A.M. Sergiyenko

Computer Architecture

Part 2. Parallel Architectures

Kyiv-2016

CONTENTS

CONTENTS.......1

ABBREVIATIONS.......3

3 PARALLELISM OF.......6

SINGLE-PROCESSOR COMPUTERS.......6

3.1 Pipelined and vector data processing.......6

3.2. Vector pipelined computers.......16

3.3 Superscalar processors.......21

3.4. Methods of superscalar processor performance increasing.......25

3.5. Multithreading.......35

3.6. Memory access parallelism.......39

4. MULTITASKING AND DATA PROTECTION.......47

4.1. Virtual memory.......47

4.2. Computer multitasking.......52

4.3 Memory protection.......55

4.4. Memory management in the IA-32 architecture.......58

4.5. Virtual memory of the IA-64 architecture.......69

4.6. Problems.......71

5. MULTIPROCESSOR ARCHITECTURE.......73

5.1. Fundamentals of multiprocessor architectures.......73

5.2. Processors with the SIMD parallelism.......89

5.3 Multiprocessor Architectures.......97

5.4 Graphics accelerator architecture.......108

6 APPLICATION SPECIFIC PROCESSORS.......111

6.1 Introduction.......111

6.2 Microcontrollers.......112

6.3 DSP microprocessors......119

6.4 Processors with hardware control. Configurable computers......122

6.5 System on chip in FPGA......128

7 COMPUTER ARCHITECTURE PROSPECTS......132

LIST OF RECOMMENDED LITERATURE......137

ANNEX 1......139

ABBREVIATIONS

ALU — arithmetic and logic unit

APIC — Advanced Programmable Interrupt Controller

ARM — Acorn (Advanced) RISC Machines

AS — Architecture State

BHT — Branch History Table

BTB — Branch Target Buffer

CISC — complex instruction set computer

COMA — Cache-Only Memory Architecture

CPL — Current Privilege Level

CPU — central processing unit

CRC — Cyclic Redundancy Check

CUDA — Compute Unified Device Architecture

DMA — Direct Memory Access

DPL — Descriptor Privilege Level

DRAM — Dynamic Random Access Memory

DRIS — Deferred scheduling register Renaming Instruction Shelf

DSMA — Distributed Shared Memory Architecture

DSP — Digital Signal Processing

GDT — Global Descriptor Table

GPIO — General Purpose Input-Output

GPU — Graphic Processing Unit

HDD — Hard Disc Drive

HSA — Heterogeneous System Architecture

IA-32 — 32-bit Intel architecture

IDT — Interrupt Descriptor Table

IoT — Internet of Things

IP — Instruction Pointer

ITLB — Instruction Translation Lookaside Buffer

I2C — Inter-Integrated Circuit

JTAG — Joint Test Action Group

LDT — local descriptor table

LP — Logical Processor

MCM — MultiChip Module

MIMD — Multiple Instruction flows — Multiple Data flows

MIPS — million instructions per second, microprocessor without interlocked pipeline stages

MFLOPS — mega floating point operations per second

MMU — Memory Management Unit

MMX — MultiMedia extension

MPI — Message Passing Interface

MRMW — Multiple Readers — Multiple Writers

MRSW — Multiple Readers — Single Writer

MT mode — Multi-Task mode

MTA — Multi-Threaded Architecture

NUMA — Non-Uniform Memory Access architecture

OpenCL — Open Computing Language

OS — Operational System

PC — Program Counter, Personal Computer

PKR — Protection Key Register

PRAM — Parallel Random Access Machine

PU — Processing Unit

PVM — Parallel Virtual Machine

RAM — Random Access Memory

RAS — Row Address Select, Return Address Stack

RAT — Register Alias Table

RID — Region IDdentifier

RISC — Reduced Instruction Set Computer

ROM — Read-Only Memory

RPL — Request Privilege Level

SAXPY — Single precision A multiplied by X Plus Y

SIMD — Single Instruction flow — Multiple Data flows

SIMT — Single Instruction — Multiple Threads

SM — Streaming Multiprocessor

SMP — Symmetric MultiProcessor

SMT — Simultaneous MultiThreading

SoC — System on a Chip

SPI — Serial Peripheral Interface

SPMD — Single Program — Multiple Data

ST mode — Single-Task mode

T-cache — Trace cache

TLB — Translation Lookaside Buffer

TR — Task Register

TSS — Task State Segment

UART — Universal Asynchronous Receiver-Transceiver

UMA — Uniform Memory Access architecture

VLIW — Very Large Instruction Word

VPN — Virtual Page Number

VSMA — Virtual Shared Memory Architecture

μop — micro-operation

3 PARALLELISM OF SINGLE-PROCESSOR COMPUTERS

3.1 Pipelined and vector data processing

3.1.1 Basic types of parallelism in computers

The main feature of the von Neumann computer architecture reviewed above is the inseparable implementation of all actions. This means that all the instructions and microinstructions are indivisible and are strictly implemented in sequence. The instruction inseparability or atomicity means that its implementation cannot be slowed down and another action be performed during this situation with some elements of the memory. For example, the following instruction could not start running before the end of the previous instruction, the datum is read by the running instruction from RAM strictly after it was written there by the previous instruction A set of instructions can not be executed in several ALUs, i.e. in parallel. Therefore, the speed limit of the von Neumann processor is determined by the period of the instruction execution. It is limited by the amount of delays of instruction decoder, ALU, program RAM, data RAM.

In this section, the architecture details are considered, which help to do without the action atomicity and to speed up the computation due to the parallelization. First, some principles of the computer design are recalled.

The parallel processing principle. To achieve the high performance and (or) the reliability of computer computations, the independent control or calculation steps are distributed among several operating and control machines (or CPUs), which are connected through some switching system.

The pipeline processing principle of information means that complex operation sequence is divided into several successively performed steps

(micro steps) so that they can be performed in parallel for the flow of such operations.

A few definitions are added:

Parallelism is an ability of simultaneous execution of independent arithmetic and logical operations or service. There are three main forms of parallelism:

— natural or vector parallelism;

— parallelism of independent branches;

— parallelism of related operations or scalar parallelism.

The essence of the ***parallelism of independent branches*** is that the independent software branches can be allocated in a program which solves the large problems, which are processed in parallel.

Under ***natural parallelism***, the ability to process the independent data by the same algorithm is understood above all. This is, for example, the ability of simultaneous processing the elements of data vectors (***vector parallelism***), performing a number of identical or similar tasks for different sets of input data.

The program for the von Neumann machine is a list of operations performed in series. The ***scalar parallelism*** means that the subsets of these operations can be performed in parallel if there are no dependencies between them.

Synchronization act is a moment of time, when a portion of calculations transfers the control to other portions, such as the moment, when an instruction passes the control to another one, a computational process activates another one, a subroutine sends the results and calls the next one for execution.

Computation grain is the average time period between adjacent synchronization acts in the parallel computing, or the synchronization period.

Regarding the scalar parallelism, the term ***fine-grained parallelism*** (synchronization period is 1 – 100 clock cycles of CPU) is often used. This is different from the ***medium-grained*** (about 100 – 10^5 clock cycles) and ***coarse-grained parallelism*** (approximately over 10^5 cycles). The medium-grained and coarse-grained parallelism usually consider the vector parallelism and the parallelism of independent branches, respectively.

Each type of computer architecture can effectively perform only those algorithms, that have inherent granulation of computations. Next, the architecture of pipelined RISC-processor is considered that implements the fine-grained parallelism at the microarchitecture level.

3.1.2 RISC-processor instruction pipeline

The main point for the development of pipelined computers was the justification of the method which was called "the water pipeline principle" by the academician S. A. Lebedev in 1956. First of all, the instruction pipeline was implemented, on which the Soviet computer БЭСМ-6 (1957 – 1966) and English machine ATLAS (1957 – 1963) were built almost simultaneously.

The instruction pipeline made it possible to get the performance of 1 million operations per second in the computer БЭСМ-6. Further, the instruction pipeline was improved and became an essential element of all high-speed computers, such as computers IBM/370 and EC10xx.

Since the mid-80s of the last century, all microprocessors were built on the RISC-microprocessor architecture, which instruction pipeline is a characteristic feature.

The RISC architecture was considered above in terms of its instruction set. Recall that one of the principles of the RISC-processor design is the implementation of a single instruction in one clock cycle. This principle becomes possible through the pipelined instruction execution.

By the pipelined instruction execution, the instruction implementation is divided into a specific sequence of separate phases. This division into phases is implemented in the architectures in different ways. But in general, this division is as follows:

— instruction fetching (IF). Loading instructions from the instruction memory into the instruction register of the processor core;

—instruction decoding (ID). Due to the simplicity of the RISC instructions, ID is very fast;

— reading the operands (RO). Because the operands are often found in the register memory of the RISC-processor, RO is very fast;

— operation (OP). It is actual instruction calculating;

— writing results (WR) back in registers or other memory.

If the sequence of instructions is executed without conditional branches, the phases of these neighboring instructions can be performed in parallel instruction pipeline stages, as shown in Fig.1.22.

Because the instruction format is simple, its length (32 bits) is known, the instruction fetching is simple. CPU reads the instruction words from memory without the need to determine the length of the instruction, as in the case of the CISC processor. The instruction decoding is also simplified, as the number of the instruction formats is minimized, the opcode, operand and address fields are always in the same position of the instruction word.

The instructions are divided into a set of instructions addressing RAM and a set of arithmetic instructions. This simplifies the RO phase. In the arithmetic instructions, the operand reading occurs only from the register memory and is usually performed by the direct addressing.

Due to the fact that all instruction phases are of nearly the same complexity, they are usually performed in the instruction pipeline. For example, a piece of the program

```
ADD R1,  R2,  R3 ;         R1=R2+R3 —  instruction 1
ADD R4 , R5, R6 ;  —  instruction 2
ADD R7 , R8,  R9 ; —  instruction 3
ADD R10,R11,R12; —  instruction 4
```

is performed as follows. As the instruction 4 selected, the instruction 3 is decoded and its operands are selected for the instruction 2, which performs addition, the result of the instruction 1 is written in the register R1 (see Fig 1.22). Although the implementation of a single instruction takes four cycles, the average processor performance is approaching to one instruction per clock (with a continuous stream of independent instructions).

In the previous example, all the instructions are independent. A special example occurs, when the neighboring instructions depend on data:

```
ADD R1, R2, R3; R1=R2+R3 —  instruction 1
ADD R4 ,R1, R6; —  uses R1, writes R4
ADD R7 ,R4, R9; —  uses R4, writes R7
```

In this program, each instruction needs the result of the previous instruction. According to the instruction pipeline phases, it is impossible, since the phase of read and write of neighboring instructions are mismatched (Fig. 1.22). But this problem is solved through a method of the ***result forwarding***. During this computations, the instruction result is sent immediately as an operand for the next instruction, as it is shown in Fig. 1.23.

The instructions which are downloading data and saving them in RAM are different from the ALU instructions, that actually the data reading lasts not for one but for several cycles due to low performance of RAM and its connection through the interface.

For example, the addition instruction in the next program piece

```
L       R3, OPER
ADD  R4, R3, R2
```

can not be performed in the next clock cycle after the operand OPER reading because it is implemented for more than one clock cycle, for example, for 2 (if RAM is a cache memory).

But if the NOP instructions are inserted between instructions:

```
L       R3, OPER
NOP
ADD R4,R3,R2
```

it increases the delay between the instructions in one clock cycle, which guarantees the timely appearance of the operand OPER in a register R3. These NOP instructions are usually inserted automatically by the compiler.

Often the compiler rearranges the instructions so as to minimize the number of additional NOP instructions, such as:

Initial program	*Program after optimization*
L R2,OP1	L R2,OP1
L R3,OP2	L R3,OP2
NOP	L R5,OP3
ADD R4,R2,R3	L R6,OP4
L R5,OP3	ADD R4,R2,R3
L R6,OP4	ADD R7,R5,R6
NOP	
ADD R7,R5,R6	

After this instruction reshuffle, the program meaning remains the same. But the delay between the operand reading from RAM and use them in the arithmetic instructions is permissible, so the NOP instruction insertion is not required.

The real programs contain a large number (approximately 25%) of conditional and subroutine call instructions. These instructions implementation has two aspects. First, in the cycle, when the processor recognizes the following instruction, the instruction that is following it (placed at the following address) is already in the pipeline and begins to be handled. Second, an instruction that really needs to be done after the branch instruction, cannot enter the pipeline immediately in the next cycle. So the pipeline needs to be stopped and be cleaned from the instructions that should not be implemented.

The easiest way to speed up the instruction execution is the method of the ***delayed branch***. Thus when the branch instruction is recognized, the

opportunity is given to one or more instructions running in the pipeline, before the branch itself is done.

Consider the following sequence of instructions:

```
     L    R9, #0
     JMP  M1
     ADD R9, R9, #1
      . . .
M1:  ADD  R9, R9,#1
```

When the JMP instruction goes to the pipeline, the instruction ADD is already in the pipeline and it starts to be executed. When the branch to label M1 is actually made and the second instruction ADD is done, the number 2, but not the number 1 occurs in the register R9, as it would be when the program is running in the normal von Neumann processor.

A method of placing and performing the instructions after the branch instruction is called as the ***branch delay slot*** method. This method helps to the compiler to minimize the NOP instructions. For example, the parameters, which are transmitted to the routine, can be loaded by instructions placed after the CALL instruction.

```
Initial program              Program  after optimization

L     R3, OP1                CALL SUBR
CALL SUBR                     L     R3,  OP1
NOP
```

The first subroutine instruction that is called can be put in the delayed branch slot as well. Then the call address is increased to this instruction length to "circumvent" this instruction during a call.

The programming of the delayed branch is an unusual thing for a programmer. But it is quite clearly and correctly performed by a compiler. Thus, the compiler for the MIPS architecture is able to rearrange the instructions in the program to fill 70 to 90% of the delayed branch slots.

3.1.3 CISC processor instruction pipeline

Modern CISC processors also have the instruction pipelines, through which they achieve high clock speed. Consider for example the evolution of the instruction pipeline of processors Intel Pentium, ..., Pentium-4, which are synonymous of P5, P6, P7.

The structure, or more precisely, the microarchitecture of the P5 processor has a five-staged instruction pipeline with the stages: instruction sampling, two stages of the decoder, operation stage and writing the result. This made it possible to have a clock speed of about 100 MHz.

In the microarchitecture P6 (PentiumPro, Pentium II, Pentium III), the instruction pipeline performs both RISC and CISC instructions, but the last instructions are performed ineffectively. It has the enlarged pipeline with 12 stages. Through the pipelining and modernized ASIC technology, the clock speed was increased to 1.4 GHz.

In the P7 microarchitecture while performing the CISC-instructions they are replaced dynamically by the chains of RISC-instructions, which are executed in the pipelined mode (see. hereinafter the "trace cache"). Thus, the reading and converting the CISC- instructions to the RISC-instructions is performed in 8 pipeline stages, and the RISC-instructions are implemented in 20 stages. This, Pentium-4 in 2005 reached a clock speed of 3.8 GHz. An interesting detail of this microarchitecture is the fact that two of the 28 pipeline stages generally do not perform any computations. They are intended to equalize the signal delay during the data transfer from one part to another part of the chip. This is because the area of the CISC-chip is too large to send signals through it without the data buffering and pipelining.

Another shortcoming, which was found in this architecture is a great delay of the pipeline flush during the unpredicted branch (see "Conditional branch predictions"). The performing reading the RAM after the recent

writing to it causes a huge pipeline slowdown as well (see "Memory access consistency").

