 | 101 | 110 | 111 | 000                           | 001 | 010 | 011 | 100        | 101 | 110 | 111 |
| $S_k$ 00                  | 00                        | 01  | 10  | –   | 00  | 01  | 10  | –   | $Y_3$                         |     |     |     |            |     |     |     |
| $S_p$ 01                  | 10                        |     |     |     |     |     |     |     | $Y_1$                         |     |     |     |            |     |     |     |
| $S_q$ 10                  | 10                        |     | 00  |     | 10  |     | 00  |     | 0                             |     |     |     | $Y_2, Y_3$ |     |     |     |

$$D_1 = \bar{Z}_2\bar{Z}_1(\bar{X}_3\bar{X}_2X_1 \vee \bar{X}_3X_2X_1 \vee X_3\bar{X}_2X_1 \vee X_3X_2X_1) = \bar{Z}_2\bar{Z}_1X_1;$$

$$D_2 = \bar{Z}_2\bar{Z}_1(\bar{X}_3X_2\bar{X}_1 \vee \bar{X}_3X_2X_1 \vee X_3X_2\bar{X}_1 \vee X_3X_2X_1) \vee \bar{Z}_2Z_1 \vee Z_2Z_1 \vee Z_2\bar{Z}_1\bar{X}_2 = \\ = \bar{Z}_2\bar{Z}_1X_2 \vee Z_1 \vee Z_2\bar{Z}_1\bar{X}_2.$$

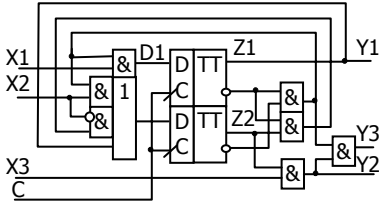


Fig.4.7

Note that the don't-care states in the combination  $X_2X_1 = 11$  and in the state  $Z_2Z_1 = 11$  (for  $D_2$ ) are assigned as a 1. The output functions are derived from the present output columns as well:

$$Y_1 = Z_1; \quad Y_2 = Z_2X_3; \quad Y_3 = \bar{Z}_2\bar{Z}_1 \vee Z_2X_3;$$

The resulting FSM network is shown in Fig.4.7.

Once the state graph or the FSM chart is built, then the FSM can be easily described in VHDL and then its network can be synthesized by a compiler. Here a case statement can be used to specify what happens in each state. Each condition box corresponds directly to an if statement. The following program describes FSM with the chart in Fig.4.6.

```
entity FSM1 is port(C,X1,X2,X3:in bit;
                    Y1,Y2,Y3:out bit);
end FSM1;
architecture beh of FSM1 is
    signal S,D:bit_vector(0 to 1);    -- state codes
begin
    LN:process(X1,X2,X3,S) begin      --LN model
        Y1<='0';Y2<='0';Y3<='0';    --usual output states
        case S is
            when "00" => Y3<='1';    -- current state Sk
                if X1='1' then
                    D<="01";         --next state
                elsif X2='1' then
                    D<="10";
                end if;
            end case;
        end process;
```

```

        else      D<="00"
        end if;
    when "01"=> D<="10"; Y1<='1'; -- state Sp
    when others=> if X3='1' then -- state Sq
        D<="00";Y2<='1';Y3<='1';
        else      D<="10";
        end if;
    end case;
end process;
RG:process(C) begin -- state register
    if C='1' and C'event then
        S <= D; -- update state on rising edge of C
    end if;
end process;
end beh;

```

The first process represents the LN of FSM, and the second process updates the state register on the clock. The signals  $Y1$ ,  $Y2$ ,  $Y3$  are turned on in the appropriate states, and they must be turned off when the state  $S$  changes. A convenient way to do this is to set them all to a 0 at the start of the process.

The methods used to derive either state graph or FSM chart for a CU are similar. First, we should draw a block diagram of the system we are controlling. Next we should define the required input and output signals to the CU. Then we can construct an FSM chart or a state graph that tests the input signals and generates the proper sequence of output signals.

Consider an example of the CU design for the multiplier unit, shown in Fig.3.7. Its operation is described by the Table 3.3. Let the unit starts his operation by the signal *Start*. The output control signals are register reset  $R$ , shift signal  $Sh$ , register loading  $L$ , result ready  $Rdy$ . Because the output signals depend only on the FSM state, the FSM is of Moore type, except the reset  $R$ . The multiplier registers shift data not in the cycle, when the shift signal enters, but in the next one. And the shift signals have to be generated in the previous state comparing to the Table 3.3. The resulting state graph is shown in Fig.4.8. Let the state coding is coding by natural numbers. The Table 4.3 is the resulting state table. The trigger stimulating functions are the following:

$$D_1 = \bar{Z}_1; D_2 = \bar{Z}_2 Z_1 \vee Z_2 \bar{Z}_1 = Z_2 \oplus Z_1; D_3 = Z_3 \oplus (Z_2 Z_1); D_4 = \overline{Start} \cdot \bar{Rdy} \cdot (Z_3 Z_2 Z_1);$$

The output signals are:

$$L = \bar{Z}_4 \bar{Z}_1; Sh = Z_1; Rdy = Z_4; R = Start \cdot Z_4.$$

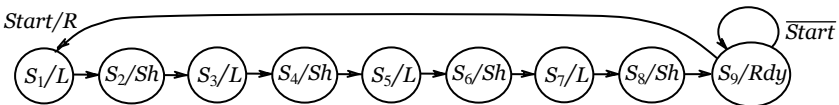


Fig.4.8

Table 4.3

| Present state<br>$Z_4Z_3Z_2Z_1$ | Next state,<br>$Start=$ |      | Pre-<br>sent<br>output |
|---------------------------------|-------------------------|------|------------------------|
|                                 | 0                       | 1    |                        |
| $S_1$ 0000                      | 0001                    | 0001 | $L$                    |
| $S_2$ 0001                      | 0010                    | 0010 | $Sh$                   |
| $S_3$ 0010                      | 0011                    | 0011 | $L$                    |
| $S_4$ 0011                      | 0100                    | 0100 | $Sh$                   |
| $S_5$ 0100                      | 0101                    | 0101 | $L$                    |
| $S_6$ 0101                      | 0110                    | 0110 | $Sh$                   |
| $S_7$ 0110                      | 0111                    | 0111 | $L$                    |
| $S_8$ 0111                      | 1000                    | 1000 | $Sh$                   |
| $S_9$ 1000                      | 1000                    | 0000 | $Rdy$                  |

We can see that this FSM behaves as a counter, and really, it is that. Therefore, such FSM can be based on the counter network, and on the contrary, counters can be synthesized as FSMs.

When a FSM becomes large and complex, it is desirable to divide the machine up into several smaller FSMs that are linked together. Each of that FSM is easier to design and implement. Also, one of the submachines may be "called" in several different places by the main FSM. This is analogous to dividing a large software program into procedures that are called by the main program. Fig.4.9 shows the FSM charts for two serially linked FSMs, which have the common clock source. The main FSM *A* executes a sequence of "some states" until it is ready to call the submachine FSM *B*. When state  $S_A$  is reached, the output signal  $Y_A$  activates FSM *B*. FSM *B* then leaves its idle state and executes a sequence of "other states". When it is finished, it outputs signal  $Z_B$  before returning to the idle state. When FSM *A* receives  $Z_B$ , it continues to execute "other states".

As an example of using linked FSMs the CPU can be considered, which contains the multiply unit in Fig.3.7 as a component with the local FSM, which the state graph is shown in Fig.4.8. When the multiplication operation is decoded, the main FSM sends to the local FSM the *Start* signal, which runs the multiplication. After multiplication finishing, the local FSM falls in the idle state  $S_0$ , and returns the signal *Rdy*, which continues the operation of the main FSM.