Thus, the CPU clock speed is not a reliable indicator of the processor performance with a particular architecture. The instruction pipeline stage number should be chosen so as to ensure the maximum performance of the microarchitecture (eg, solving the test problems) at the maximum clock frequency that is acceptable to it. Unlike CISC-processors, RISC-processors have a smaller chip area and a shorter instruction pipeline, so they have a higher clock speed and higher overall performance and less power consumption.

3.1.4 Software pipelining

When hardware pipelining, the instruction implementation is divided into stages, such as instruction fetching, execution, and writing. But the processor executes a single instruction per clock cycle only after filling the pipeline.

Similar processes occur in the software pipelining. By it, a single loop of the loop nest is divided into S stages, such as data reading from RAM, intermediate result calculation and data writing in RAM. Then the first stage of the *i*-th iteration is executed, the second stage of the *i*–1-st iteration is, etc. in a single program loop. This loop is arranged in the way, that the instructions of the pipeline filling are placed before the loop and the flushing instructions are placed after it. Below is a program example of adding two arrays with the program pipelining.

Initial program (11 cycle iteration)	After optimization (8 cycle iteration)
L R2, ARRA ; array A address	L R2, ARR1 ; array A address
L R3, ARRB ; array B address	L R6, [R2] ; loading A(0)
L R4, #(N*4) ; array length	L R3, ARRB ; array B address
L R5, #4 ; address increment	L R7, [R3] ;loading B(0)

```
M1: L R6, [R2] ; loading A(i)
    L R7, [R3] ; loading B(i)
    NOP
    NOP
    ADD R6,R6,R7 ;   A(i)+B(i)
    S     R6, [R2] ; sum storing
    ADD R2,R2,R5; address modific.
    ADD R3,R3,R5; address modific.
    SUB R4,R4,R5 ; end of iteration?
    JNE M1 ; jump to cycle
    NOP
```

```
    L    R4, #(N*4) ; array length
    L    R5, #4 ; address increment
M1: ADD R3,R3,R5; address modific.
    L    R7, [R3] ;  loading B(i+1)
    ADD R8, R6, R7 ;A(i)+B(i)
    S    R6, [R2] ; sum storing
    ADD R2,R2,R5
    SUB R4,R4,R5
    JNE M1 ;
    L R6, [R2]; loading A(i+1)
```

Often, the software pipelining is used in the floating point calculations, when the latent delay of the pipeline is very large (more than 6 – 8 cycles). It is widely used in the signal microprocessors, and in VLIW-microprocessors, which execute plenty of parallel actions in a single instruction cycle.

3.1.5 Problems

1) Estimate the granularity of computations in your PC at the intruction level.

2) Estimate the granularity of computations in your PC at the level of the OS function calls.

3) Estimate the granularity of computations using the Web service and TCP/IP protocol.

4) The RISC program contains 25% of branch instructions. The instruction pipeline has 5 stages. Estimate the average program speed in instructions per clock cycle when the branch delay slot method is not used.

5) In the RISC processor, the data reading from RAM lasts 3 clock cycles, the program contains 25% of data loading instructions. Estimate the speedup of the program when the NOP instructions after the loading instructions are removed during the program optimization.

3.2. Vector pipelined computers

3.2.1 Vector parallelism algorithms

In the algorithms of the structured data processing, the vector operation that has the natural parallelism is the most common. The ***vector*** is a one-dimensional array with a size, that is known at compile time. Often a vector is formed from a multidimensional array. In the sequential languages, such as Fortran, some vector operation is expressed by the DO cycle:

```
      DO 10 I = 1,N
10    C(I) = A(I) + B(I+16);
```

The vector operations like addition, multiplication, division, comparison, etc. are universal in nature and often are included in the parallel programming languages. The vector operation, given above by a cycle, is represented in the MATLAB language more clearly:

```
C = A (1: N) + B (16: N + 16);
```

The areas of the vector use are operations on large arrays, digital signal and image processing, linear algebra problem solving, modeling physical environments, meteorology, optimization, design automation, etc.

3.2.2 Arithmetic pipeline

In the late 60's, the arithmetic pipeline was introduced into the processor datapath. It has become the central device in the supercomputer CRAY-1 (1976). The ***arithmetic pipeline*** has the multistaged structure, each stage performs a single micro-operation for the data stream calculating. Then, a set of data is processed in this pipeline but in the different stages.

This arithmetic pipeline enables to efficiently compute the vectors of numbers, but during the processing of the separate numbers, the most of its stages are idle, which is shown below.

The supercomputers usually solve the numerical problems using floating-point double precision data. Such operations can be divided into several dozens of micro-operations, which are performed in the respective stages. As a result, such a pipeline has the maximized clock frequency up to several gigahertz. Therefore, it possible to obtain the performance of the arithmetic pipeline up to several hundreds of millions floating point operations per second, i.e. hundreds and thousands ***MFLOPS*** (Mega Floating Point Operations Per Second).

The majority of supercomputers in 70–80s and early 90s were the vector pipelining computers. But then they did not survive the competition with the multiprocessor computers.

The floating-point ALU of the modern microprocessors is usually the arithmetic pipeline. The floating point signal microprocessors and graphic coprocessor can be classified as the vector-pipelined computers as well. Therefore, the specialists should closely examine the properties of the vector pipelining architectures.

3.2.3 Vector pipelining architecture

The ***vector pipelining*** computers have a special architecture, which is adapted to perform the vector operations. The features of this architecture are the following:

— a set of arithmetic pipelines to perform operations of addition, multiplication, division, floating point, and others;

— vector memory, which is analogous to the register memory, in which a cell serves to store a single vector (usually 64 words);

— sliced RAM; N blocks of this RAM allow the computer to access N adjacent cells of a single vector simultaneously;

— hardware devices to generate the address sequences of the vector elements (***address generator***);

— ability to connect the output of one arithmetic pipeline to the input of another one to perform a single group operation or a set of neighboring arithmetic instructions on the same data without writing the intermediate results in RAM (***pipeline concatenation***).

The typical ***group operation*** is an SAXPY (Single precision A multiplied by X Plus Y) instruction, which performs the action A * X + Y on the single precision floating point vectors. By its implementation, the multiplication and addition pipelines are concatenated.

Vector pipelining computers are subject to the pipelining principle. The principle is that the calculation of complex function is divided into a number of successive steps performed by the computing resources, called stages. Moreover, the data flow is throughput consistently these stages and due to the continuous flow, these data are processed in parallel, and adjacent data are processed simultaneously on adjacent levels. Through the pipeline stage specialization, and increasing the number of these stages, both the clock frequency is increased to the extremal values and the hardware costs are minimized.

Fig. 3.1 shows a timing diagram of a pipeline loading when it performs the vector operations for the vector of the length *N.* the diagram shows that the vector operations are associated with the time overhead. According to Fig. 3.1, the performance of a vector operation can be estimated by the formula:

$$T = L + kT_C N, (*),$$

where T_C is the clock interval, k is the average number of cycles for a single result, i.e. the period of computations, L is the pipeline startup, i.e., the time

required to fill the pipeline, including time for preparation of vector operands. In the simplest case, it can be estimated as $L \approx SkT_C$.

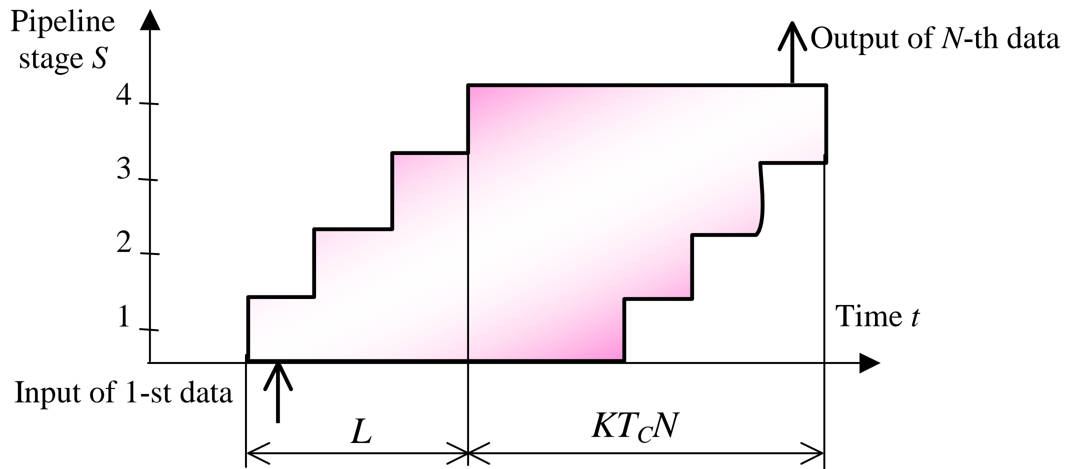

Fig. 3.1. Time diagram of the pipeline loading

Typically, the value L is larger for computers, in which the vectors are usually stored in RAM than for computers with the frequently used vector RAM.

The period kT_C for the result calculating is associated with the data input period. Many computers have $k = 1$ and this period coincides with the period of the clock. In the high-performance computers with several pipelines, such as a modern graphics accelerators, the number of pipelines M can reach from tens to thousands, and then, at the maximum load of them, this figure reaches $k = 1/M$.

From the formula (*), it follows that the average duration of obtaining a single result is equal to

$$T_R = kT_C + L/N.$$

The performance of the pipelined computer could characterize the number of results that it produces per unit of time:

$$R = \frac{1}{T_R} = \frac{N}{kT_C N + L}. \qquad (**)$$

If $L = 0$ or when $N \to \infty$, then R approaches to $R_\infty = 1/(kT_C)$. This value is called as an ***asymptotic performance***. It is the maximum performance with no overhead of the running pipeline. If $N = 1$, we obtain the minimum performance that is significantly less than the performance of the conventional scalar computer.

The value $N_{½}$ of the ***half performance length*** is of the interest. It equals to the length of the vector, for which the computer reaches a half of the asymptotic performance. Let $kT_C = 1$ ns, $L = 40$ ns. Then, according to (**), $R_\infty = 10^9$ Flops, and

$$\frac{10^9}{2} = \frac{N_{½}}{10^{-9}N_{½} + 40 \cdot 10^{-9}},$$

then solving this equation gives $N_{½} = 20$.

Assuming that $L \approx SkT_C$., i.e., the number of pipeline stages for this example is $S = 40$, one can reach the following conclusion. If the length of the vector is less than a half of the number of pipeline stages, then the pipeline computer performance is less than a half of the asymptotic performance. This finding should be taken into account when the vector data are processed in modern processors.

3.2.4 Problems

1) One program for the vector pipelined computer performs multiplication of 64-component vectors, and then adds the result to another vector. Another program does the same but using the group operation. Estimate the speed-up of the second program if the pipeline length is equal to 30.

2) Solve the problem 1, when the vector length is equal to 1000.

3) Solve the problem 1, when the vectors, which are stored in the outer RAM, the pipelined access to it lasts 250 clock cycles.

4) The pipeline length is 32. What is the vector length to achieve 80% of the asymptotic performance? How many times is it longer comparing to the half performance vector?

5) The graphic accelerator has the pipeline length of 400. What is the half performance vector length for it? How degrades the performance, when the vector length is equal to 64 comparing to the asymptotic performance?

3.3 Superscalar processors

3.3.1 Superscalar architecture

The von Neumann architecture was almost always the leading one by the computing expanding. The appearance of each new computer generation can be caused by the attempt to expand the bottleneck of this architecture. This bottleneck is formed by the communication link between the instruction and data memory, and the CPU core.

Thus, the cache RAM was built in the i486 microprocessor. This enables the CPU core to perform simultaneous reading 16 bytes of instructions, write and read 4 bytes of data to/from the cache RAM. This flow of information between the core and the cache RAM became five times greater than the flow between CPU and the main RAM.

A new step of the von Neumann architecture began with the advent of the superscalar processors. The main feature of the superscalar architecture is simultaneous fetching of multiple instructions from the cache RAM, their decoding, and distribution them to multiple instruction streams, that are executed in the separate blocks. The terms of such parallel instruction execution are:

1) the adjacent instructions should be independent on the data, addresses, and control;

2) free hardware resources should be assigned to the instructions for their implementation.

So, the superscalar processor can simultaneously run multiple instructions in several execution blocks. In other words, it implements the scalar parallelism at the microarchitecture level.

Another feature of the superscalar architecture is the separation of instruction cache RAM from the data cache RAM, providing the parallel access to both data and instructions. It also expands the von Neumann architecture bottleneck. Thus, the Pentium superscalar processor increases the throughput between the CPU core and cache RAM in three times compared to the i486 processor a the expense of such a separation.

The third feature of the superscalar processor is that it has sufficient operational resources and equipment to detect the neighboring instructions that can be performed simultaneously.

The superscalar architecture set except IA-32, IA-64, and their clones include the architecture families named Power PC, SuperSPARC, PA-RISK, IBM Power, MIPS, ARM and others. Now, these processors have up to six instruction streams.

Consider for example the SuperSPARC architecture. Specialists of the SUN Microsystems company determined the frequency of the instructions in the typical problems that are solved in the SPARC systems (see Table. 3.1). Analysis of the instruction statistics in Table 3.1 shows the feasibility of division the instruction stream into three parts: fixed-point arithmetic instructions, floating point instructions, RAM addressing instructions. And this division is implemented in the SuperSPARC processor. Thus a group of

four neighboring instructions is fetched from the cache RAM simultaneously. From this group, no more than three instructions are selected.

Table 3.1. The instruction frequency in various applications

Instruction type	Application with integer data	Application with floating point data
Fixed point arithmetic	50%	25%
Floating point arithmetic	0	30%
RAM reading	17%	25%
Writing in RAM	8%	15%
Conditional jump	25%	5%

These instructions meet the following conditions. There must be no more than:

— two instructions giving integer results;

— one instruction addressing the data RAM;

— one floating point instruction;

— one control and branch instruction that has to be the last in the group.

Measuring SuperSPARC CPU performance showed that, on average, it simultaneously executes from 1.4 to 2 instructions at the peak of 3 instructions.

The development of superscalar architecture towards increasing the number of instruction streams is associated with the increasing of the complexity of the logic circuits, which recognize the independent instructions in the instruction flow. On the other hand, the possibility of the scalar parallelism for any practical problems is limited. Therefore, it is inappropriate to increase the number of streams exceeding six to eight.

The superscalar processors parallelism is limited by:

— scalar parallelism level existing in the program. The average parallelism level in usual programs is from 2 to 3 instructions;

— hardware resources, now up to ten instruction streams can be realized;

— parallelism detection unit complexity.

To improve the superscalar processor performance can be fulfilled in the following ways:

— increasing the superscalar parallelism in the program by using the appropriate programming techniques and compiler to increase parallelism at the instruction level;

— to perform automatically the instruction overlap increasing during the instruction stream decoding;

— to specify the parallelism explicitly in the compiled program to facilitate the parallelism recognition in the hardware equipment;

— to perform two or more independent software processes to increase the number of independent instructions in the stream (multithreading);

— reducing the instruction pipeline stalls, for example, to foresee the conditional branches;

— reducing the pipeline depth, so the pipeline is filled more quickly, and the data dependence cycles are shorter (calculating the result D, writing D in memory, reading D for calculations).