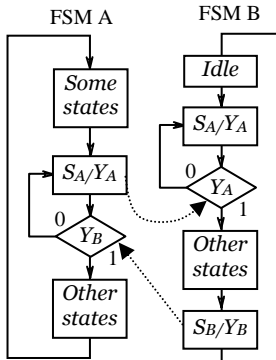


Fig.4.9

## 4.6 Microprogram controllers

CISC computers obtained their large instruction set by using a technique called **microprogramming**. The microprogram approach breaks down every basic operation into a microinstruction. The microprogram is written in the similar way as the computer program is. But the microcode instructions are one level deeper and more primitive than assembly language. In fact, an assembly-level instruction is created by using a sequence of micro-operations. Microprogramming allows the designer to create an instruction by combining the operations at the microcode level.

The block diagram of the usual microprogram controller (MPC) is shown in Fig.4.10. Each microcode sequence is stored in a microcode ROM. When CPU is given an instruction such as load word, the respective microcode sequence is accessed. This sequence includes the operations: start, get address data from



given register, send address to MU, get data from MU, send data to given register, end.

By this process, the instruction opcode from IRG is sent to the sequencer SEQ. The **sequencer** is a unit of the MPC, which determines the location of the microcode corresponding to the specified instruction, reads the first microinstruction, then the second, and so on, until the instruction sequence is completed. The type of the microinstruction, like normal flow, conditional branch, subroutine call, etc., as well as the branch address information, is derived from special fields of the microinstruction. The multiplexor MUX selects the input signals *COND*, which serve as the conditional branch signals.

The control fields of the ROM are divided to ones with the horizontal coding *Ch*, and ones with the vertical coding *Cv*. By the **horizontal coding**, each control signal has its own bit of the microinstruction word. The disadvantage of this coding is that for complex CPUs the microinstruction is too long (hundreds of bits). By the **vertical coding**, a set of control signals, which could not occur as a 1 simultaneously, are coded by a single bit field. They are generated by the decoder of this field. As a result, the microinstruction length is much shorter. But the critical path in the network is increased in the decoder delay, and some independent control signals have to be generated in sequence. All this decreases the CPU speed.

The microprogramming is a powerful approach for increasing the instruction set of CPU. Adding a new instruction simply requires that additional code be written and stored in the ROM, so that modifications to the instruction set do not require extensive hardware changes. Comparing to FSM, the main MPC drawbacks are the following. The MPC could not analyze large set of input signals simultaneously (usually only a single condition), the number of independent output signals (by the horizontal microprogramming) is limited by the microinstruction width. Therefore, the microprogram can grow dramatically by the cost of microinstructions, which prove a set of conditions and output many signals (by the vertical microprogramming) in sequence, which is followed by the unnecessary decrease of the CPU speed. The control of two executed instructions is impossible, because only a single microprogram could run in a time. Therefore, the parallel execution of the program in CPU, for example, by the pipelining, is impossible too. The critical path in the MPC includes the condition selector, sequencer, address decoder, ROM array, microinstruction field decoder, etc. As a result, the MPC is usually too slow comparing to the FSM. The easy way of the microprogram writing is not actual now, because all the modern digital hardware CADs support the FSM design, and do not provide the microprogramming.

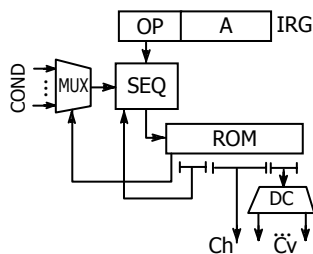


Fig.4.10

### 4.7 Example of CPU design

Consider the design of CUs for the simplest but convenient CPU. It contains RAM, three-port FM and a simple ALU, named LSM (see Fig.4.11). The access to the asynchronous RAM needs the 13 bits of the address  $A$  (when the RAM volume is 8K words) and the control signal  $WR$ . The access to the FM needs three addresses  $AB$ ,  $AD$ ,  $AQ$  and write signal  $WR$ . When the FM volume is 8 words, then  $AB$ ,  $AD$ ,  $AQ$  are 3-bit wide busses. The 3-bit wide control code  $F$  controls LSM, which performs up to 8 operations. The shifter SHU performs the right shift to a single bit. CU contains instruction fetch network (IFN) based on the program counter (ICTR) and instruction register (IRG), and on FSM with the flag register (CCRG).

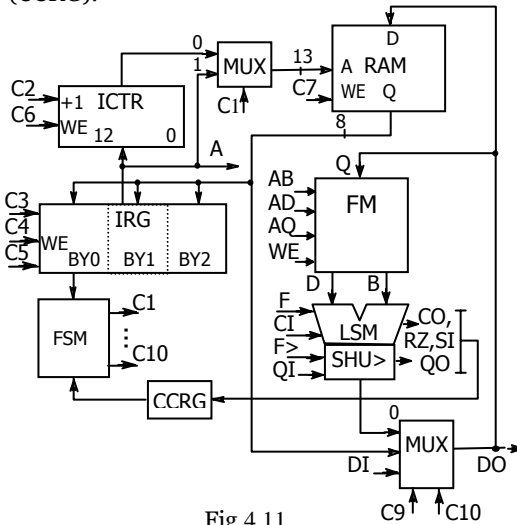


Fig.4.11

Table 4.3

| $F_1F_0$ | $M = 0$             | $M = 1$                                |
|----------|---------------------|--|
| 0 0      | 0                   | $\bar{B} \cdot D \vee B \cdot \bar{D}$ |
| 0 1      | $\bar{B} + D + C_i$ | $B \vee D$                             |
| 1 0      | $B + \bar{D} + C_i$ | $B \cdot D$                            |
| 1 1      | $B + D + C_i$       | 1                                      |

assigned, which gives  $2 \times 2^3 \times 2^{13} = 2^{17}$  different codes.

Instructions JC break the natural instruction flow, when the given condition is satisfied. Consider four such conditions (RZ – zeroed result, SI – negative result, CO – carry output and QO – shifted out bit) with its inversions, then up to  $8 \times 2^{13} = 2^{16}$  different JC instructions may occur, where  $2^{13}$  is the jump address number. The

IN and OUT instructions give the moving direction (to CPU or from CPU) and the peripheral unit address (PU). The peripheral unit number is considered to be less than the RAM volume, and then  $2 \times 2^{13} = 2^{14}$  such different instructions may be.

The resulting amount of the instructions is estimated as  $2^{18}$ . Then for the instruction code at least 3 bytes are needed. But 3 bytes provide to code up to  $2^{24}$  different instructions, which are redundant. This redundancy can be used to decrease some instruction length, especially for that, which code number is much

Consider CPU which performs arithmetic (AO) and logic (LO) operations, data moving instructions from FM to RAM (FR) and in reverse order (RF), conditional (JC) and unconditional (JMP) jumps, input (IN) and output (OUT) instructions. 4-bit wide code controls the arithmetic and logic operations (3 bits for LSM and one for SHU, see Table 4.3). Three 3-bit wide address fields needed for the FM. The amount bit width is 13 bits, which gives  $2^4 \times 2^3 \times 2^3 \times 2^3 = 2^{13}$  different codes. In the instructions FR and RM the moving direction, FM address are

less than  $2^{16}$ . Such instructions are two-byte ones. The instructions that need the RAM address are the three-byte instructions. The derived instruction set is shown in Fig.4.12.

To simplify the LN, which provides the address transfer from IRG to ICTR, this address is placed in the second (BY1) and third (BY2) bytes. The first instruction byte (BY0) is used for opcode coding (OP). For example, the code 11111111 is used for coding the instructions

| OP     |    | BY0                          |           |  |  |  |  |  |  | BY1                |    |    |           |              |  |  |  | BY2      |  |  |  |
|--------|----|------------------------------|-----------|--|--|--|--|--|--|--------------------|----|----|-----------|--------------|--|--|--|----------|--|--|--|
|        |    | 7 6 5 4 3 2 1 0              |           |  |  |  |  |  |  | 7 6 5 4 3 2 1 0    |    |    |           |              |  |  |  | 76543210 |  |  |  |
| JMP    |    | 1 1 1 1 1 1 1 1              |           |  |  |  |  |  |  | 1 1 1              |    |    |           | A RAM        |  |  |  |          |  |  |  |
| JC     |    | 1 1 1 1 1 1 1 0              |           |  |  |  |  |  |  | A CC               |    |    |           | A RAM        |  |  |  |          |  |  |  |
| FR,RF  |    | 1 1 1 1 1 1 0 <sup>0/1</sup> |           |  |  |  |  |  |  | A FM               |    |    |           | A RAM        |  |  |  |          |  |  |  |
| IN,OUT |    | 1 1 1 1 1 1 1 1              |           |  |  |  |  |  |  | 1 0 <sup>0/1</sup> |    |    |           | A RAM (A IO) |  |  |  |          |  |  |  |
| AO     | 0  | AQ                           | 0 0 0 0 1 |  |  |  |  |  |  |                    | AB | AD | F1,<br>F0 |              |  |  |  |          |  |  |  |
|        | 1  |                              | 0 0 0 1 1 |  |  |  |  |  |  |                    |    |    |           |              |  |  |  |          |  |  |  |
|        | CO |                              | 0 0 1 0 1 |  |  |  |  |  |  |                    |    |    |           |              |  |  |  |          |  |  |  |
|        | SI |                              | 0 0 1 1 1 |  |  |  |  |  |  |                    |    |    |           |              |  |  |  |          |  |  |  |
|        | QO |                              | 0 1 0 0 1 |  |  |  |  |  |  |                    |    |    |           |              |  |  |  |          |  |  |  |
| LO     |    |                              | 1 0 0 0 1 |  |  |  |  |  |  |                    |    |    |           |              |  |  |  |          |  |  |  |
| SH>    | 0  | AQ                           | 1 0 0 0 0 |  |  |  |  |  |  |                    |    |    |           |              |  |  |  |          |  |  |  |
|        | 1  |                              | 1 0 0 1 0 |  |  |  |  |  |  |                    |    |    |           |              |  |  |  |          |  |  |  |
|        | CO |                              | 1 0 1 0 0 |  |  |  |  |  |  |                    |    |    |           |              |  |  |  |          |  |  |  |
|        | SI |                              | 1 0 1 1 0 |  |  |  |  |  |  |                    |    |    |           |              |  |  |  |          |  |  |  |
|        | QO |                              | 1 1 0 0 0 |  |  |  |  |  |  |                    |    |    |           |              |  |  |  |          |  |  |  |

Fig.4.12

JMP, IN, OUT, the code 11111110 is used for JC, etc. In the instruction JC except the RAM address the condition code (A CC) is placed. Such condition number is equal to 8 (4 direct and 4 inverse), and for their coding 3 bits are used, which are placed in positions 5, 6, 7 of BY1. In the instructions FR, RF these positions are utilized to code the FM address, which plays the role in the data transfer. And the transfer direction is coded by the zero bit of BY0, i.e. the code 11111101 represents the FR instruction, and 11111100 – does RF.

To distinguish codes of instructions IN, OUT, JMP the positions 7, 6, 5 of the byte BY1 have the coding 100, 101, and 111, respectively. Consider that input and output are implemented through a fixed register of FM (say, the zero). Instructions AO, LO output the codes in LSM, operand addresses AB, AD and result address AQ. All these codes have the length of 3 bits. The instruction AO is distinguished from LO, that the bit  $M$  blocks or unblocks the carry bits in LSM. This bit has the position of the 4-th bit in BY0.

When the instruction AO is implemented, the carry in bit  $C_i$  has to be controlled. For example, when the subtraction  $B - D$  is calculated, then the addition  $B + \bar{D} + C_i$  is carried out, and the carry in bit has to be  $C_i = 1$ . And by addition  $B + D$  this bit is a zero. By the shift left operations the addition  $B + B = 2B$  is fulfilled, and the bit  $C_i$  plays the role of the shifted in bit. In general, in the LSB of LSM can be put in 0, 1, CO, SI, and QO from CCRG. Therefore, each AO instruction has 5 types (0, 1, CO, SI, and QO) depending on the bit, which is used as a carry input for LSM. These types are coded by the codes 000, 001, 010, 011, 100, respectively, which stay in positions 3, 2, 1 of BY0, and code 01 in positions 4 and 0 of BY1 distinguish the AO code.

The instruction LO does not afford the code for the carry input, because by the

logic operations all the carry bits are blocked. Therefore, LO is distinguished by the code 11 in positions 4 and 0 of BY1.

The right shift operations (SH>) are the single address instructions, because the operand  $Q$  is loaded from FM, is shifted and is stored in the same register of FM. By this process, the MSB is released, and it can be written as 0, 1, CO, SI, or QO. Therefore, 5 kinds of this instruction are present. Because according to Table 4.3, LSM needs two operands, then LSM will translate a single operand, when one of the following logic operations is implemented:  $B \cdot D$  or  $B \vee D$  when  $B = D$ . If the shifted in bit is coded as well as it is in AO operation, then all the needed coding information can be placed in a single byte BY0 (see Fig.4.12). It is important to take in consideration by the FSM synthesis, that  $F_0 = \bar{F}_1$ , which selects  $B \cdot D$  or  $B \vee D$ , and  $AB = AD = AQ$ .

The instruction cycle is the period of time, when a single instruction is performed. The algorithm of the instruction cycle fulfillment depends on the opcode and on the instruction length. Consider the two byte instruction, which adds two words from FM. Then the instruction cycle consists of the next steps:

- 1) first instruction byte is selected from RAM by the address from ICTR;
- 2) this byte is stored in the first byte of IRG, and ICTR is incremented to a 1;
- 3) second byte is loaded in IRG as well as the first byte is in steps 1), 2);
- 4) operands  $B$  and  $D$  are selected from FM by the addresses  $AB$  and  $AD$ , and they are transferred to LSM;
- 5) result from LSM is stored to FM.

The next instruction makes the steps 1) – 5) and so on. All these activities are controlled and synchronized by the output signals of the FSM. Consider the FSM which is built on the PLA cells, which triggers are switched on the rising edge of the clock  $C$ . Then the waveform diagrams of the control signals in CPU looks like ones in Fig.4.13. All the instruction cycles start from the clock cycle  $T_1$  (the first instruction byte fetching). At this process, the first byte address from ICTR is transferred through the input 0 of the multiplexor (because  $C_1 = 0$ ) to the input A of RAM. This address selects a byte from the RAM, which is stored to 3-bit wide IRG. For the proper CPU operation, these activities should be finished till the

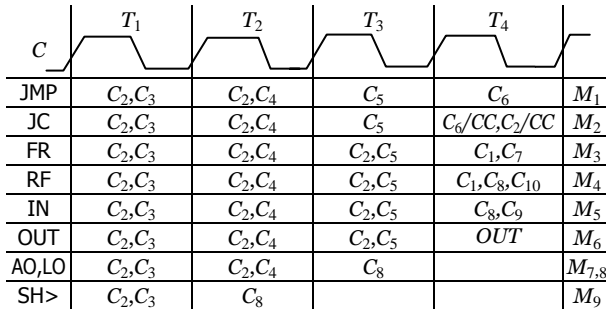


Fig.4.13

cycle  $T_2$  beginning. The clock rising edge in  $T_2$  strobes this byte storing it in the place BY0 of IRG. This is provided by the signal  $C_3 = 1$ . This edge increments the address in ICTR, controlling by the signal  $C_2 = 1$ . This means that the second byte address of the current instruction or the first byte address of the next instruction is calculated. This process illustrated by Fig.4.13, where the control signals are indicated, which are equal to a 1.