3.3.2 VLIW-architecture

Another way of implementing parallelism at the instruction level is the processor architecture with a long instruction, i.e., VLIW-architecture (Very Large Instruction Word). The long VLIW-instruction contains multiple independent fields, which control the different operating units.

If we assume that the superscalar processor performs strictly the same number of instructions, even empty ones, in each cycle, we get the VLIW-processor architecture.

Compared to the superscalar processor, the instruction stream from the program RAM is increased in the VLIW-processor, but there is no need for recognition of the independent instructions. Both the programmer and the compiler are responsible for the VLIW-processor loading.

Some signal microprocessors can be attributed to VLIW-processor, for example, SHARC. Intel Itanium microprocessor architecture has the architecture EPIC (explicit parallel instruction computer), which is a VLIW-architecture with 16-byte instruction word length, three instructions per word.

3.4. Methods of superscalar processor performance increasing

3.4.1 Conditional branch prediction

When performing a conditional branch instruction, the next instruction is either the instruction that is already in the pipeline, or the instruction, which address is specified in the branch instruction. In the latter case, the instruction pipeline must be flushed.

The branch instruction execution has the unequal probability of its branches. If the instruction from the most probable branch is loaded into the pipeline the need for the pipeline flush occurs much less frequently, and the CPU speed is much higher. To realize this feature, the mechanisms of the conditional branch prediction are involved in the superscalar processors.

The most common branch, that requires foresight, is a branch instruction for the verification of a program cycle end. Thus the probability of a return to the beginning of the cycle is close to unity.

Such an instruction can be detected in compilation time. In the first RISC-processors, in which the prediction mechanism was involved (IBM Power), the compiler was setting a flag in the branch instruction, which assigns the probable branch direction.

In modern processors, the prediction is performed by the special equipment during the program execution in accordance with some strategy. To predict the branch address, the Branch Target Buffer (**BTB**) is used. This table operates as a cache RAM and stores the addresses of instructions to which the branches are done previously. For example, the processor Pentium-4 has BTB with a size of 4096 elements.

To predict the direction of the conditional branch, a mechanism is used, which is based on studying the behavior of branches in the program during its execution. This mechanism takes into account both a local behavior of branches (such as "usually jumps," "typically does not jumps") and a global law ("is changed with a particular law", etc.). The history of these branches is written in a special Branch History Table (**BHT**).

Current prediction algorithms implemented in hardware ensure the correct predictions by more than 90%. In the perfect prediction systems, the following prediction algorithms are used:

— combination of local and global mechanisms for predicting usual branch instructions considering the history of their behavior;

— static prediction for the instructions that are performed the first time, based on empirical relations. For example, "jump back" is assumed to be made, because the branch can start the cycle, and "jump forward" is assumed as undone;

— prediction of short cycles, which detects a jump at the end of a short cycle and calculates the number of iterations of the cycle, allowing correctly predict the moment of time of the cycle release;

— prediction of an indirect branch, i.e. a branch to the address in the memory cell, defining of target addresses for various instruction results even with the alternating program behavior;

— prediction of the target address of the Return instruction. It utilizes a special hardware stack for the return addresses, called as a Return Address Stack (RAS) for testing instructions Call — Return.

The branch prediction unit operates with the instruction decoder in parallel and independently from it. Through the effective implementation of the foresight of the branch address in the processors Pentium-III, Pentium-4 and K8 with proper prediction, they loose averagely only one clock cycle for the branch instruction execution. This means that the minimum time overhead, for example, to perform the iteration cycle or one jump in the chain of branches is equal to two cycles. In fact, the prediction in this chain of executed instructions runs in its own cycle, which consists of two stages: prediction and reading a new line of bytes from the instruction cache RAM and decoding the instructions from this line.

When an instruction enters the execution pipeline stage, it is clarified, whether this branch is truly predicted or not. At the time of this instruction completion under the wrong prediction, the execution of all following instructions is canceled and the reading instructions from the cache RAM for the true address is started.

This procedure is called a ***pipeline flushing***. The time (in cycles) on the instruction branch execution beginning with its reading from the cache RAM is called the ***unpredicted branch pipeline length***. This time is characterized by the time losses in the ideal conditions when the instruction goes through all the pipeline stages and is never delayed due to external causes (interrupts, cache misses, etc.). The unpredicted branch pipeline length takes the value of

11 cycles for Pentium-III to 30 cycles for Pentium-4E. In the real conditions, the losses of improperly predicted branches can be higher.

3.4.2 Trace cache and cycle unrolling

The instruction cache RAM of the processors like Pentium-4 differs from the Instruction cache RAM (I-cache) of the predecessor processors. A lot of instructions read from the external RAM in it are recoded to other instructions or sets of micro-operations (μops) before writing to the cache RAM. Therefore, not separate instructions but the sequences of instructions or μops are distinguished in this cache RAM. These sequences are called as traces, and the cache RAM is called as Trace cache (T-cache).

The ***trace*** is a sequence of instructions or μops that perform the same actions as is specified in the program portion. The traces are formed in accordance with the dynamic order decoded instruction flow. At the time of decoding, the initial branch prediction is performed. And if the expected branch is made, the target instruction is put in a trace directly under the branch instruction. The trace can contain a set of such "embedded" branches.

During the formation of the trace, the instructions from a short program cycle are written without instructions that control their execution. This process is called a ***loop unrolling***. If the unrolled instruction cycle does not fit into one trace, it is divided among several traces. Another way is when this cycle is rebuilt with the increased number of instructions in its core, which represent the adjacent loops, but with fewer iterations.

T-cache has several advantages:

— placing the cache RAM after the instruction decoder and storing in it a set of μops enables the processor designer to simplify the decoding stages and increases the clock frequency;

— dynamic replacement of CISC-instructions to the equivalent sequence of RISC ones or μops enables the CPU to use the positive properties of RISC-instructions, i.e., to perform one instruction per clock cycle, to increase the clock frequency;

— "building in" the predicted branches in the trace minimizes the losses on the branch implementation. Now, the instructions before the branch, after branch, and branch itself can be done in a single clock cycle. In the classical processor such an instruction sequence is performed at least for three cycles;

— unrolling a cycle into a trace reduces the time spent on the branch instructions and instructions, which are tracking the cycle end;

— the instructions addressing is simplified, the associative address is provided only to the first block of the μops, but not to every line of the conventional cache RAM;

— possibility of recognition and execution of the ***speculative calculations***. They are the preliminary calculations, which are not performed in the operating units. For example, by the speculative calculations, the situation of zero operand addition is identified and its execution is canceled, the final value of the cycle counter can be calculated, and the respective instructions are removed from the loop.

Thus, T-cache increases the parallelism of the program just before its instructions are executed. As a result, the speedup of the program execution is increased to the extremum level.

T-cache in Pentium-4 processor consists of 6 cell blocks. One cell contains a single μop or an RISC-type instruction. A single block is read from the T-cache on two cycles, i.e. three μops or instructions are executed per clock in this processor. The T-cache has the volume of 12K cells or 2048 blocks.

The principal feature of the T-cache is no direct relation between the instruction address and the appropriate μop place (μop or sequence of them) in the cache RAM due to the fact, that the variable length instruction of the IA-32 architecture is converted to one or more fixed-length μops. Furthermore, the μops are stored in the form of traces in the order of their execution. This violates the monotony of operations and continuity of the correspondence between instruction addresses and μop places in the cache RAM. Finally, the loop unrolling makes the traces of instruction sequences of several cycles fully violates the mutual correspondence between program instructions and μops. Then, a single instruction is mapped in several μops, which correspond to the different iterations of the loop or in μops belonging to several traces.

The instruction address mapping mechanism into the μop position is needed only for those instructions, to which a jump is done. They are the first μops in each trace. All other μops form a chain of blocks to the end of the trace.

Both I-cache T-cache have the similarities as they are the associative memories. But they have the fundamental difference. In the classical cache RAM each memory block is associatively addressed, while in the T- cache only the first block of the trace has the associative address. The rest of the trace cells have the incremental addresses from 0 to 255 (the 0-th cell follows the 255-th cell). All blocks in a trace are connected to a bidirectional list. When some block is removed from the trace, it is "reduced", i.e. the previous block is labeled as the last one.

If the instruction, to which a jump is fulfilled, is missing in the T-cache, i.e. no trace beginning with such an address is present in it, then the instructions are read directly from the cache RAM of the second level, is decoded and executed. Simultaneously a formation of a new trace is performed, and it is loaded in the T-cache. If there is a conditional branch instruction, then the

trace, starting from this instruction, is built according to the results of the branch prediction block.

The decoder processes the incoming instruction stream at a speed of not more than a single instruction per clock cycle, depending on the instruction format and presence of its prefixes. If an instruction can not be converted into a sequence of a small number of μops (up to four), it is replaced by a call of a "micro-subroutine" that generates a set of μops and sends them to execution.

There are some restrictions that do not allow to fill all six cells of a block by μops and need to form the next block. These requirements are the placement of μops which represent one instruction in one block, and limit the number of branch μops, they can not be more than two.

When a branch instruction occurs in the flow, the trace building is not interrupted and is continued under the direction of the predicted branch. If the branch is estimated as done, then after the jump μop the μops from the selected direction are placed to the trace. But the trace formation stops, when during the jump μop execution it becomes clear, that the branch is incorrect. And the trace is abrupted after this jump μop. Also, the trace forming is stopped, when the indirect branch, or subroutine call, or return instruction is found. The maximum length of the trace is equal to 64 blocks (384 μop cells).

It may happen, that correctly predicted branch is the cycle branch. In this case, there is an opportunity to unroll the cycle. Then several iterations of it are placed in the trace to improve the speed and sampling the instructions by reducing the overheads to the branches and tracking the cycle end.

By the implementation of the next iterations of the cycle, the μops are read from the T-cache till the end of the trace at the rate of one block per 2 clock cycles.

In the more superior models of the IA-64 architecture, the separate μop cache is used except the T-cache. But its purpose and behavior are the same, as ones of the T-cache.

3.4.3 Register renaming and reorder buffer

The ***register renaming*** is a method to remove the data dependencies like writing after reading and writing after writing, which occur in a sequential program between the register variables. This reduces the strict conditions of the data dependencies and increases the number of instructions that can be performed simultaneously in the superscalar processor.

With this renaming, the processor does not write a result in the specified register but writes it in a special buffer, which registers have the dynamically changing addresses. For example, instead of putting the stalls after the first instruction:

```
MUL  R1, R2, R3
ADD  R2, R4, R5
```

the processor renames the register R2 in the second instruction, for example, to R33, and the result is written to R33, but not in R2. This resolves a dependency "write after reading" between the two instructions, and they can be performed simultaneously. In the following instructions, the dependence on data through the register R2 is restored. These instructions will read the data from R33 instead of R2.

For the first time, this method was used in 1967 at IBM 360/91 to perform the floating point operations. This computer is also known that the pipelining and dynamic instruction fetching have been introduced in it. The dynamic fetching means that the fetched instruction is not executed, but is waiting in a queue to free resources and scheduling, so to speak, it is placed on the shelf. Therefore, this method is called as an ***instruction shelving***. The

mentioned queue reorders the instruction flow, therefore, it is named as a ***reorder buffer***. Because of the complexity of its hardware implementation, this method was not widely used until the 90-s.

The first superscalar processors have not activated this method. It was implemented step-by-step. First, partial renaming was introduced in IBM Power, PowerPC. Then the renaming was used in all superscalar processors. The ***partial renaming*** applies some subset of instructions, such as floating point and the full renaming considers all the instructions.

Depending on the microarchitecture, the buffer registers are:

— part of the register memory (attached memory);

— separate register memory block;

— reorder buffer like circular buffer;

— part of the instruction shelving buffer.

Each buffer register can be 1) free, 2) ***architectural register***, i.e., its address is the address in the architecture, 3) renamed register without a datum, 4) renamed register with a valid result.

Initially, the first *n* registers are the architecture registers. Then, when it becomes necessary, an additional register is selected from the available registers and it is the renamed register without a datum. After writing a result in it, it becomes a renamed register with a result. When the respective architectural register becomes free, the renamed register is assigned as the architectural one with the same address. Finally, when the architectural register is not used for a long time, it goes into a free register.

The reorder buffer and the register renaming are used in the Intel processors since Pentium–IIp. In this architecture, each executed instruction corresponds to a separate pointer to the register in a buffer. This buffer can also serve as a part of the Deferred scheduling register Renaming Instruction Shelf (DRIS).

The number of registers in the buffer is a trade-off between the hardware cost and the program acceleration. In most architectures, the number of registers is in 1.5 – 8 times higher than the number of architecture registers.

The logical network, which defines the free registers, is of great complexity. Typically, it analyses a table, which maps the registers in the instruction flow and determines, which of registers are not used for a long time.

3.4.4 Problems

1) The processor has the 7-staged instruction pipeline with an instruction decoder in the second stage. Estimate the speed-up factor of the processor due to the prediction block operation if it has the probability of true predictions of 90% of branches. Consider that the program contains 25% of the branch instructions, the program is placed in cache RAM.

2) The conditions are the same as in the problem 1, but 50% of branch instructions are conditional long jumps to the instructions placed in RAM. Consider that a single cache miss lasts 100 clock cycles.

3) The conditions are the same as in problem 1, but the unpredicted branch pipeline length is equal to 30 cycles.

4) Consider a program loop, which contains 10 instructions, 3 of them provide the loop organization including a jump instruction, all the data are stored in the registers. Calculate the speed-up of the program execution if the loop is unrolled with a factor of 4. Note, that the unpredicted branch pipeline length is equal to 11 cycles, but the branch prediction is not implemented, the iteration number is a multiple of the unrolling factor.

5) The conditions are the same as in the problem 4, but the branch prediction is implemented.

6) The conditions are the same as in problem 4, but the unrolling factor is equal to maximum, and the loop before and after unrolling is placed in a

single trace of the T-cache. Note, that each instruction is of the RISC type and has the length of 4 bytes, and the maximum length of the trace is 1020 bytes.

7) Why are the reorder buffers much rarely used in the modern processors than earlier?

3.5. Multithreading

3.5.1 Threads and multithreading

A ***thread*** is a major program unit that reflects the medium- or coarse-grained parallelism. It is usually relatively independent computing process, subroutine module of medium or high complexity. For a set of threads, that belong to a single program, the operational system assigns the same memory and processor resources. Within a single-processor architecture the following features of the parallel execution of threads:

— sequential execution of threads in the time slots;

— event controlled sequential execution of threads;

— parallel execution of threads.

The sequential execution of threads in the time slots is named as a ***Time-Slice Multithreading***. The processor in this mode is switched between the threads at fixed time intervals. This mode uses the processor resources inefficiently, especially if several threads of different importance are pending;

The event controlled by the sequential execution of threads is named as a ***Switch-on-Event Multithreading***. Such an event can be a synchronization moment of related processes. Often it is a task switching during long pauses in computations, for example, during the cache miss or file access. In this case, a process that awaits the loading data from the slow memory is suspended, freeing the CPU resources for other processes.

Due to the multithreading, the delay of the access to the slow memory or other processors may be hidden from the program user. Therefore, this effect is named as a ***latent time hiding***.

The disadvantage of the sequential multithreading consists in the frequent exchange of the thread context, which requires a lot of machine cycles (see interrupt handling). It also makes it difficult to exploit the parallelism of multiple operating units.

Among parallel execution flow methods, the ***Simultaneous Multithreading*** (SMT) is commonly used. In this case, two or more software threads run on a single processor at a time, that is, without switching between them. Then, the CPU resources are allocated dynamically according to the principle: "Do not use — give to another." This helps to utilize the CPU resources more perfectly.