The next activities depend on the opcode, which is stored in byte BY0. If it is the code of the instruction JMP, then in cycles  $T_2$  and  $T_3$  the second and third instruction bytes are selected and fixed in IRG, and in the cycle  $T_4$  in ICTR the jump address from the instruction field is fixed by the signal  $C_6$ .

The instruction JC waveforms are distinguished from JMP waveforms only in activities in the cycle  $T_4$ . If the jump condition is satisfied (i.e.  $C_C = 1$ ) then microoperation  $C_6$  is implemented, else microoperation  $C_2$  is (i.e. ICTR increment).

The features of the waveforms for instructions FR and RF are the next instruction address forming in cycle  $T_3$  (signal  $C_2$ ), address transfer in the RAM in cycle  $T_4$  (signal  $C_2$  in FR and signal  $C_8$  in RF). The signal  $Y_0$  provides the word transfer from the RAM through the input I of the multiplexor to the input Q of FM.

In the cycle  $T_4$  of the instruction IN, a byte is stored in FM (signal  $C_8$ ) from the bus DI (signal  $C_9$  provides that). The signal  $OUT$  in cycle  $T_4$  of the instruction OUT tells that the correct data will be set by the next clock edge in the outputs A and DO for its output from the CPU.

Instructions AO and LO are two-byte wide ones, and instruction SH> is one byte wide instruction. They are implemented for three and two cycles, and are finished by the result storing to FM.

From the waveform diagrams (Fig.4.13) the decision is followed that the control signals  $C_i$  ( $i = 1, \dots, 10$ ) have to be formed according to the following equations

$$\begin{aligned}
 C_1 &= T_4 M_3 \vee T_4 M_4; \\
 C_2 &= T_1 \vee T_2 M_1 \vee T_2 M_2 \vee T_2 M_5 \vee T_2 M_6 \vee T_2 M_7 \vee T_2 M_8 \vee T_3 M_3 \vee T_3 M_4 \vee T_3 M_5 \vee T_3 M_6 \vee T_4 M_2 \bar{C}_C; \\
 C_3 &= T_1; \\
 C_4 &= T_2 M_1 \vee T_2 M_2 \vee T_2 M_3 \vee T_2 M_4 \vee T_2 M_5 \vee T_2 M_7 \vee T_2 M_8; \\
 C_5 &= T_3 M_1 \vee T_3 M_2 \vee T_3 M_3 \vee T_3 M_4 \vee T_3 M_5 \vee T_3 M_6; \\
 C_6 &= T_4 M_1 \vee T_4 M_2 C_C; \\
 C_7 &= T_4 M_3; \\
 C_8 &= T_2 M_9 \vee T_3 M_7 \vee T_4 M_4 \vee T_4 M_5 \vee T_3 M_8; \\
 C_9 &= T_4 M_5; \\
 C_{10} &= T_4 M_4,
 \end{aligned}$$

where  $M_i$  ( $i = 1, \dots, 10$ ) is a decoded opcode, i.e. (see Fig.4.13)

$$\begin{aligned}
M_1 &= B_{07} \cdot B_{06} \cdot B_{05} \cdot B_{04} \cdot B_{03} \cdot B_{02} \cdot B_{01} \cdot B_{00} \cdot B_{17} \cdot B_{16} \cdot B_{15}; \\
M_2 &= B_{07} \cdot B_{06} \cdot B_{05} \cdot B_{04} \cdot B_{03} \cdot B_{02} \cdot B_{01} \cdot \bar{B}_{00}; \\
M_3 &= B_{07} \cdot B_{06} \cdot B_{05} \cdot B_{04} \cdot B_{03} \cdot B_{02} \cdot \bar{B}_{01} \cdot \bar{B}_{00}; \\
M_4 &= B_{07} \cdot B_{06} \cdot B_{05} \cdot B_{04} \cdot B_{03} \cdot B_{02} \cdot \bar{B}_{01} \cdot B_{00}; \\
M_5 &= B_{07} \cdot B_{06} \cdot B_{05} \cdot B_{04} \cdot B_{03} \cdot B_{02} \cdot B_{01} \cdot B_{00} \cdot B_{17} \cdot \bar{B}_{16} \cdot \bar{B}_{15}; \\
M_6 &= B_{07} \cdot B_{06} \cdot B_{05} \cdot B_{04} \cdot B_{03} \cdot B_{02} \cdot B_{01} \cdot B_{00} \cdot B_{17} \cdot \bar{B}_{16} \cdot B_{15}; \\
M_{7A} &= B_{04} \cdot \bar{B}_{00}; \quad M_{7L} = B_{04} \cdot \bar{B}_{03} \cdot B_{00}; \\
M_8 &= \bar{B}_{00} \cdot B_{04},
\end{aligned}$$

where  $B_{ij}$  is the  $j$ -th bit of the  $i$ -th byte of the opcode,  $M_{7A}$  and  $M_{7L}$  are decoded opcodes of the arithmetic and logic instructions.

From derived equations and diagrams it is followed, that the FSM contains a set of AND gates and OR gates, decoder of opcodes and the clock cycle counter. Its network diagram is shown in Fig.4.14. The array of  $3 \times 9$  AND gates forms the products  $T_i \cdot M_j$  ( $i = 2, 3, 4; j = 1, \dots, 9$ ). The products  $T_1 \cdot M_j$  may not be derived, because in the cycle  $T_1$  the signals  $C_2$  and  $C_3$  are generated for any instruction. The outputs of AND gates with the indexes  $i$  and  $j$  are OR-ed in the OR gates according to the equations for  $C_k$  ( $k = 1, \dots, 10$ ). It is considered that the products  $T_4 M_2 \bar{C}_C$  and  $T_4 M_2 C_C$  are formed in two steps: 1) the condition flag strobe  $S = T_4 M_2$  is formed, 2) the jump condition signals  $SC_C$  and  $SC_{\bar{C}}$  are formed.

The analysis of equations  $C_k$  shows that they can be minimized, and at this cost the logic hardware can be simplified. This is a usual praxis, but then, regularity and visual properties of the FSM block diagram are lost.

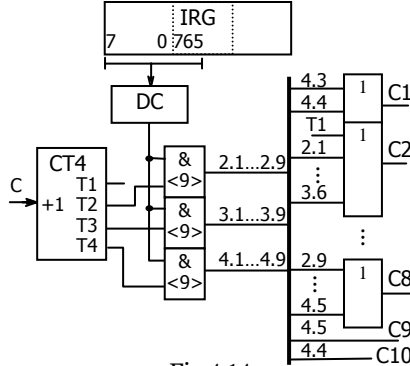


Fig.4.14

# 5. Interfaces

## 5.1 Common busses

The CPUs of first generations were built according to the centralized structure, in which all the units, like magnetic disc storage, I/O terminals, RAM, were attached directly to the CPU, and they have operated under its control. Because the speed both of CPU and peripheral units was relatively small, CPU most of its time was utilized for the peripheral unit service. The common bus interface was introduced at first in the computer PDP-8 in 1965. It has given the units the opportunity to operate independently and then to send the data to each other. In the **common bus** system, all the units are connected to a single bus through a standardized hardware interface. Each pair of the units including CPU can exchange the data. But in a moment only a single data exchange can be executed. Therefore, all the units connected to the bus have to obey the common rules of data exchange. These rules are named as an **interface protocol**. To attach a new device to the bus, its hardware interface and protocol must satisfy the common conditions of the bus.

The units, attached to the bus, are divided into masters and slaves. The **master** occupies the free bus, activates the data exchange, and releases the bus after exchange is done. Any unit can be master, but none couple of masters, or none couple of data sources can be active simultaneously. The information from the master is transferred to every unit, attached to the bus. The **slave** accepts the information if it needed it. It can transfer the data to the master as well, if the master selects this slave.

The electric circuit of the common bus is usually based on the open collector (open drain) buffers or tristate buffers. The open collector bus has a single loading resistor, which consumes the energy when the bus is in the low state, and which value is high (hundreds and thousands of Ohms). The process of sending a 1 after a 0 consists in the loading the wire capacitance through the loading resistor. Therefore, this process is rather slow, and the open collector bus is a low speed bus. The tristate bus has none loading resistor. The opened transistor in the tristate buffer serves as such a loading. Because the open transistor resistivity is rather small (ones and tenths of Ohms), the capacity loading process is much quicker, and such a bus has higher speed.

The bus wire behaves on the high frequencies as a long electric line. All the ends or unhomogenities in it can reflect the signal, generating the noise. To prevent this process, the special loadings are attached to the wire ends. These provide the input and output impedance, which is equal to the line impedance (50–200 Ohms). Many modern ICs, like FPGAs have the I/O buffers with the programmed impedance, providing the proper adjustment to the bus line impedance.

The bus speed is optimized by increasing the signal current or by decreasing the voltage range of logic levels. Then the bus capacitance can be loaded more frequently. The second way is more attractive because of power consumption effectiveness. But by this process, the noise magnitude increases comparatively to the signal magnitude. To increase the noise immunity, the low voltage dual signal

lines (LVDS) are introduced. Such a line consists of a couple of parallel wires, one of them transfers positive signal, and another one – does negative signal. The signal receiver is built as a comparator of signals in both wires. If a noise is accepted by wires, it has the equal magnitudes in both of them, and these magnitudes are subtracted in the comparator inputs, providing the high noise immunity.

Often, especially in SOCs, common busses are implemented on the base of multiplexers, as shown in Fig.1.21. Here the bus speed achieves its maximum, because the wire capacitance is minimized. Such a bus architecture will be discussed below. The another speed limit of the common bus is caused by the data edge **skewing** in inhomogeneous parallel busses. This skewing is a root of the data synchronization problem, when the data are latched in the bus receiver. To solve this problem, the input ports are arranged by the programmed delays, which compensate the data skews. It is an opinion, that in the near future the inter-processor communications will be held using the sequential transfers through a single bit width lines, where this problem is absent. In this situation, the only solution to route the data flows is the use of multiplexor or switching matrix.

The bus lines are divided to address bus, data bus, control lines, power lines and reserved lines. Address lines transfer the source or destination address. Each unit has its own address range and has to distinguish by own if the given address belongs to its range. The data bus can be from 8 to 64 bit wide. Some busses like AMBA have the wide up to 1024 bits. It serves for the data transfer between the units with the given addresses. The control lines serve to point the transaction type (read or write), to indicate if the unit is ready to send or receive data or interrupt signals, to synchronize units.

If the bus provides a set of bus masters, a situation can occur when two masters try to access the bus simultaneously. To resolve this problem the bus **arbiter** is usually used.

The busses are distinguished as synchronous and asynchronous ones. Fig.5.1 illustrates the interaction of a master and a slave in the **synchronous bus**. The master generates clocks  $C$  and sends them through a separate line to all the slaves. It sets the slave address on the bus  $AB$  strobing it by the address acknowledge signal  $AAK$ , and then it sends the request signal  $RRQ$  for data reading or writing.  $AAK$  and  $RRQ$  can be combined in time. The slave outputs the data in the data bus  $DB$  as a response to the master's signals. The synchronous mode in this example consists in that, that all the signals have to appear precisely in accordance with the common clock signal. If the slave could not output the data in time, then it has to activate the line "wait" and deactivate it if the data is ready.

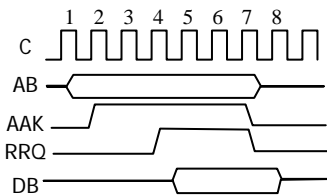


Fig.5.1

Fig.5.2 illustrates the interaction of units attached to the **asynchronous bus**. The master unit outputs address to the line  $A$  and the reading request  $RRQ$ . The last one is simultaneously an **acknowledge** signal of the address correctness. The master activity is marked as  $U_M$ . All the slave units, which activity is marked as  $U_S$ , decode the address. Only selected unit, which address satisfied, outputs the data  $D$



to the bus *DB*, and it acknowledges the data correctness by a signal in the line *DAK*. This signal strobes the data input in the master. After data receiving, the master resets the signal *RRQ* signaling that the data is already not affordable. By this signal, the slave releases the bus, setting it in the state of the high impedance, and it resets the signal *DAK*. It shows that the reading transaction is finished.

Comparing synchronous and asynchronous busses, we can assign the simplicity of the synchronous bus, and the flexibility of the asynchronous bus, which helps to devices with different speeds to operate together.

In the previous examples, the data transfer is performed by a single word and it consists of two steps: address set and data transfer. Such a transfer mode slows down the data array transfer, which is often used, for example, in the data exchange between DRAM and NVRAM. To speed up such transactions, many busses implement the **block data transfer**, in that the first word address is followed by the block data length. Because the neighboring word addresses in the block are different in a 1, both master and slave automatically form them using the built-in counters. Such counters are named as direct memory access (**DMA**) counters, and units, which provide the block transfer, are named as DMA units. As a result, a single word transfer is decreased in the time of address setting, its decoding and acknowledgement.

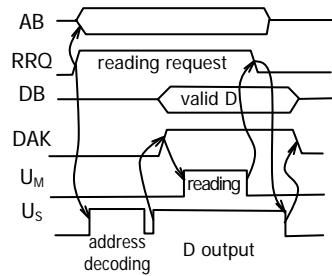


Fig.5.2

## 5.2 AMBA interface in SOC design

A new approach to SOC designs is the use of platform technology. The **platform** is a standard flexible integral architecture, which general properties have to be not exchanged for several years. The platforms can be implemented both in ASICs and in FPGAs. These platforms have libraries that contain pre-designed and pre-verified intellectual property (IP) cores. An **IP core** is a documented project of a module, which can be adapted to the customer needs when it is customized in the SOC. Users can mix-and-match the functional IP core from the library to assist in design of the SOC. To connect IP cores together successfully, they have to be arranged with the standardized interfaces and communicate with a particular bus protocol. Below the AMBA interface is described as a bright representative of on-chip busses, which are widely used in the SOC design.

The ARM processor is a most widely used RISC architecture built in the SOC. The ARM processor is provided with the AMBA bus, which is an open specification from the ARM corporation. This bus can be used not only with the ARM CPU but also with another CPU cores and application specific devices. The properties of the AMBA bus are similar to ones of another standard busses like IBM CoreConnect bus, Altera Avalon bus, VSIA Virtual Component Interface, and others. In the AMBA bus architecture there are three distinctive busses:

advanced system bus (ASB), advanced peripheral bus (APB), and advanced high-performance bus, called the AHB.