The architecture IBM Power5 also applies the SMT mode. The full implementation of SMT is quite complicated. Thus, the logic networks of SMT implementation occupy 24% of the Power5 processor core. According to the IBM company, the use of SMT improves the server performance by about 35%.

3.5.2 Hyper-Threading

A simplified approach of SMT is the basis of the Intel ***Hyper-Threading***. technology. This technology utilizes only 5% of the crystal space for the Hyper-Threading implementation.

For the first time, the technology Hyper-Threading was implemented in the processor Intel Xeon with the Pentium-4 architecture. In one physical processor, two Logical Processors (LPs) are formed that share the CPU computing resources. The operation system and applications "see" exactly two logical CPUs, and can distribute the works between them.

One of the purposes of the Hyper-Threading is to allow a single thread to run in the virtual CPU as well as in the real CPU. For this, the processor has two basic operation modes: Single-Task (ST) and Multi-Task (MT) modes. In the ST mode, only one LP is active and uses all available resources (ST0 and ST1 modes). Another LP is stopped by the instruction HALT. When the second thread is running, then another LP is activated and CPU is put on the MT mode. The stop instruction HALT is given by the operating system.

The state or the context of each of the two LPs, called, the Architecture State (AS) is stored in a separate block. It includes the states of the registers of different types. They are general purpose, control, and interrupt service registers.

Each LP has its own Advanced Programmable Interrupt Controller (**APIC**) and a set of registers, which are supported by the Register Alias Table (**RAT**). RAT tracks the correspondence between eight general purpose registers of the IA-32 architecture and 128 physical registers of CPU, providing the register renaming.

When two threads are carrying out, two sets of counters, named Next Instruction Pointers are running. Two active LPs gain access to a shared T-cache alternately, that is, during each even cycle. While only one LP is active, it gets the exclusive access to the T-cache. Similarly, the access to the microinstruction ROM is performed. Also, the blocks that support the virtual addressing, i.e., the Instruction Translation Lookaside Buffers (ITLBs) are doubled. The instruction decoding block is shared between both software threads as well.

Five schedulers, which assign the operations to resources, process the decoded instructions regardless of their belonging to a processor LP0 or LP1 and send the instructions to the respective processing units, depending on the readiness of instructions and availability of resources.

Cache RAMs of all levels are fully shared between LPs. As a result, the most applications which have the acceleration in the multiprocessors, can be effectively computed in the Hyper-Threading mode. But there are problems. for example, if a process is situated a waiting loop, it can take all the physical resources of the CPU without letting the second LP to work. Thus, the performance can fall (to 20%), when using the Hyper-Threading mode. To minimize this effect, the Intel company recommends using the instruction PAUSE instead of empty waiting cycles.

The multi-threaded processing architectures are improved towards increasing the number of virtual processors, improved instruction dynamic fetching, increase of the operating unit workload. Thus, the Sun UltraSPARC T1 processor, which appeared in late 2006, has eight CPU cores, each of them can process up to four software threads. Therefore, the total number of parallel threads is equal to 32. The number of pipeline stages is reduced to six in it. This has simplified the branch prediction and scheduling networks and increased the processing unit loading. In 2013, the number of cores in the next architecture UltraSPARC T5 is increased to 16 and the total number of simultaneous program threads is achieved 128.

3.5.3 Problems

1) Name the conditions of the executed tasks when the time-slice multithreading is better than the switch-on-event multithreading.

2) Why is the multithreading better fitted for the coarse-grained parallelism implementation?

3) Why does the simultaneous multithreading optimize the resource balancing of CPU?

4) The Intel Xeon architecture has two thread hyper-threading, and the UltraSPARC has four thread multithreading. Explain the conditions, which determine the number of parallel threads.

5) The number of pipeline stages is usually decreased in the processors by the introduction of the multithreading in them. Explain why.

3.6. Memory access parallelism

3.6.1 Memory access consistency

Most algorithmic languages have a simple semantics of the sequential memory access. This allows the programmer to assume that all memory operations are performed in a strict order, which is defined by the program. In fact, in the today's computing system several accesses can be done to a memory at the same time, such as from the adjacent program instructions, from the DMA unit, from different program threads running in parallel. In addition, the order of the accesses, which is given by a programmer, can be exchanged both by the program compilation and by its implementation in the modern microprocessor. Therefore, it is important to consider the issues, which are related to the memory access consistency.

To analyze the memory accesses, the multiprocessor model is convenient to study. In this model, multiple Processing Units (PUs) have the parallel and independent access to the shared memory. A similar situation occurs in a single PU, which operates in a multi-threading mode.

A programmer during the programming process usually implies a model of the strictly sequential memory access or a sequential consistency model. A computer system is ***sequentially consistent*** if the result of any calculation in this system is the same as the calculation, which is performed by

a single processor, i.e., the calculation order is given by the program instruction sequence.

There are two requirements to the sequential consistency:

— program instructions have to be executed in the same order as they do in a single processor;

— the sequential order of instructions must satisfy the unchanged order of the memory accesses.

In the consistent system, the memory is connected to PUs through a switch, which connects the memory to PUs in an order, which depends only on the compiled program. Consider the following example (Dekker's algorithm):

```
PU1: F1 = 0                        PU2: F2 = 0
     If F2 = 0 then                     If F1 = 0 then
     F1 = 1                             F2 = 1
     {Critical procedure}                  {Critical procedure}
```

When PU1 is trying to start a critical procedure, it checks the flag F2 and sets its flag F1 to prevent PU2 to perform a critical procedure. The same does PU2. If the system is consistently agreed, only PU1 is the first to start a critical procedure, regardless of different circumstances. Further different consistency circumstances are considered in this example.

3.6.2 Sequential consistency violations

Consider some typical examples of architecture improvements that lead to the sequential consistency violations. The complexity of the consistent memory accesses depends on whether the cache RAM exists, or not.

First, consider an architecture without cache RAM, but with a writing buffer. All modern processors have a writing to an intermediate buffer. The actual instruction puts a result in it, which allows completing the instruction

implementation before the RAM writing transaction is completed. The system structure with two buffered PUs is shown in Fig. 3.2.

The processor may put the writing operation flag in the buffer and continue its work, not to wait for the end of the writing operation. The reading this flag is performed without buffering. The presence of buffers can disrupt the consistency. At a time, when the flag F1 = 1 writing is passed through the buffer, which is delayed by the time t_3, PU2 can read the zero flag F1 and start the critical procedure not in time.

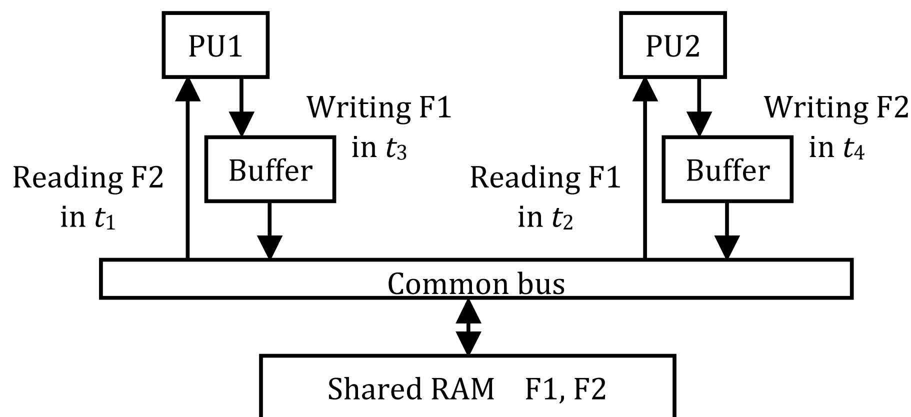

Fig. 3.2. Violation of the memory access consistency due to the write buffers

Next, consider the architecture, which has the write-reading operations of different time. The system structure in Fig. 3.3 has several memory units with different access delays.

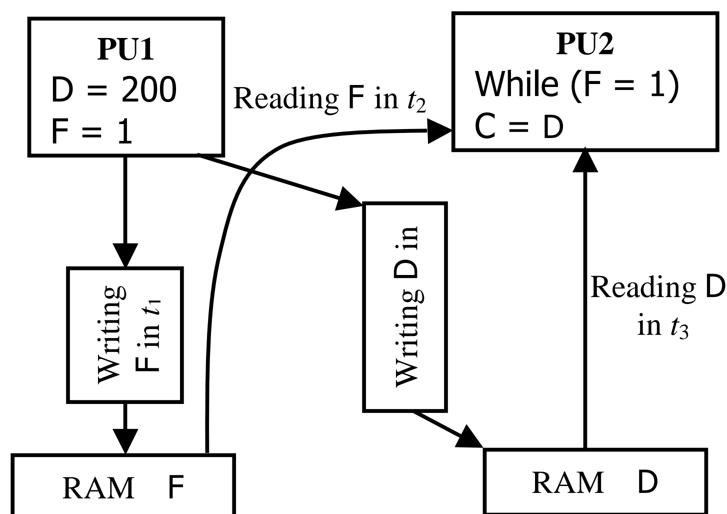

Fig. 3.3. Violation of the sequential consistency to RAMs with different delays

The similar situation also happens if different instructions are executed in the pipeline mode with different execution time and, accordingly, different times of writing in the buffer memory. The first PU writes a new value D and sets the flag F in order to signal a second PU to read the new value. But because of various delays of accesses to both RAMs, the second PU reads actually the old value D.

3.6.3 Non-blocking read operations

In the consistent processor, the read operation, that follows the writing operation, is blocked until the read cell receives a new value. For the dynamic scheduling simplification, this blocking is not used. This can also lead to the consistency violation.

The examples given above showed that the processor must ensure that its previous memory access operation was finished before the next operation begins in accordance with the order of instructions in the program. This requirement is called the instruction order requirement.

3.6.4 Sequential consistency in the architecture with a cache RAM

Memory caching or data duplication in the main RAM and cache RAM, which are used by several processes, is leading to situations, which are similar to those, described above. Therefore, the cache RAM systems must fulfill the same requirements of sequential consistency. The following situations are added to the situations shown above, which are related to the data duplication.

Cache RAM coherence protocol is a discipline of replacing the old values in the processor memory for a new one in all its copies. A new value is usually distributed by the method of invalidating or of updating of all its copies.

The cache RAM coherence is performed under the conditions:

— write operation should be recognized by all Pus, that are able to write;

— write operations in the selected memory cell must be perceived by all PUs in the same order.

However, these conditions are not enough to guarantee the memory access consistency in all conditions. To implement the cache coherency, the following methods are used.

Detection of the writing end. In the case, when the cache RAM is absent, the write acknowledge signal can be generated, when the written datum gets the storage. But this time is too early for a system with a cache memory.

Let each PU in Figure 3.3 has its own cache RAM. Let the operand D is stored in the cache RAM of PU2. Now, PU1 writes F after writing D in RAM, but before the moment when D comes to the cache RAM of PU2. Then, it is possible that PU2 reads a new value of F and still is reading the old value from its cache RAM, which makes the consistency violation. Therefore, PU2 should wait until D is rewritten in the cache RAM, or it is discharged before the writing to F.

That is why, the writing operation into a cell, which has a data copy in the cache RAM, requires either discharging, or overwriting confirmation signal. Moreover, all these signals must be collected together either in the shared RAM, or in PU, which initiated this writing. Only after the confirmation signal, this PU is allowed to assume, that the writing is completed.

Write operation indivisibility. The distribution of data exchanges to multiple cache RAMs is the operation, that takes time and can be divided into phases (not to be atomic). Therefore, the operation indivisibility condition should be obeyed.

First, the write operation in a shared cell should be completed sequentially, and for all PUs the sequence of writings should be accepted in equal order.

Secondly, it is forbidden to read a new value, until a signal is received that the old value is canceled in all cache RAM s or is overwritten there.

There are two general strategies for dealing with writes to a cache:

— ***write-through***, when all data, which are written to the cache by PU, are also written to RAM at the same time;

— ***write-back***, when data are written to a cache, and a ***dirty bit*** is set in the affected cache line. The modified line is written to RAM only when this line is replaced, otherwise, if this bit is reset then the line is discharged.

Write-through caches are simpler, and they automatically deal with the cache coherence problem, but they increase bus traffic significantly. Write-back caches are more common where higher performance is desired.

3.6.5 Cache RAM coherence optimizations

To increase the cache RAM speed, the coherence protocol is usually optimized. First, the independent program threads are able to make parallel and overlapped writings to cache RAM. In this situation, the written values are delayed in the writing buffer to the desired number of cycles, and the invalidating signal is sent beforehand to the cache RAMs.

Second, the read operations are delayed in accordance with the program requirements. This delay allows reading the data once again if a write confirmation signal occurs.

3.6.6 Consistency requirement relaxation

As we see, the consistency requirement satisfaction, on one hand, requires extensive additional hardware costs of the parallel computing

system. On another hand, it leads to the computation slow-down even to the point where a complex computer system has a speed, which is lower than the speed of a conventional processor.

Therefore, generally, the relaxed consistency models are often used. In these models, an instruction set is divided into those that must satisfy the consistency requirements and those that may violate these requirements.

The former instructions include specialized instructions which force the hardware to follow the consistency rules. There is also a category of read-modify-write instructions, which are used for writing and reading the synchronizing flags or semaphores, such ones in Fig. 3.2. They also include special synchronizing instructions. These are instructions such as a barrier instruction, which establishes a barrier, i.e., a full stop between the number of writes and subsequent readings of the shared memory.

The weakened consistency is used for the rest of the instructions. This weakening can be on several levels depending on the computer architecture and on the weakening set. The slightest weakening is to permit reading a shared cell after writing to it. One must be conscious that, in fact, writing can really occur after reading, and this case is impossible when programming the appropriate synchronization instruction.

The further weakening is the ability to write after writing and to write or read after reading. Finally, the most relaxed weakening is permitted any violation of the memory access consistency for all instructions except the synchronization instructions.

3.6.7 Compiler effect

The compilers that rearrange the instructions, which access to the shared memory, may also violate the consistency, as well, as the hardware does it. Therefore, if the compiler makes the optimization, it must check

whether it provides the memory access consistency. As a result, the compilers typically perform the automatic optimization with a caution, and the effectiveness of such an optimization is very far from desirable.

On the other hand, the programmer should be aware, that because of the compiler influence, his program could lose the memory access consistency in the unpredictable circumstances.

3.6.8 Problems

1) Find the program solution, which prevents the consistency violation, which illustrated by Fig. 3.2.

2) CPU executes a subprogram of a large data array moving. Calculate the speed-up of this subprogram execution if the cache RAM is added, which operates in the write-back mode. Note, that the subprogram is a loop, which is executed for 10 clock cycles for moving a single 4-byte word if the data are in cache RAM and the cache hit occurs, a single access to the outer RAM adds a pipeline stall, which lasts 10 clock cycles, the cache line occupies 128 bytes.

3) The conditions are the same as in the problem 2, but cache RAM operates in the write-through mode.

4) Why is not the strict consistency model used in the real computer architectures?

5) How does the barrier synchronization work?

6) How can the compiler violate the memory access consistency?