Several masters and slaves can be connected to the AHB, but at a time only one master is allowed access. The master to be allowed access is selected by an arbiter. The AHB-APB **bus bridge** serves as a slave on the AHB, and the only master in the APB. The various low performance peripherals on the APB serve as the APB slaves. ASB is an alternative system bus suitable for use where the high-performance features of AHB are not required. Below the AHB is considered, which shows the main properties of the AMBA bus. Its signals are the following:

HCLK – bus clock. All signal timings are related to the rising edge of HCLK;

HRESETn – bus reset signal, is active low and is used to reset the system;

HADDR[31:0] – 32-bit system address bus, which is given by the master;

HTRANS[1:0] – transfer type. Master indicates the type of the current transfer, which can be Nonsequential (code 10), Sequential (code 11), Idle (code 00) or Busy (code 01);

HWRITE – transfer direction, when high, master indicates a write transfer and when low – a read transfer;

HSIZE[2:0] – transfer size, master indicates the size of the transfer, which is typically byte (code 000), halfword (code 001) or word (code 010). The protocol allows for larger transfer sizes up to a maximum of 1024 bits;

HBURST[2:0] – burst type, master indicates if the transfer forms part of a burst. Four, eight and sixteen beat bursts are supported and the burst may be either incrementing or wrapping;

HPROT[3:0] – protection control, master provides additional information about a bus access, which intended for use by any module that wishes to implement some level of protection. The signals indicate if the transfer is an opcode fetch or data access (HPROT[0] = 0 or 1), as well as if the transfer is a privileged mode access or user mode access (HPROT[1] = 1 or 0). For bus masters with a memory management unit these signals also indicate whether the current access is cacheable (HPROT[3] = 1) or bufferable (HPROT[2] = 1);

HWDATA[31:0] – write data bus. It is used to transfer data from the master to the slaves during write operations. Minimum data bus width of 32 bits is recommended. However, this may easily be extended up to 1024;

HSELx – slave select. Decoder each AHB slave has its own slave select signal and this signal indicates that the current transfer is intended for the selected slave. This signal is simply a combinatorial decode of the address bus;

HRDATA[31:0] – read data bus. It is used to transfer data from bus slaves to the bus master during read operations;

HREADY – transfer done. When high, slave indicates that transfer has finished on the bus. This signal may be driven low to extend a transfer;

HRESP[1:0] – transfer response. Slave provides additional information on the status of a transfer. Four different responses are provided, Okay, Error, Retry and Split (HRESP = 00, 01, 10 and 11, respectively);

HBUSREQx – bus request. Master x signals to the bus arbiter that it requires the bus. There is such an signal for each bus master in the system;

**HLOCKx** –locked transfers. When high, master indicates that it requires locked access to the bus and no other master should be granted the bus until this signal is low;

**HGRANTx** – bus grant. By this signal the arbiter indicates that bus master x is currently the highest priority master. Ownership of the address or control signals changes at the end of a transfer when HREADY is high, so a master gets access when both HREADY and HGRANTx are high;

**HMASTER[3:0]** – master number. These signals from the arbiter indicate which bus master is currently performing a transfer and is used by the slaves, which support split transfers to determine which master is attempting an access;

**HMASTLOCK** – locked sequence. Arbiter indicates that the current master is performing a locked sequence of transfers;

**HSPLITx[15:0]** – split completion request. This 16-bit split bus is used by a slave to indicate to the arbiter which bus masters should be allowed to re-attempt a split transaction. Each bit of this split bus corresponds to a single bus master.

Some bus lines can be absent, if needed. For example those, which support the split transfers, can be removed.

The AMBA AHB bus protocol is designed for the use with a central multiplexor interconnection scheme. All bus masters drive out the address and control signals indicating the transfer they wish to perform and the arbiter determines which master has its address and control signals routed to all of the slaves. A central decoder is also required to control the read data and response signal multiplexor, which selects the appropriate signals from the slave that is involved in the transfer. Fig.5.3 illustrates the structure required to implement an AHB design with two masters and two slaves. The multiplexor MUXA selects address and control information from the masters to the slaves, the multiplexers MUXDW and MUXDR transfer the data when writing and reading, respectively.

All transfers must be aligned to the address boundary equal to the size of the transfer. For example, word transfers must be aligned to word address boundaries (that is  $A[1:0] = 00$ ).

For transfers that are narrower than the width of the bus, for example a 16-bit or 8-bit transfer on a 32-bit bus, then the bus master only has to drive the appropriate byte **lanes** (upper or lower halfword, 3-th,..., or 0-d byte). The slave is responsible for selecting the data from the correct lanes.

The AMBA protocol allows **burst** transfers by a master which has been granted bus access. The individual transfers within a burst are called as **beats**. Four, eight and sixteen-beat bursts are defined in the AMBA AHB protocol ( $HBURST = 010, ..., 111$ ), as

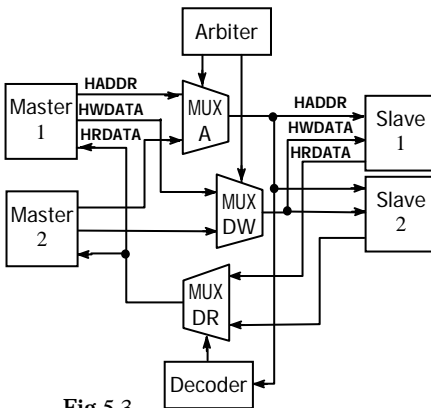


Fig.5.3

well as undefined-length bursts (HBURST = 001) and single transfers (HBURST = 000). Both incrementing and wrapping bursts are supported in the protocol. **Incrementing** bursts access sequential locations and the address of each transfer in the burst is just an increment of the previous address. For **wrapping** bursts, if the start address of the transfer is not aligned to the total number of bytes in the burst (size×beats) then the address of the transfers in the burst will wrap when the boundary is reached. For example, if the start address of the transfer is 0x34, then it consists of four transfers to addresses 0x34, 0x38, 0x3C and 0x30. Bursts must not cross a 1kB address boundary. The first beat of the burst transfer has to be of Nonsequential type, and the others – of Sequential type.

The address and data of the different beats in a single burst are transferred in a pipelined fashion. A write burst which writes data  $D_1$ ,  $D_2$ ,  $D_3$  to addresses  $A_1$ ,  $A_2$ ,  $A_3$  respectively is shown in Fig.5.4. Note that data  $D_i$  and address  $A_{i+1}$  are transmitted in the same clock cycle on the HADDR and HWDATA lines. Thus the address and data phases of consecutive beats within a burst can overlap.

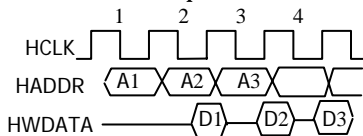


Fig.5.4

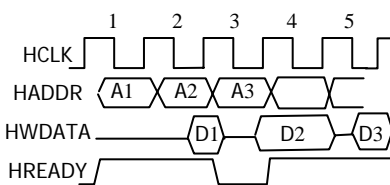


Fig.5.5

The protocol allows a slave to insert wait cycles by deasserting a HREADY signal if the slave is not ready to service a transfer. This extends the data phase of a transfer. Due to the pipelined nature of the bus, the address phase of the next transfer also has to be extended. Fig.5.5 shows the writing of  $D_1$ ,  $D_2$ ,  $D_3$  to addresses  $A_1$ ,  $A_2$ ,  $A_3$  with the insertion of a single wait cycle in the transfer of  $D_2$ .