4. MULTITASKING AND DATA PROTECTION

4.1. Virtual memory

The principle of the virtual memory considers that the user, preparing its program, does not deal with the physical RAM but with a ***virtual memory***. This memory is flat, and its capacity is equal to all the CPU address space, that is defined by the address bit width of the instruction fields and basic registers. For example, the architecture IA-32, which ancestor is the i80386 processor, can manage the virtual memory of up to 64 TB (terabytes). This is the ***potential virtual memory*** volume. The ***real virtual memory*** volume is significantly less than the potential one. It is defined by the RAM volume and by the part of the hard drive, that stands out running the virtual memory implementation.

The user has in his possession the entire address space of the computer regardless of its physical RAM volume and memory spaces for other applications in the multiprogram mode. This ensures the flexible dynamic allocation of memory and the significant convenience for the programmer. In the modern computer, all this is achieved without reducing its speed at the cost of the complications of hardware, and operating system.

At all stages of the program preparing, including its loading in RAM, the program is presented in ***virtual addresses***. Only at the very execution of the machine instruction, the virtual addresses are translated into real addresses of the computer, named as the ***physical*** or ***execution addresses***.

The conversion of the virtual addresses into physical ones is simplified if the physical and virtual memories are divided into identical blocks. These blocks are called ***pages***. The virtual and physical memory pages are provided by numbers, called the physical and virtual page numbers, respectively.

Each physical page can store one of the several virtual pages. The byte numbering bytes both in virtual and physical pages is the same. The program during its loading can be placed to any available physical pages, regardless of whether they are consecutive or not. Memory paging enables more efficient carrying out the information exchange between the external memory and RAM, as the program page should not be loaded until it is needed.

Initially, the initial program page is loaded and gets the control. If in the runtime the another page has to be loaded, then the automatic operation system call is done, that organizes the load of this page. This page before its loading usually forces a process, when the unneeded program page frees the space by moving itself to the external memory. Such a page replacement is called as a ***page swapping***. The respective part of HDD, where the virtual memory is allocated, and where the swapped pages are stored is named as a ***swap file***. The size of this file depends on the hard disk volume and the operating system type and can be defined by the computer system administrator.

The correspondence between the virtual and physical memory locations is set the page table. It considers, that the physical pages can be stored both in RAM and in external memory (HDD). A simplified paged memory scheme is shown in Fig. 4.1.

The table pages for each program are formed by the operating system during the memory distribution and it is processed every time when the changes in the memory allocation are made. The procedure for the memory addressing is the following. The number a virtual page is selected in the address field and is used to access a cell in the page table, that indicates the number of the physical page. This number is concatenated with the lower bits of the virtual address, i.e. with the position of a byte in the page, and forms a physical address for the access to RAM.

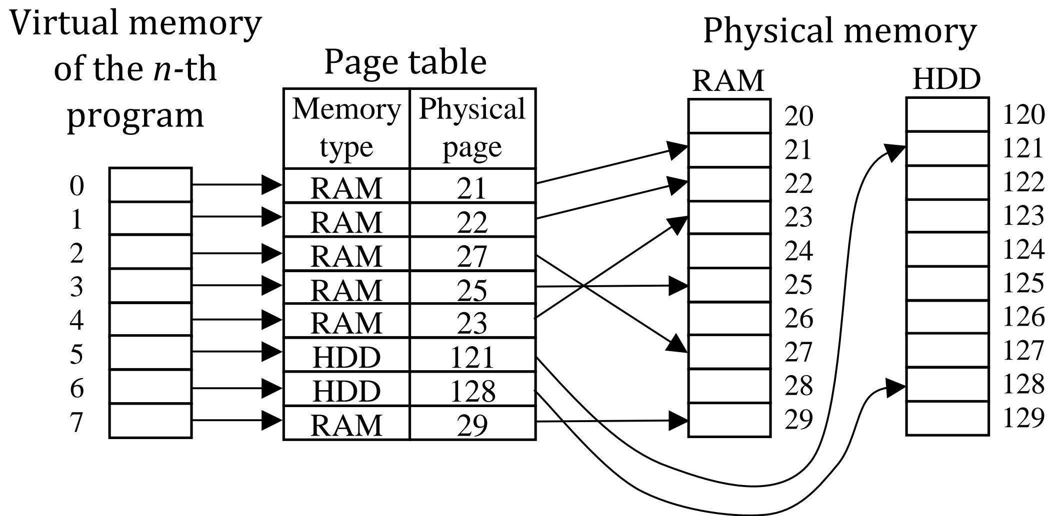

Fig. 4.1. Mapping the logical memory to physical pages

The formation of the physical address is drawn in Fig. 4.2. Here, the each physical page number refers to its base address.

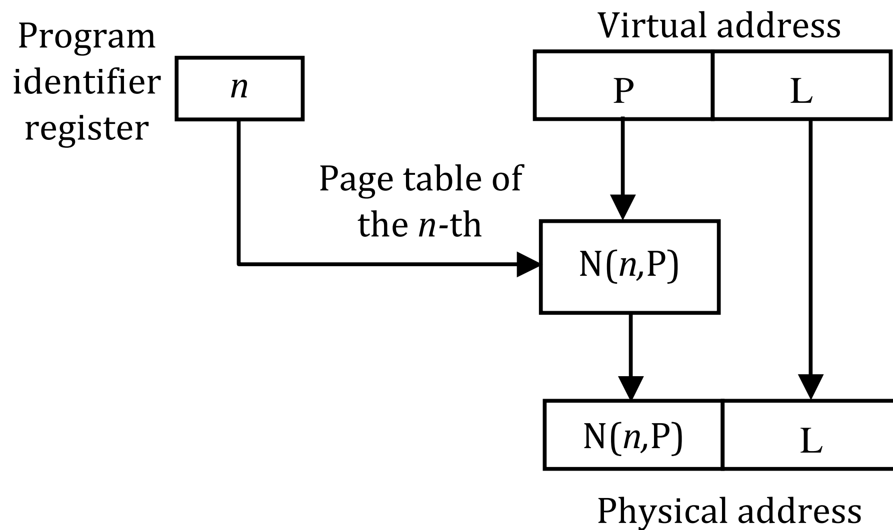

Fig. 4.2. Deriving the physical address

For each of the programs, running in the multiprogram mode, its own virtual memory is organized and a separate page table is created. Besides, all the programs share a single physical memory (RAM and external memory).

The page tables are stored in RAM as well. The reference to a line of the active page table is performed at the address, which is determined by the identifier of the active program and by the virtual page number.

If the page table indicates the placement of information in the outer memory, then the reference to it can not happen immediately, because the operating system must arrange the loading the respective page to RAM.

The mechanism of the page addressing in the real computer systems is much more difficult. Thus, to refer to the page table of the relevant program, the operating system should apply firstly to the directory, where the base addresses of all the page tables are stored. Besides, the modified pages from RAM are moved to HDD not directly but through the disk cache, because of the high probability to access these pages.

To speed up the address conversion, a small cache RAM is used, which stores the frequently used rows of the active page table. This cache RAM is called as a ***Translation Look-aside Buffer*** (TLB). In addition, the operational system forms in RAM a common table, that stores information about the virtual pages and respective physical pages for more recently used pages (several dozens), including ones for various programs.

The length of the program modules is usually random. So, it is convenient that each module has its own byte numbering starting from zero. It is desirable that the compiled program could work with dynamic memory allocation, without requiring the complex efforts to link its different parts in a common piece of code. In modern computer systems, this problem is solved by the use of the memory segmentation.

The virtual memory of each program is divided into parts named as ***segments***, which have independent byte addressing. Such a virtual address has an additional field of the segment number. There is a hierarchy of the program addressing, which consists of four steps: program identifier,

segment, page, byte. This hierarchy is represented by the hierarchy of tables for the conversion of virtual addresses to physical ones. The program identifier table indicates the start address of the respective segment table. The segment table contains the information about the segments of this program indicating the start addresses of the page tables of the corresponding segments. The page table determines the position of each page in the respective memory segment.

The physical segment pages may not be in a sequence, the pages of a single segment can be located partially in RAM, and partially in HDD. The paged memory organization, memory segmentation and combinations thereof were introduced in the third-generation computers such as Burroughs B5000. In the personal computers, these memory organizations were used beginning at Intel 8086 architecture. In the architecture i80286, the memory segmentation has got the protected mode.

However, the protected mode (multitasking, virtual memory) computers became the most widely used with the emergence of 32-bit processors, such as the IA-32 architecture. This architecture has the hardwired memory management unit, which supports a mechanism of the protected paged memory segmentation. This support has allowed the system software developers to build the logical address space in accordance with the computer purposes.

The processors with the IA-32 architecture can operate in both real and protected modes, and support the following options for the logical memory organizations:

— linear logical address space, i.e., the memory is an array with continuous numbering;

— segmented logical address space, when there are several segments, each of which has a variable number of bytes;

— paged logical address space, which means a large number of pages of fixed length;

— segment-paged address space, which contains a number of segments, which in turn consist of a natural number of pages.

The architecture IA-32 supports up to 16 000 segments of the different volume. The size of each segment can be up to 4 GB, that allows the programmer to manage the virtual memory capacity of up to 64 terabytes (in the multitask mode up to 16 000 segments can be allocated for each new task).

A programmer can use a protected mode and divide the virtual memory into ***segments***. Thus each program code module can be provided by a logical segment of memory. The programmer can divide the logical address space to, for example, segments of data, of software codes, of a stack and some additional segments, employing the segment addressing mode.

The virtual memory is also divided into pages. The most convenient for the program modules page size is equal to 4 KB. This size is well suited for the functioning of operating systems, IO subsystems provides frequent hits by the TLB access. The paged memory organization provides the swapping algorithms with more rational form. In the Pentium processors, the page size can achieve the volume of 4 MB.

4.2. Computer multitasking

The multitask operation was put into computers in the 60-s to reduce the downtime of costly operational resources and increase the operating system efficiency. Now the multitask mode is used in all universal computers, as it is a prerequisite for the functioning of all modern computer systems. The introduction of the multitask mode is associated with a number of complications of the CPU architecture.

To protect the data or programs from the unauthorized access or accidental erroneous, and for the virtual addressing, the processors used the two- or three-level virtual address translation. To speed up the address mapping, TLB is typically used. For existing applications, the virtual address space, which is limited to 32-bit addresses is enough. A gradual transition to 64 - bit virtual addresses takes place now.

Another aspect of the multitasking concept is the ***context switching***. All modern operating systems, when a task is replaced by another one, provide the correct displacement of the first task of the second one in the common computer resources. During this process, one context is replaced by another context.

Here, the context is a state of internal CPU registers and other cells to be replaced to ensure the correct execution suspend of a program and start the execution of another. Each task, process or program thread has its own ***Task State Segment*** (TSS). During the context switching, the state of a task that is displaced, is rewritten in its TSS. The context from TSS of the new task is downloaded to the CPU registers. In that time, the state of the instruction address register is restored, and the new task resumes its execution from the place of its interruption.

The number of internal CPU registers in the CISC-processor is signify-cantly less than this number is in the RISC-processor. Therefore, the RISC-processor context switching is slower because of the need to rewrite all its numerous registers to TSS. But because of the complexity of the superscalar CISC-architecture, the context switching also requires a lot of time. Thus, the context switching in the IA-32 architecture consists of automatical rewritten at least 104 bytes of information from different registers to TSS.

The reason for changing the context in computers is calling a task by another one or the interruption. During the interruption, the task should store

its context in TSS in a way that provides for resuming exactly its execution. At the time of the interruption, an instruction is executed at various stages of a pipeline. To not save the state of all registers of the instruction pipeline, the CPU enables to complete the actual instruction execution and then to start the context switching.

Just after the context switching, the new task begins its operation with the RAM access and the cache miss. Therefore, the information relating to the old task that is stored in cache RAM, TLB, BTB, is partially or completely replaced. As a result, the context switching is associated with the high temporal overheads.

Note, that in the modern computer, the number of simultaneous tasks (threads) achieves up to several thousand, and all of them afford the frequent context switching. As a result, the actual computer performance is determined mainly by the context switching speed. To increase this speed, the context switching is accelerated by different ways:

— hardwired context overwriting;

— storing TSS in the cache RAM;

— increasing the volume of the cache RAM at different levels;

— storing tasks, that are often performed, in RAM instead of in HDD;

— preserving TLB, BTB flushing for the frequently used threads;

— the flash memory or DRAM addition for storing the swap file and system files.

4.3 Memory protection

4.3.1 Introduction

If a set of independent programs can be placed simultaneously in the memory then special measures are required to prevent or to limit the access of one program to the space belonging to another program. A similar problem arises when the user program tries to access the space of the operation system. The causes of the unauthorized reference to other memory areas can be a hardware failure, errors in the user program, especially during its debugging. The consequences of such failures are especially dangerous if the operating system software is disturbed, as after this a computer system may behave unpredictably.

In order to stop the destruction of some program by another one, it is enough to protect the program memory from attempts to write it (write protection). But it can be allowed for the programs to read data in this memory area.

In other cases, such as restricted access to the information, it is necessary to prohibit both writing and reading the memory for other programs. This writing and reading protection helps to debug the programs as well. By this process, the control of going beyond the allowed memory limits is performed.

Different options for the memory protection can be selected:

• the attitude to the memory of another program is set, which determines whether the protection applies only to the write operation or to writing and reading.

• the following relationships to the memory of its own program is set:

— access for writing and reading is permitted;

— only reading is permitted;

— access is permitted but to the address taken from the program counter;

— access is permitted to the address taken from any register, except the program counter.

If the memory protection is violated, the program stops its execution and the memory protection violation interrupt occurs.

The protection of the program tamper in the wrong memory area can be arranged in different ways. This protection implementation should not significantly reduce the CPU speed and require too much hardware costs. Consider the most common ways to implement the memory protection.

4.3.2 Protection of individual memory cells

In the small computing devices, such as part of the control system, the debugging of a new application in parallel with the operation of the general program is of demand. This can be achieved by the allocation in each memory cell of a special protection bit. The setting of the protection bit forbids the writing in this cell. This is, so-called, a method of the control bit.

In the multiprogram systems with the memory segmentation, not individual cells but memory blocks are protected. There is often the ability to specify the different access modes for different programs to the specific memory areas or segments.

4.3.3 Method of boundary registers

The idea of the method is that two boundary registers are attached that define the upper and lower bounds of the memory where the program has the right of access. Each memory reference is verified whether the address is in the boundaries. The access address is compared with the addresses in the boundary registers. When the address is mismatched, the

memory access is canceled and the respective interrupt request is formed, which transfers the control to the operating system.

The content of the boundary registers is set by the operating system before the working program starts its execution. If the base register is used for the dynamic memory allocation then simultaneously determines the lower memory bound. The upper memory bound is calculated by the operating system according to the length of the program to be loaded in RAM.

Thus, this method allows predicting the output of the memory access from the allocated memory area. This is convenient for executing small programs in linear addressing mode within a single memory segment. Since it is necessary to add two registers and two comparators to the processor structure for each protected memory area, this area is usually only one.

4.3.4 Method of protection keys

This method is more flexible. It allows the application to access the memory areas that are not consecutive. The processor memory is divided into blocks in a logical relation. Each memory block is assigned a code, which is a memory protection key. Thus each application in the multiprogram architecture is given a program key.

The access to a given memory block for reading and writing is allowed if the keys of the block and program are matched. The exclusion is when the key has zero code, i.e., it is privileged. The protection key method does not preclude the use of a control bit method to protect the individual memory cells and the boundary register method to protect a certain memory area.

All these methods are used mutually in the modern computers. The separate block, which fulfills the memory protection methods, is called as a ***Memory Management Unit*** (**MMU**).

4.4. Memory management in the IA-32 architecture

4.4.1 Memory protection methods

Let us briefly consider the memory protection methods in the computer systems based on the IA-32 architecture. The memory blocks under protection are segments and pages. The protection system is based on the following principles:

— checking the access attributes (protection keys) before the memory access;

— ring structure of the hierarchical privilege levels (priorities), so as to prevent the damage to the most important programs, which are placed in the core, by the errors of the programs allocated in the external rings (see Fig. 4.3).

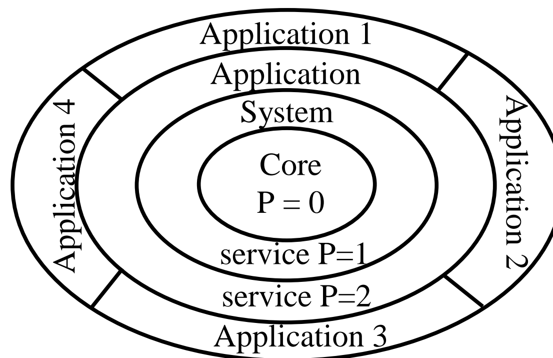

Fig. 4.3. Priority ring structure of the memory protection

The mechanism with a set of privilege levels provides the reliable memory protection. The privilege level is set both in the program descriptor and in the descriptor of the active software or data segment. The memory access is allowed if these levels are agreed.

The zero level is the most privileged one. It is given to the segments or pages, which have the most limited access. The third level is the lowest one, and it is given to the most available memory segments or pages. Applications

can access data segments or programs only with the same level of privileges or less restricted (with higher numbers of privileges).

The multitask mode is organized by four levels of protection and attribute verification system at the cost of separating the applications from each other and from the operating system. The protection mechanism involves the program interruption in the event of a system failure.

The hardware implementation of the hierarchical memory protection and of the access attribute verification allows the system to prevent the vast majority of accidents (up to 95% of failures of the incorrect memory access in the multitasking systems).

4.4.2 Memory protection implementation

IA-32 architecture has two addressing modes for the compatibility to the i8086 architecture:

— ***real address mode***, which is i8086 compatible;

— ***protected mode*** of the virtual addresses.

The latter mode provides the physical addressing of the memory with the volume up to 4 GB and access to the virtual memory up to 2^{46} = 64 terabytes. The rapid execution of a program, which is developed for the i8086 microprocessor with the memory protection and multitasking, is performed in an i8086 virtual processor mode, which is a variant of the protected mode.

4.4.3 Memory segmentation

When the processor in the protected mode, the segment, data and stack parameters are defined by the relevant segment descriptors. A set of segment descriptors is stored in the memory as the respective tables: Global Descriptor Table (**GDT**), local descriptor table (**LDT**) and interrupt descriptor table (**IDT**). The location of these tables in the memory is determined by the

relevant registers GDTR, LDTR and IDTR. The access to these registers is performed by the privileged instructions LGDT, LLDT, LIDT, SGDT, SLDT and SIDT.

The program accesses the descriptor through the 16-bit selector. It is loaded into the appropriate segment register: Code Selector (CS), Stack selector (SS) and Data Selectors (DS, ES, FS, GS). The loading of the appropriate selector is executed by the respective instructions LDS, LES, LFS, LGS and LSS.

Each selector has three fields: Request Privilege Level (RPL) to access the memory, Table Indicator (TI), which selects either global GDT or local LDT descriptor table, index INDEX to select one of 8192 descriptors from the descriptor table. Fig. 4.4. illustrates the structure of the segment descriptor.

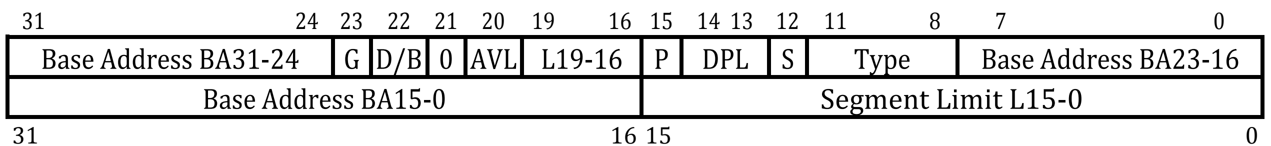

Fig. 4.4. Segment descriptor of the IA-32 architecture

As seen in Fig. 4.4, the segment descriptor occupies 64 bits, including 32 bits of the segment base address BA and 20-bit code of the segment length L. The segment length is equal to L + 1 bytes if G = 0 and L + 1 pages of the size 4 KB when G = 1. If the bit D/B = 0, then the relative addresses and operands are 16-bit words, otherwise, their bit width is 32.

The presence bit P = 0 means that this segment is absent in RAM and this descriptor should not be used to form an address. The DPL field indicates the segment protection level. The most secure segment has DPL = 0, and the least protected one has DPL = 3. The system bit S = 0 means that the descriptor provides the access to the system resources, such as task status

segment TSS, descriptor table, gateways to access the other threads, interrupt handling subprograms. The bit S = 1 provides the access to the usual programs or data segments.

In the field TYPE, the access rights to the segments of different types are coded. The software segments have the field TYPE = {1, C, R, A}, while the data segments have TYPE = {0, E, W, A}.

A bit A = 1 in this field means that this segment has already an access. OS periodically is seeking for segments for which A = 0 and removes them from the memory, because there was no access to them for a long time. This mechanism is releasing RAM for the other segments. Bit R = 1 means the reading permission for the program segment. Bit W = 1 allows the writing operation to the data segment. Note that the writing to the program segment is prohibited in any case. If bit C = 1, then the segment is permitted to be accessed by the programs with the less privilege.

Bit E indicates the direction of the data placement regarding the segment border. When E = 0, the data are placed starting at the address BA towards the increasing addresses, and if E = 1 they are placed starting at the address BA + L + 1 towards the reducing addresses. This is sufficient for the stack memory handling.

Fig. 4.5 shows the scheme of the translation of the virtual addresses to the linear addresses using the segment descriptor. The linear address can be used directly as a physical address or can be then converted into a physical address using the paging method, as it is shown below.

The descriptor of LDT is selected by the segment selector index from GDT. During this process, the access rights are checked by comparing the selectors RPL and DPL of the descriptor. If RPL > DPL, and C = 0, then an exception of the protection violation occurs. The segment descriptor from LDT specifies the address of the beginning and the end of the selected

memory segment, and the 32- or 16-bit relative address indicates the offset within this segment.

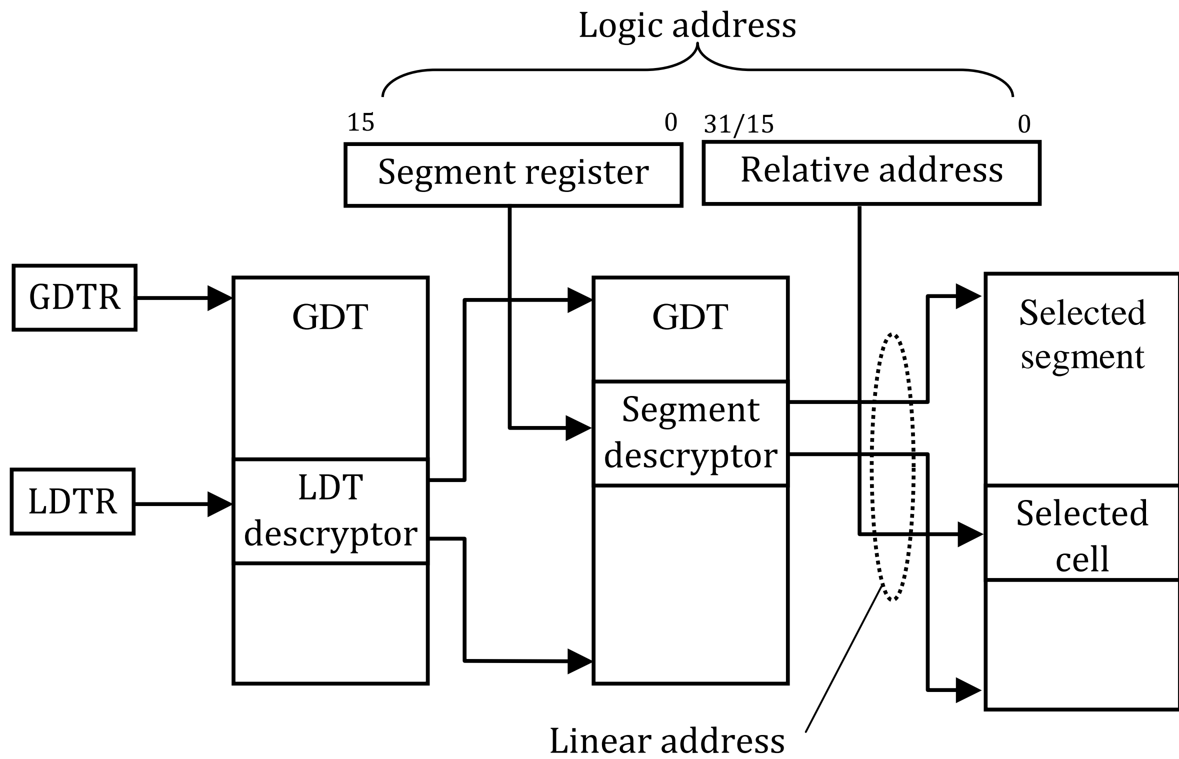

Fig. 4.5. Scheme of the virtual address to the linear address translation

In fact, reading the GDT and LDT tables occur only during the first access to the memory segment. The read descriptors are written to the shadow parts of the segment registers and LDTR, which serve as a cache memory for the descriptors. If the segment selector selects the GDT, then the segment descriptor is read from this GDT, which selects the physical memory segment without accessing LDT.

4.4.4 Paged memory organization

The paged memory is implemented by the processor only in the protected mode. The linear address is considered as the union of three fields (Fig. 4.6). The Directory field indicates the line number in the partition table with the 4 kB pages or in the page table with the 2 MB pages. The Table field

specifies the line address in the page table, and the Offset field does the relative byte address that is chosen on the page. Both Table and Offset fields address to a byte in the 2 MB page.

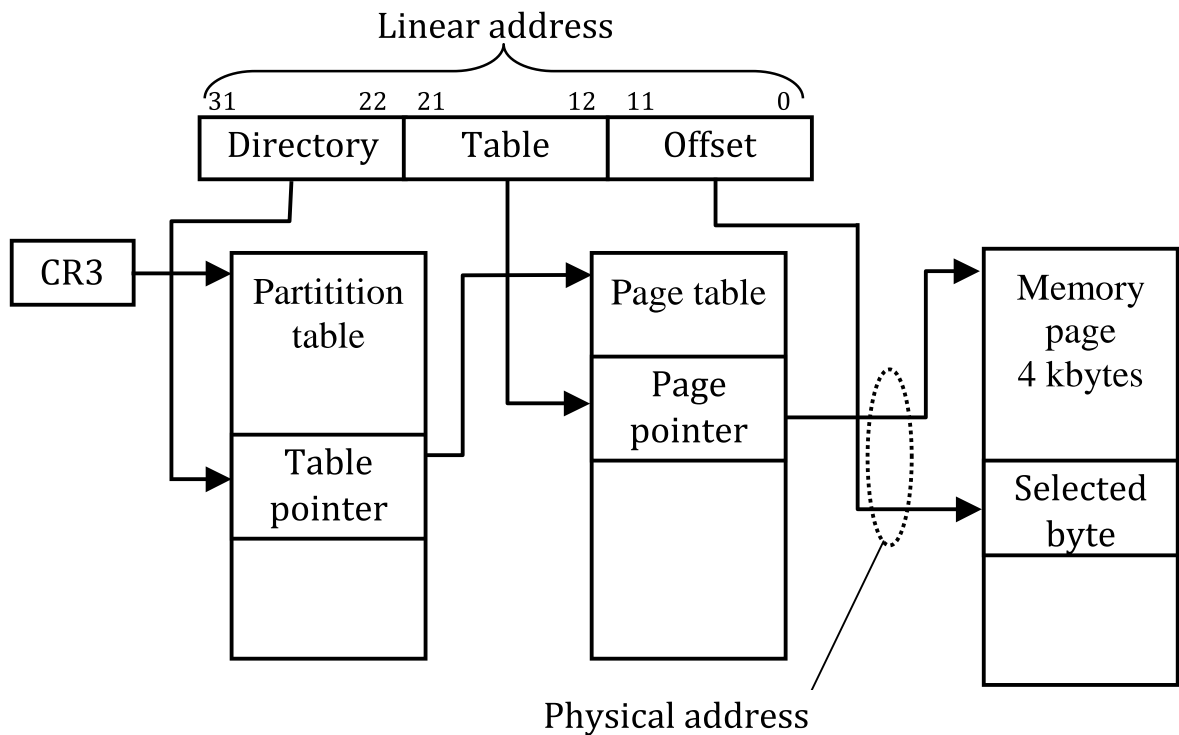

Fig. 4.6. The conversion of the linear address to the physical one

The control register CR3 stores the base address of the partition table and the bits which control the mode of the cache RAM for storing the pages. The PG bit in the control register CR1 selects either linear or paged addressing mode. The bits RAE and PSE of the control register CR4 set the page size equal to 4 KB or 2 MB in the of 32-bit or 36-bit physical address space.

The page table pointer format is shown in Fig. 4.7. The presence bit P = 1 indicates the presence or the page or table in the memory. If there is an access request to the user program segment (RPL = 3), then by U/S = 0, this access is forbidden, while by U/S = 1 and R/W = 0 it is allowed to read only.

Bits PWT and PCD are used to manage the cache RAM by the page addressing.

Fig. 4.7. Format of the pointer to the page table of the IA-32 architecture

The access bit A is automatically set while accessing the actual page. The modification bit D is set if there was a writing to this page. The operational system uses these bits as well as the bits AVL to determine which pages can be removed from the main memory.

The page size bit PS indicates the page size either 4 KB or 2 MB.

The globality bit G is set by the program to mark that the page is often used in solving the problem, and the base address of this page is not removed from the TLB buffer.

Apparently, to get an access to a certain logical memory cell, the processor must implement two additional accesses to the tables in the same memory. To speed up this process, the actual page pointer is automatically loaded into the TLB buffer, which serves as a small volume cache memory. Only when the access to another page is executed, which pointer is not present in TLB, a delay occurs to load a new page pointer to TLB.

4.4.5 Memory protection

The rules of access to the program and data segments are established in accordance with the key protection method and four privilege levels (see Fig.4.3):

— data segment can be selected by a program, which has the same or higher level of privilege;

— software segment can be called by a program with the same or lower level of privilege.

The Descriptor Privilege Level (DPL) (Fig. 4.4), the Request Privilege Level (RPL) in a segment register and the Current Privilege Level (CPL), which is stored in the code segment register CS are compared during each memory access.

The access to the data segment is performed using the selectors in the segment registers DS, ES, FS or GS. The access to the addressed data is permitted if the following comparison matches:

$$DPL \geq \max (RPL, CPL).$$

Note, that CPL = 0 means the maximum privilege level. If this rule is violated, then the interruption "general protection violation" is activated.

The access to the stack segment is performed using the stack segment register SS. It is allowed if DPL = RPL = CPL and if the selected segment has been granted to be written. The inequality is not allowed because for each of four privilege levels a separate stack segment is assigned.