In order to prevent an excessive number of wait cycles, the protocol allows the release of bus access to the other masters. This is co-ordinated by the slave which either informs the arbiter of its temporary inability to service a

master (a Split response) or informs the master to retry the transfer (a Retry response). The provision of **split transfers**, that is, temporarily suspending a transfer and resuming it later when the slave is ready, raises many important questions. The pipelined nature of the AMBA bus further complicates the situation.

The AHB protocol specifies a certain behaviour that must be respected. Firstly, the master must perform pipelined accesses: every transaction must be performed in two phases. First comes an **address phase** during which the address and control signals are driven. At the end of this phase, the slave selected by the address samples the address and control signals and begins its response during the **data phase**. The response includes the driving of certain control signals and either the emission of the read data or the sampling of the write data at the end of the cycle. This rule is referred to as the *Pipeline rule*.

Then the slave can drive HREADY low to stretch the length of a bus cycle. The

master must be able to stall its execution to respect the slave's request. This is the *Stretch* rule. The AHB protocol also has provisions for several bus masters: only one master can have access to the slaves at a given time. All of the other masters must wait until the bus is assigned to them. This is the *Arbitration* rule.

Several accesses must be made in an uninterrupted, or atomic, fashion. The AHB protocol offers the possibility of performing locked access sequences, that the arbiter cannot interrupt to grant the bus to another master. Before the first address phase, the master must warn the system that the locked transfers are about to begin by asserting the HLOCK signal. The signal must be de-asserted during the address phase of the last access. This is the *Lock* rule.

Finally, a slave can issue various responses to a master's request. The Error response indicates that the access has failed. The Retry and Split responses must be handled in a special way: every clock cycle, the master must observe the status of HRESP. If one of the two aforementioned responses is given, then the master must immediately drive the Idle value on its HTRANS output. This cancels the address phase that followed the one that caused the response. On the following bus cycle, the master must retry the access that had caused the unusual response. This rule is referred to as the *Exception* rule.

Consider the design of the master core, which has to be attached to the CPU for the AMBA interface communication, so called, **wrapper**. The master FSM provides the fulfilment of the interface protocol. The first step is to set up an FSM initial state that respects the *pipeline* rule. The FSM diagram contains a single step over the node RUN. Events or assignments that must occur at every bus cycle can be performed on transition (1), which goes back to this node. The next step is to include the *Stretch* requirement. The core must be stalled during the extra clock cycles of the bus cycle, waiting on the resuming the process at every cycle. Consider the core has a static design, then it is possible to stall it by modulating the clock signal (i.e. stopping it at strategic moments). Fig.5.6 shows a FSM state graph that can handle bus cycle stretching: if HREADY=0 at the end of a cycle, the state becomes NOT\_READY, and the core's clock is stalled (transition 2). The FSM stays in the NOT\_READY state (3) until HREADY = 1 (4).

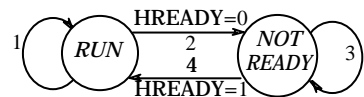


Fig.5.6

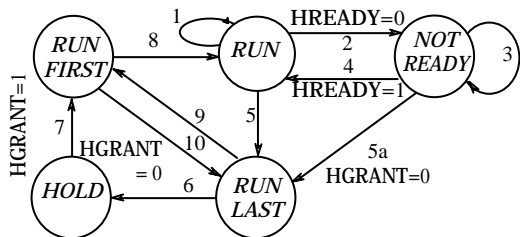


Fig.5.7

The next requirement to implement is the *Arbitration*. A master should not attempt any access without first making sure that the bus belongs to it. If the bus is granted to another master, it must wait. A master that loses the bus has a

final bus cycle in which it can accomplish its data phase while another master prepares its address phase. Fig.5.7 shows the new states that implement the *Arbitration* rule.

Here, if the master notices that it is losing the bus ( $HGRANT = 0$ ), then it enters the  $RUN\_LAST$  state (transition 5 or 5a), in which it completes its data phase and prepares the address phase for the time when the bus will be given back to it. At the end of the bus cycle, the FSM enters the  $HOLD$  state, where the core's clock is inhibited in order to keep the address from changing again. The master must sample the incoming data on the transitions (6) and (9) and "feed" it to the CPU when the clock will be released. Once the bus is finally given back to the master, the core enters the  $RUN\_FIRST$  state (7). The clock is still inhibited here, to keep the core's address from changing on entry. The address that was prepared during  $RUN\_LAST$  finally becomes visible on the bus and the transaction can finally take place. The master stays in the  $RUN\_FIRST$  state until the end of the address phase (i.e. a bus cycle), where it returns to the  $RUN$  state (8). There are also transitions that are taken if the arbiter grants the bus immediately after taking it away (9) or vice-versa (10).

A wrapper requiring the *Lock* requirement usually has a signal that allows it to indicate that it is beginning a locked sequence. This signal, say  $MLOCK$ , can usually be driven to the AHB signal  $HLOCK$ .  $HLOCK$  must be asserted on the bus cycle that precedes the first address phase of the locked sequence, and to be de-asserted during the last address phase.  $MLOCK$  may be asserted or deasserted earlier or later. If it is asserted or de-asserted early, it is possible to delay it in the  $RUN$  state. If it is de-asserted late, the bus is locked for an extra cycle. The problem is the situation where  $MLOCK$  is asserted late (i.e. in the same time as the first address phase of the locked sequence). Then, the core must be delayed by one bus cycle in order to allow the arbiter of the bus to become aware of the change on  $HLOCK$ . Fig.5.8 shows the resulting FSM graph that allows this.

When  $MLOCK = 1$  the master goes into the  $LOCK\_FIRST$  state on the next bus cycle (transition 11). This state sets a flag and stalls the core for one bus cycle (by inhibiting the clock) before returning to the  $RUN$  state (12). With the flag activated, the master don't return to the  $LOCK\_FIRST$  state, but must check at every bus cycle (1) to determine if it must deactivate the flag ( $CORELOCK = 0$ ).

Stalling the core causes a problem on the AHB bus: the access that caused the transition to  $LOCK\_FIRST$  appears as two identical accesses on the bus. The master must thus disable one of the "two" accesses by driving  $HTRANS = IDLE$ . It makes more sense to disable the first access than the second, so there must be another condition in the  $RUN$  state that detects ( $CORELOCK = 1$  and  $LOCK\_FLAG = 0$ ) and drives  $HTRANS = IDLE$  if the condition is true.

The final requirement to observe is the *Exception* rule. During the data phase of an access, the slave can indicate an error or an exceptional situation by using the  $HRESP$  signal. The master usually has to perform the task itself, hiding the details of the procedure from the CPU. In order to handle the exception transparently, the master has to halt the CPU immediately after the address phase that caused the faulty response. The CPU remains blocked until the exception is handled. Fig.5.8 shows how the control unit handles the exceptions.