If the access to the program segments is considered, then the dependent and independent code segments are distinguished. They are encoded by a bit in their segment descriptors. Typically, the dependent programs are ones, which may be called by the programs with different privilege levels. They are, for example, the subroutines in the system libraries. The access to the dependent segment is allowed if DPL ≤ CPL. The access to the independent segment is allowed if DPL = CPL.

To have an access to the independent segment with a greater privilege level, the ***gateway descriptor*** is used. This descriptor defines the entry point of the subroutine, which should cause code and a privilege gDPL. The operating system installs the gateway for specific user applications that require access to some system routines with high gain. This way excludes the possibility of unauthorized access to privileged applications.

The gateway descriptor is allowed to be read if its privilege level satisfies gDPL ≥ (RPL, CPL). Then its privilege level is compared to the calling subprogram privilege level, and the access is allowed if gDPL ≥ DPL. Consequently, the access to the independent segment becomes to be allowed if DPL ≤ CPL.

4.4.6 Interrupt handling

In the protected mode, the interrupt service routine is called through the Interrupt Descriptor Table (**IDT).** The base address and the limit of this table are stored in the register IDTR. Just as GDT and LDT, IDT stores the descriptors that provide a linear address formation and memory protection. But instead of segment base address and limit (see Fig.4.4) the interrupt descriptor stores the 16-bit segment selector and a 32-bit relative address that define the input point to the interrupt processing program. And the *i*-th descriptor corresponds to the *i*-th interruption. Thus, up to 256 descriptors take up to 2 kilobytes of memory.

The interrupt descriptors are distinguished as interrupt gateways or trap gateways, depending on the interrupt number. In the first case, a bit IF = 0 is reset in the state register EFLAGS during the interruption, which prohibits the further masked interrupts, and in the second case, the value IF is not changed.

During the interrupt vectorization, the respective interrupt descriptor with the corresponding number is selected by the hardware. The segment selector code is rewritten in the register CS. Further, the linear address is formed as in the case of a subroutine call (see Fig. 4.5).

During the software interrupt, the processor checks the value DPL of the interrupt descriptor. The subroutine call is executed if the privilege level CPL of the interrupted program, is below or equal to DPL, i.e. CPL ≤ DPL. Also,

the privilege level of the software segment with the interrupt service subroutine must be less than or equal to the privilege level CPL.

4.4.7 Multitasking

The fast task switching is the key to the performance of the modern processors. To support the multitasking, the architecture IA-32 has, in addition to the memory protection, a special data structure, called a Task State Segment (**TSS**) and the Task Register (**TR**), which stores the 16-bit TSS selector (a task number).

The obligatory part of TSS takes 104 bytes and contains all the context, which is necessary for the task. Additional, optional part of TSS may contain other information, which is useful for the operating system, such as the name of the task, comments, bitmap of allowed virtual interrupts (in the i8086 virtual processor mode) and the bit map of addresses of permitted IO devices.

The first two bytes of TSS store the selector of the TSS descriptor of the previous task, which execution originates the call of this task. This selector, called a return selector, is used to return to the previous task at the end of the current task. At this moment, it is rewritten back to TR. Some fields of TSS store all segment registers, general purpose registers, status register EFLAGS and the program counter, named as the (Expanded) Instruction Pointer **EIP**.

When switching the tasks, the TSS content of a new task is automatically rewritten in the registers of the processor core. During the next switching, the register contents is automatically saved in the TSS segment of this task, and these registers are rewritten by a new task context from TSS and so on.

The access to the TSS table, namely, the start of the context swapping between TSS and the core registers is initiated by the loading the task selector to TR, which points to the TSS descriptor of this task in GDT. Such a loading is

done by a privileged instruction LTR during the task initialization by the operational system. The second way of TR loading is implemented automatically when the processor performs the cross-segment jump JMP, or far subroutine call CALL, or return IRET instructions, forcing the task switching. The direct access to the TSS fields is not allowed.

When switching the tasks, the privilege level DPL in the TSS descriptor of a new task is compared to the privilege level CPL of the old task so that the new task should have no higher protection level.

To load the task with greater privileges by the CALL instruction, the task gate descriptor is used, which is stored in GDT, LDT or IDT. At this moment, the CALL instruction calls this descriptor, which in turn refers to the TSS descriptor, which is stored in GDT. And when a task is called with greater privileges, a pair of pointers to the stack in the respective fields of TSS are used, which have the respective privilege level. For example, when the system subroutine is called, then the system stack is used, and the pointer registers are SS0 and ESP0. This mechanism is done to ensure that tasks of each protection level has its own stack segment, and provides the content protection of the stacks.

A bit NT = 1 in the EFLAGS register indicates that the task is nested. So, after a series of switchings by the CALL instruction, the multiple nesting tasks are in operation. And by the IRET instruction, the switching to a task occurs, which selector is stored in the respective TSS field, when NT = 1. Otherwise, IRET performs normal return from the subroutine, returning from his own stack the contents of registers CS, EIP and EFLAGS.

The contents of a number of fields in TSS does not change. These fields are LDT selector of this task, CR3 content, stack pointers of the tasks with higher priorities, which this task can be switched to. They are filled by the operating system in a mode when the TSS segment is set as a data segment.

4.5. Virtual memory of the IA-64 architecture

Comparing to the 32-bit architecture such as IA-32, the 64-bit address space needs other mechanisms of the virtual memory handling. The 64-bit address can represent up to 16 billion GB of the address space. The number of entries to the page table is equal to $4 \cdot 10^9$ billion while forwarding 64-bit addresses and addressing the 4-kB pages. In addition, the 64-bit application unlikely uses much of the entire virtual address space in their programs, but the processed data can have a large volume.

These circumstances have led to changes in the virtual addressing of 64-bit architectures. Such an architecture is considered as an example of the IA-64 architecture, which gradually replaces the IA-32 architecture. Its key features include:

— the introduction of several address spaces (regions) and protection keys that enable better use of the TLB lines;

— the page size is variable and it is set from 4 kilobytes to 256 megabytes, which optimizes the TLB loading;

— the region identifier is added to the TLB line that allows the operation system to save the TLB state while switching the context.

Fig. 4.6 shows the scheme of the 64-bit virtual address translation to the physical address. Bits 61-63 define the virtual address register number of the region, which stores a 24-bit Region IDdentifier (**RID**). This identifier with the Virtual Page Number (**VPN**) determines the line number in the page table. The line of this table has two fields: the number of the physical page and access privileges, including key rights and access bits.

The RID registers allow the operational system to map eight of the 2^{24} possible address spaces of up to 2^{61} bytes each. The operational system uses RID to distinguish the general and private addresses. For example, region 0 is

used for the user data, region 1 is for the general libraries, region 2 is for the shared files, and region 7 is used for the OS kernel. Then, when the context is switched, not all TLB lines are canceled, but only those that are not shared.

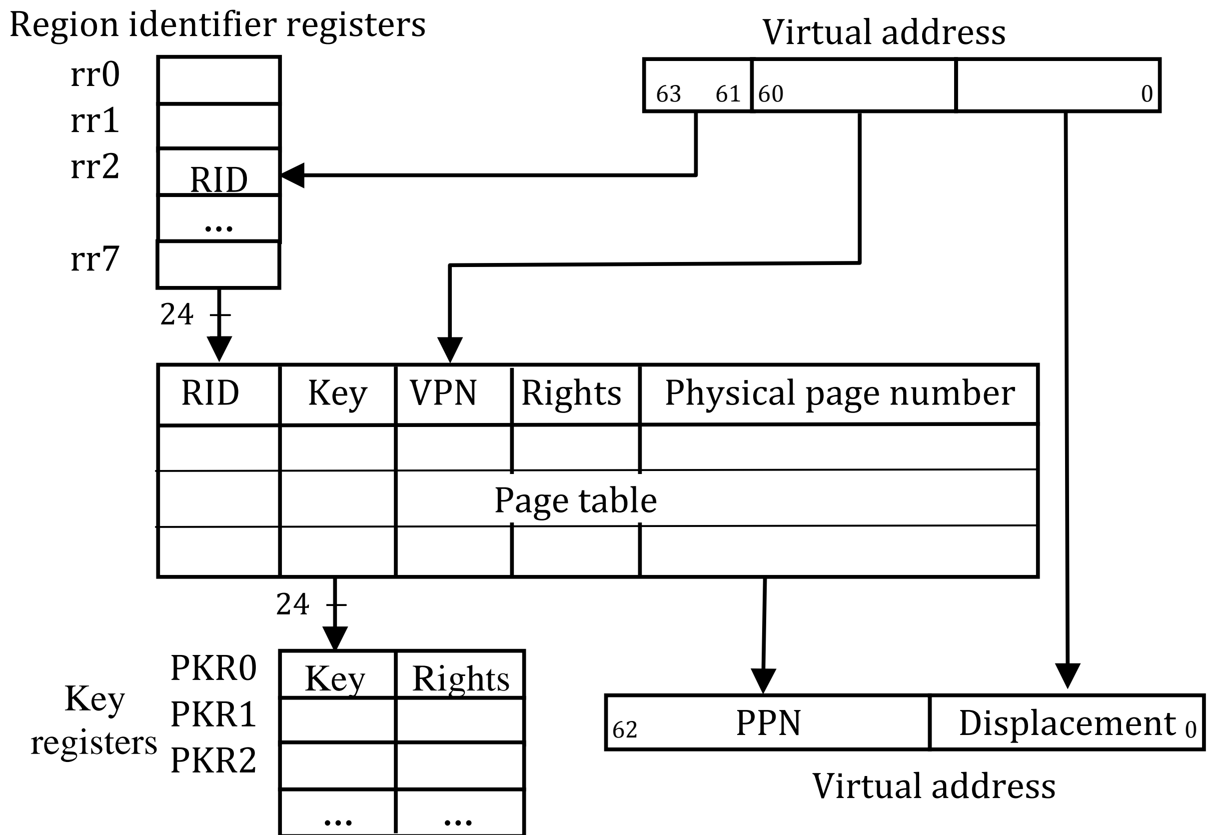

Fig.4.8. Scheme of the virtual address converting in IA-64

The region identifier registers allow the shared use of the memory regions. But for the programs, it is important to have an access to the shared memory areas of less volume, such as data tables, buffer areas between the programs. In this case, the Protection Key Registers (PKRs) (Fig.4.8) provide the access control at the page level. And the line in the page table has a protection key field, which is formed when the table is filled. If a row is selected in TLB, the CPU checks a match of the protection key field of the selected PKR register to the respective TLB field and authorizes the access when matched, otherwise, the memory protection interruption is caused.

To speed up the search for the access keys, PKR is organized as cache RAM. This arrangement allows for the virtual addressing multiple processes or threads that interact. They have the access to the same TLB line before and after the context switching without clearing TLB. This method increases the TLB effectiveness because its lines are shared by several programs. Such shared page access is called as the **virtual page aliasing**.

4.6. Problems

1) Why do the modern computers without the swap files?

2) Propose the methods for speed-up the page swapping.

3) Why is the virtual memory usually based on the memory paging technique?

4) Which architecture components provide the high-speed translation of the virtual address to the physical one?

5) For which purposes has the virtual memory architecture both page and segment organized memory?

6) Why is the write access to the TSS content prohibited even for the operational system?

7) Consider the IA-32 architecture operates in the time-slice multithreading with a period of 10 microseconds. Calculate the minimum overheads to the context switching if the access to RAM lasts 20 clock cycles. Note, that for a single memory access operation a single word is transferred, the clock frequency is 1 GHz.

8) Consider the conditions as in the problem 7, but the context the information is increased to 65K of data about the IO devices.

9) For which purposes the gate descriptors are used?

10) How many programs can be running in the IA-32 architecture simultaneously?

11) What is the maximum number of programs running in the IA-64 architecture?

5. MULTIPROCESSOR ARCHITECTURE

5.1. Fundamentals of multiprocessor architectures

5.1.1 Introduction

The third section was devoted to studying the possibilities of parallel operations in a single processor. Thus the basic ways to implement the scalar, vector and concurrent branch parallelism in these processors were described. It is believed that today the possibility of increasing the productivity of a single processor by the computation parallelization are already expired. Therefore, the further productivity gains are associated with the computer calculation organization in the multiprocessor architectures.

5.1.2 Characteristics of the parallel system tasks

Any application that is running on the processor with the von Neumann architecture, usually loads the processor, meaning it runs without the downtime. But once the processor utilizes the parallelism then the loading of its blocks is less than the maximum value, depending on the degree of the program parallelism. Some questions about the workload of the superscalar and pipelined processors depending on the code parallelism were discussed in the third section. Next, consider these questions about the multiprocessor systems. First of all, some definitions that characterize the work of parallel computing system are given regardless of the parallel system model, its interprocessor communications, distributed memory structure and operating system.

The ***parallelism level*** of the computational algorithm is the number of operations that can be performed in parallel. Consider a fast algorithm for

computing the sum $y = a_1 + a_2 + ... + a_8$. This sum can be calculated in three steps:

1) $y_1 = a_1 + a_2$; $y_2 = a_3 + a_4$; $y_3 = a_5 + a_6$; $y_4 = a_7 + a_8$;

2) $y_{12} = y_1 + y_2$; $y_{34} = y_3 + y_4$;

3) $y = y_{12} + y_{34}$.

As we can see, the parallelism level of this algorithm changes from step to step. At the first step, the parallelism level is $P_1 = 4$, and at the third stage, it is $P_3 = 1$.

An integral parameter, which characterizes the parallelism, is the ***average parallelism level*** P, which is equal to the ratio of the total number of operations to the stage number. In the algorithm given above the average parallelism level is $P = 7/3 = 2.33$. This algorithm is called as a doubling algorithm because at each step the number of operations is reduced by half. For the number of terms n, the number of the algorithm steps is equal to $\log_2 n$ and average parallelism level is:

$$P = \frac{1}{\log_2 n}\left(\frac{n}{2} + \frac{n}{4} + ... + 1\right) = \frac{n-1}{\log_2 n} \approx \frac{n}{\log_2 n}$$

For comparison, the average degree of parallelism while adding two vectors of length n is $P = n$. Recall that this algorithm refers to an algorithm with the natural parallelism.

The ***acceleration*** of the parallel algorithm is equal to the ratio of time $T_1$of its execution in a single processor to the time T_P of its execution in p processors, i.e.

$$S_P = \frac{T_1}{T_P}. \tag{5.1}$$

The efficiency of the parallel algorithm is equal to

$$E_P = \frac{S_P}{P}.$$

which characterizes the average CPU loading by the algorithm implementation. If the algorithm reaches the maximum achievable acceleration, then $S_P = P$ and $E_P = 1$. But in fact, the algorithm efficiency is close to unity only in the trivial cases. It is reduced not only from the small degree of algorithm parallelism, but also from its ineffective execution in a particular computer system. For example, the efficiency is reduced when the algorithm schedule is unbalanced or when the time losses to the data transfers and the memory conflict resolvings are great.

Data preparation time is a delay that is caused by the data exchange, shared memory conflicts or synchronization. This time is needed to place the data, which are required for calculations or the results in corresponding memory cells.

Load balancing is the distribution of tasks or operations of an algorithm among PUs of the system that allows the operational system to load them by the useful work as a larger share of the time. Often such a balancing is refereed to as a mapping the algorithm in the computer system.

The load balancing is static or dynamic. During the static balancing, operations or tasks, and often data are distributed on PUs before the computation start. Typically, such a balancing is performed by the compiler or by the programmer. By the dynamic balancing, the operations, tasks and data are distributed between PUs during the computing process. In this balancing, the system manager selects the ready to perform tasks from a task queue and assigns them to PUs, which are freed.