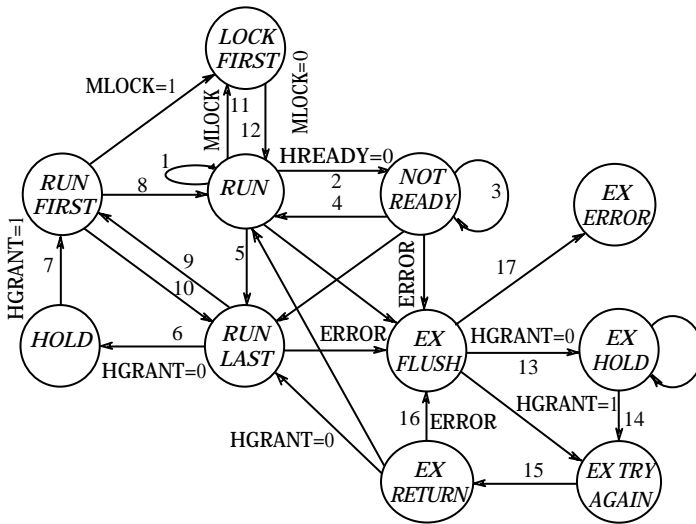


Fig.5.8

The FSM enters the EX\_FLUSH state upon detecting an exception. This drives the HTRANS = IDLE signal as specified by the protocol. If the exception was either RETRY or SPLIT, the wrapper proceeds to either the EX\_HOLD state (if the bus is not available) via transition (13) or the EX\_TRY\_AGAIN state (14). In the EX\_HOLD state, the wrapper simply waits for the bus to be granted. In the EX\_TRY\_AGAIN state, the wrapper drives the address and control signals that had caused the exception. This implies that the wrapper keep track of these values every time the RUN or RUN\_LAST states are entered (even when RUN loops on itself). On the next bus cycle, the wrapper moves on to EX\_RETURN (15), where the slave can respond to the previous address phase. If the access causes another exception, the wrapper returns to EX\_FLUSH and the process begins anew. Otherwise, the wrapper returns to the RUN or RUN\_LAST state. The clock restarts, allowing the core to pick up any read data that it might require. The ERROR response is handled differently (17), the designer is free to implement any behaviour that is appropriate.

The next step is the FSM design on the base of its state diagram on Fig.5.8. This diagram becomes more concrete by adding the proper labels to all the nodes and edges. The resulting FSM can be synthesized as it is shown in the chapter 4.5.

The derived graph serves as an example. The real wrappers can support both the complex and simple interface protocol depending on the system functionality.

## List of abbreviations

|         |  |       |  |
|---------|--|-------|--|
| ALU     | arithmetic and logic unit                  | IFN   | instruction fetch network                |
| ASIC    | application specific<br>integral circuit   | IRG   | instruction register                     |
| ASSP    | application specific<br>standard product   | KM    | Karnaugh map                             |
| BF      | Boolean function                           | LC    | logic cell                               |
| CAD     | computer-aided design                      | LE    | logic element                            |
| CD      | encoder                                    | LED   | light emitting diode                     |
| CISC    | complex instruction set<br>computer        | LN    | logic network                            |
| CMOS    | complementary metal-<br>oxid-semiconductor | LRU   | least recently used                      |
| CI      | carry in bit                               | LSI   | large scale integration                  |
| CO      | carry out bit                              | LSM   | multipurpose summator,<br>ALU            |
| CPLD    | complex programmable logic<br>device       | LSB   | least significant bit or byte            |
| CPU     | central processing unit                    | LUT   | look-up table                            |
| CTR,CTn | counter, counter modulo n                  | MNOS  | metal-nitrogenium-oxid-<br>semiconductor |
| CU      | control unit                               | MPC   | microprogram controller                  |
| DC      | decoder                                    | MPU   | multiply unit                            |
| DFF     | D-type flip-flop                           | MSB   | most significant bit or byte             |
| DRAM    | dynamic RAM                                | MU    | memory unit                              |
| DSP     | digital signal processing                  | MUX   | multiplexor                              |
| EEPROM  | electrically erasable PROM                 | NVRAM | non-volatile RAM                         |
| FA      | full adder                                 | PC    | program counter                          |
| FET     | field effect transistor                    | PLA   | programmable logic array                 |
| FIFO    | first in—first out—type stack              | PROM  | programmable ROM                         |
| FF      | flip-flop                                  | RAM   | random access memory                     |
| FM      | fast memory, register array                | RG    | register                                 |
| FPGA    | field programmable gate<br>arrays          | RISC  | reduced instruction set<br>computer      |
| FPM     | fast page mode                             | ROM   | read only memory                         |
| FRAM    | ferroelectric RAM                          | SDRAM | synchronous DRAM                         |
| FSM     | finite state machine                       | SHU   | shifter unit                             |
| HA      | half adder                                 | SM    | adder, summator                          |
| HDL     | hardware description language              | SOC   | system-on-the-chip                       |
| IC      | integral circuit                           | SRAM  | static RAM                               |
| ICTR    | instruction counter, the<br>same as PC     | TTL   | transistor-transistor logic              |
|         |  | WD    | Waych diagram                            |
|         |  | WE    | writing enable                           |
|         |  | XOR   | exclusive OR                             |



## Bibliography

1. Алексенко А.Г. Основы микросхемотехники. –3-е изд., перераб. и доп. – М.: Юнимедиастилл, 2002. – 448 с.
2. Бабич М.П., Жуков І.А. Комп'ютерна схемотехніка: Навчальний посібник. – К.: МК-Прес, 2004. – 412 с.
3. Бойко В.А. и др. Курсовые и дипломные проекты. Требования к оформлению документации. – К.: Корнійчук, 2003. – 176 с.
4. Жмакин А.П. Архитектура ЭВМ. –СПб: БХВ, 2006. -320 с.
5. Корнейчук В.И. Тарасенко В.П. Основы компьютерной арифметики. – К.: Корнійчук, 2002. – 176 с.
6. Микропроцессорные системы: Уч. пос. для вузов/ Е.К. Александров и др. – СПб.: Политехника, 2002. –935 с.
7. Поляков А.К. Языки VHDL и VERILOG в проектировании цифровой аппаратуры. – М.: Солон, 2003.- 320 с.
8. Процюк Р.О., Корнейчук В.И., Кузьменко П.В., Тарасенко В.П. Компьютерная схемотехника (краткий курс). – К.:Корнійчук, 2006. –433 с.
9. Рябенкий В.М., Ушкаренко О.О. MAX+plus II. Основы проектирования цифровых устройств на ПЛИС. –К.:Корнійчук, 2005. –253 с.
10. Самофалов К.Г. и др. Цифровые ЭВМ: Теория и проектирование –3-е изд. – К.: Вища школа, 1989. – 424 с.
11. Сергиенко А.М. VHDL для проектирования вычислительных устройств. – К.: Корнійчук, ДиаСофт, 2003. – 208 с.
12. Сергиенко А.М., Корнейчук В.И. Микропроцессорные устройства на ПЛИС. – К.:Корнійчук, 2005. – 108 с.
13. Схемотехніка електронних систем: В 3-х кн. Кн.2. Цифрова схемотехніка: Підручник/ В.І. Бойко та ін.- 2-е вид., доп. і перероб.-К.: ВШ, 2004.-423 с.
14. Хамахеер К., Враешич З., Заки С. Организация ЭВМ. – СПб.: БХВ – Питер, 2003. -848 с.
15. Шкурко А.И., Корнейчук В.И. и др. Компьютерная схемотехника в примерах и задачах. – К.: Корнійчук, 2003. –144 с.
16. AMBA™ Specification. – ARM Limited, 1999. –230 p. (<http://www.arm.com>)
17. Bertola M., Bois G. A methodology for the design of AHB bus master wrappers // Proc. Euromicro'2003. –2003. – p. 223-233.
18. Dewar R.B.K., Smosna M. Microprocessors: A Programmer's View. – NY: McGraw-Hill Pub., 1990. – 462 p.
19. Keating M., Bricaud P. Reuse Methology Manual. 2-nd ed., Kluwer Academic Pub. –1999. – 286p.
20. Roth C.H. Digital Systems Design Using VHDL. –Boston: PWS Pub., 1998. – 470 p.
21. The Synthesis Approach to Digital System Design / P.Michel, U.Lauther, P.Duzy, – ed-s. Kluwer Academic Pub., – 1993. –415 p.
22. Uyemura J.P. A First Course in Digital Systems Design: An Integrated Approach. – CA: Pacific Grove, 2000. – 495 p.