Consider the following mathematical model of parallel algorithm acceleration, which is a specification of the formula (5.1):

$$S_P = \frac{T_1}{(\alpha_1 + \alpha_2/P)T_1 + t_d} \qquad (5.2)$$

where α_1 is the share of operations, which are performed by a single PU, α_2 is the share of operations performed with a parallelism level P, t_d is the total time of the data preparation. By $\alpha_1 = 0$, $\alpha_2 = 1$, $t_d = 0$ we obtain $S_P = P$ and the maximum acceleration. Consider a case, when $\alpha_1 = \alpha$, $\alpha_2 = 1 - \alpha$, $t_d = 0$. Then the formula (5.2) looks like:

$$S_P = \frac{1}{\alpha + (1 - \alpha)/P} \tag{5.3}$$

This formula expresses the ***Amdahl's law***. This law provides that the calculations are performed by the maximum or minimum of parallelism. Let the algorithm contains one-tenth of operations which cannot be performed in parallel, and the rest of operations can run in parallel as much as possible. Then $\alpha = 0.1$ and (5.3) takes the form

$$S_P = 1/(0.1 + 0.9P) = 10P/(P + 9) < 10.$$

Thus, when any number of PUs with a perfect balancing of their load and there is no overhead in the data preparation, then for $a = 0{,}1$ it is impossible to speed up the calculations more than ten times.

Also, consider a case when in (5.2) time t_d approaches to T_1 or is greater than it. Then for an arbitrary number of PUs and any degree of parallelism, we get $S_P < 1$, that is, a slowdown of the algorithm takes place. So it is possible, that the problem has the intensive data exchanges, frequent memory conflicts, and costly synchronization. Then using a multiprocessor system is less profitable than the programming a single processor. This, of course, is an extreme case. But because of the large value of t_d in many problems a limit is reached when the increase in the number of involved PUs does not justify itself.

So, to organize the parallel computing is preferable only when not only by a high degree of the algorithm parallelism, but when it is possible to achieve a decent acceleration of the algorithm implementation in a parallel

system. Exceptions may be a case when the amount of data to be processed does not fit in the memory of a single PU. Another case is when the purpose of the parallel processing is not its acceleration, but the ease of use, for example, by keeping the distributed database.

The design of the software for the high-performance parallel computers performance is the non-trivial task and it may take hundreds and thousands of hours. Of course, the new parallel program is created if it is assumed, that it will be used quite often by many users, or if it is planned, that its timely results would have the extraordinary value. Otherwise, it should not start a programming because its accelerated performance will not cover the cost of its creation.

5.1.3 Multiprocessor architecture classification

There are two main types of information flows in the computers: instruction flow and data flow. The flow of instructions is a sequence of instructions that are decoded and executed by a CPU. The flow of data is data that is transferred between the RAM and the ALU in CPU. Different computer architectures have involved different numbers of instruction and data flow. Flynn was the first in 1966 who has noted that the computer architectures are divided into well-defined sets on whether the flows of instructions or data is single or multiple ones. He divided all the architectures into four categories:

— single instruction flow, single data flow (**SISD**),

— single instruction flow, multiple data flows (**SIMD**),

— multiple instruction flows, single data flow (**MISD**),

— multiple instruction flows, multiple data flows (**MIMD**).

Conventional von Neyman computers fall into the category of SISD-processors. Parallel computers are either SIMD or MIMD-computers.

If a single control unit is using and all processors perform the same instruction in sync, then such a computer is classified as SIMD. In MIMD-computer, each processor has its own control unit and can perform different instructions on different data.

The category MISD assumes that one data stream is processed by different programs, i.e. various streams of instructions. In fact, these computers are not in use. The pipelined computer is similar to the MISD-model in that data flow passes through the uniform pipeline stages, which carry a set of microinstruction flows according to the flow of the instructions.

The parallel MIMD architecture consists of several processing units (PUs) and memory blocks connected to each other through the switching net. According to the principles of the data exchange between PUs, the MIMD architectures are divided into two categories: shared memory architecture and messaging architecture. In the ***shared memory architecture***, PUs communicate through a central memory, which is shared between them. To transfer the data, PU writes it into a common central memory cell, which is read by another PU (Fig. 3.2). This computer architecture is called as a ***multiprocessor***.

In the ***messaging architecture***, PU sends the data represented in the form of a message to another PU strigt through the network. Thus, it is not passed in two steps, as in the multiprocessor, but in a single step.

The computer memory can be concentrated in a single memory block or may be distributed among all the PUs. Accordingly, the MIMD architecture is divided into the architecture with the global memory and with the distributed memory. The messaging architecture is typically the distributed memory architecture. But there are also multiprocessors with the distributed memory. They have the shared memory which is evenly distributed among all PUs, each PU "knows" which PU contains a particular shared memory cell. If

the data, which must be read, is stored in another PU, the special FSM or the operational system sends a message to that PU that it has to send the addressed data. This architecture is called as a Distributed Shared Memory (**DSM**) architecture.

In section 3.6, a big role of the consistency of accesses to the shared memory was noted. The easiest way to achieve the memory access consistency occurs if the time delay of the access to any memory cell is the same for all PUs. The architecture, which satisfies this property, is called as a Uniform Memory Access architecture (**UMA**). The multiprocessor with such a uniform access is called as a Symmetric MultiProcessor (**SMP**).

In another kind of architectures, the access to the shared memory cells, which reside PU, is much shorter than the access to the cell located at the other PU, i.e., the memory access latency is non-uniform. This architecture is known as a Non-Uniform Memory Access (**NUMA**) architecture. Consequently, the DSM architecture is usually the NUMA architecture.

Among DSM-architectures are those, in which the virtual addresses of the cells are permanent. But these cells migrate constantly from one PU to another one. If PU requires a certain memory page with the data, it finds it in another PU, rewrites it and sells a message to the rest of PUs, that now it is the owner of this page. Thus, the local PU memory behaves as a cache RAM, and the shared memory block is not used, so this architecture is called as a Cache-Only Memory Architecture (**COMA**).

To send the data from one PU to another one, it is not necessarily to use the memory cell at the address, which is specified by the programmer. This address can be the random one. Such an architecture, which is based on the randomly addressable memory, is named as a Parallel Random Access Machine (**PRAM**). In this architecture, the logical address is converted into the random physical address. Due to this, the computer becomes much fever

conflicts of parallel access to the adjacent memory cells because the actual accesses are implemented to the remote cells, which are stored in different memory units. Some SMP architectures are implemented as the PRAM-architecture.

Often there are combined and hierarchical architectures. For example, the widely used ***cluster computing systems*** have the DSM architecture with messaging at the top level. They contain from tens to thousands of nodes. Each node represents a lower level of the hierarchy and is an SMP-processor with the shared global memory containing from 2 to 128 PUs.

5.1.4 Switching system of the multiprocessor computers

According to the formula (5.2), the algorithm accelerating depends significantly on the time of the data preparation. In the parallel systems, this time is determined mainly by the interprocessor communication delays. Therefore, an important factor in how the PUs interact each other in the computer system is a switching system. Its parameters and the principles of the physical implementation substantially affect the computer's performance as a whole.

The switching systems are classified by a set of features such as operation mode (synchronous or asynchronous), management strategy (centralized or decentralized management), the method of switching (circuit or packet switching) and connection topology relations (topology graph, static or dynamic topology).

In the switching system with the synchronous operation mode, the transmitted data are accompanied by a clock signal, which indicates when these data are valid. Often one clock signal source is used by all PUs of the system. So, PUs are performing the data transfers per clock cycles.

In systems with the asynchronous mode, the destination PU sends a signal to the source PU, which confirms the data reception. Because of this, such systems are slower than synchronous systems. But they prevent the bad signal races, provide the reliable information transmitting and less risk of blocking the computational process.

In modern computers, a combined switching mode is usually used., At the lower level, the level of bytes, words, the synchronous mode is used, and at the top level, the level of packets, the asynchronous mode is implemented.

In the switching systems with the ***centralized control***, a single central control unit is used for controlling the data exchange between all the components of a computer system. In the systems with the ***distributed control***, these functions are distributed among all system components. In the case of the failure of the central control unit, for example, in the general switch of the multiprocessor, the whole computer system is failed. But for the distributed control, such as the system with the local network, the failure of a single node does not cause the system failure.

In accordance with the principle of switching the data flows in the computer networks, the switching systems are classified as ones with network switching and ones with packet switching. In the system with the ***network switching***, the data transmission path is established between the source and destination before the data transfer, which is sustainable over the transfer of all data through it.

In the ***packet switching*** system, a set of data is divided into particles, called packets. When forwarding packets from the data source, the packets are moving from one node of the connection system to another one with the intermediate storing until they achieve the destination. It has to be taken into account that the travel routes of different packets of a single data array may be different and have different delays.

5.1.5 Switching system topology

The most important aspect of the parallel computing system is how its PUs interact each other. This is particularly important for systems that have only local memory in PUs. The interprocessor communication configuration significantly affects the order of connections of the lines that interconnect the individual processors.

The ***topology*** of the switching system is a function that maps a set of PГs and memory blocks to a set of links. In other words, the topology describes how to connect PUs and RAMs to each other. The most common way for expressing the topology is the topology graph, whose nodes denote PUs or RAM blocks and edges relationships between them. Often the PUs of the parallel system are called as the nodes.

The choice of the computer topology is a trade-off between the switching system complexity and latency of the data transfer between any PUs. Thus, the cost of the switching system may outweigh the cost of processors in the SMP architecture. The complexity of the switching system can be expressed in a number of edges in its topology. This number is two times less than the total number of the input-output (I/O) ports of the nodes. The number of I/O ports of PU is equal to the degree of the respective node.

The data transfer delay is estimated by the diameter D of the topology graph. It is equal to the maximum number of steps (hops), which are necessary to transfer a packet between any of PU in the system. Here, a hop means sending a packet between two adjacent PUs.

In a system with the full graph, the topology function maps each PU to all the PUs of the system (see Fig. 5.1, a). Therefore, this topology is called as every with each topology. Its diameter is $D = 1$, but it is necessary that each of P PUs has at least P–1 I/O port, which is unacceptable for a large number of P.

In the systems, in which each PU is connected to several neighboring PUs, the system designer can achieve a compromise between the of the communication complexity, and its bandwidth and latency. The grid or lattice topology has such a property. In the one-dimensional lattice, each PU, except for the outside ones, is connected to two adjacent PUs, ie, the i-th PU is connected to the $i-1$-th and $i+1$-th PUs. This is so-called, linear topology (see Fig. 5.1, b). The data, which are sent from source PU to destination PU, consistently pass the transit nodes, located between the source and destination. If the source and destination are outside PUs, then $P-1$ hops are needed, i.e., $D = P-1$.

The ring topology (Fig. 5.1, c) differs from the linear one in that, that its first PU is connected to the P-th PU. This reduces the system diameter from $P-1$ to $P/2$. The links in the ring system may be both bidirectional and unidirectional.

The two-dimensional lattice structure is widely spread. Each PU in it is connected to the northern, southern, eastern and western neighbors (see Fig. 5.1 d). For such a square grid of P PUs, the diameter of the system is equal to $2\sqrt{P}-2$.

The outside PUs of the two-dimensional array can be connected, forming a toroid topology (Fig. 5.1, e). This reduces the diameter of the system by a half. Such a topology has the famous in 70's ILLIAC-IV system, in which 64 PUs were connected in a two-dimensional grid of dimensions 8·8.

The grid processor characterized by the simplicity and locality of the interprocessor communications. In a three-dimensional grid, each of the PUs is connected with six neighbors (Fig. 5.1, h). The two- and three-dimensional grids are used in the supercomputers. Their topology is consistent with the graphs of the algorithms, which perform the modeling the physical substance with the extremally high speed.

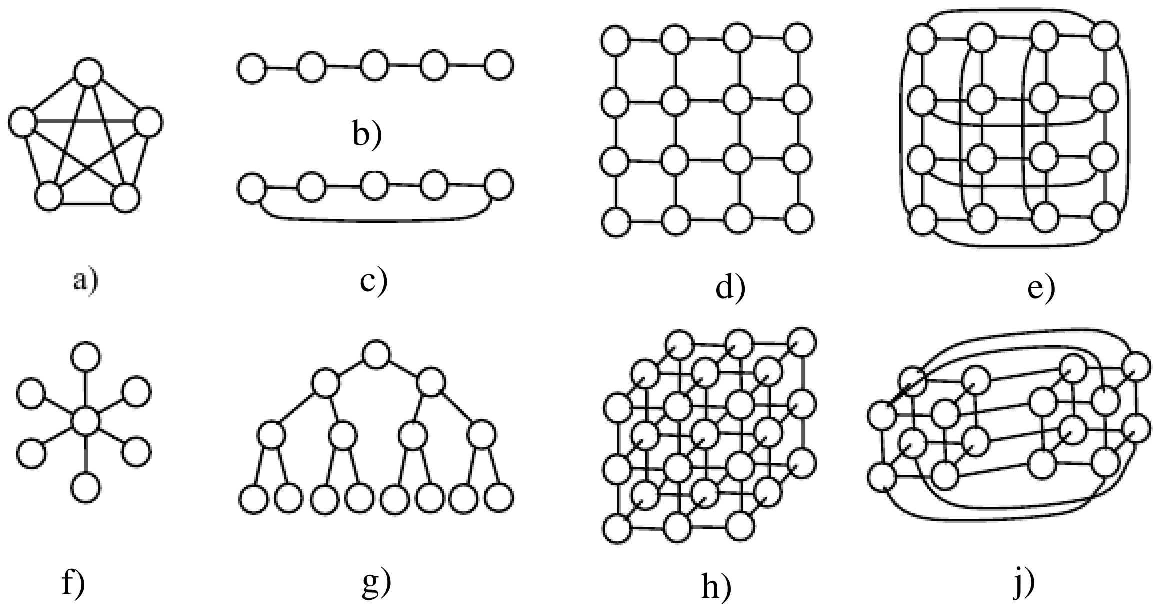

Fig. 5.1. Topologies of the multiprocessor systems

The hypercube topology is obtained when building the n-dimensional lattice for $P = 2^n$ processing units (Fig. 5.1 j). The diameter of the hypercube is only $\log_2 P = n$. The number of I/O ports of each PU is n. This number defines the boundary of the increasing of the PU number in real systems with large numbers of P. For example, for a system with $P = 256$ PUs the number of I/O ports for the interprocessor communications is $\log_2 256 = 8$.

In many high-performance computing applications, the computations start in a single subtask. Then they gradually spread to a large set of parallel subtasks and the result is sampled in one completing subtask. The structure of such a computational process is especially suited to the tree configuration of the interprocessor communications. The tree of PUs consists of several layers. One processor of the upper layer is connected to q PUs from the next layer. If $q = 2$, then the tree is a binary tree (Fig. 5.1 g).

The diameter of the binary tree, which consists of $2n–1$ PUs, is equal to $D = 2{\cdot}n - 2$. If multiple data movings are running between PUs of the lower layers, the conflicts arise from the use of links in the upper layers of the tree.

To reduce such conflicts and reduce the latent delays, the "thick" tree topology is used. In this tree, with each branch to a higher level, the communication bandwidth is doubled.

The star topology (Fig. 5.1, f) can be considered as the degenerate case of the tree topology. The diameter of the star is only two. But the communication bandwidth of the system is limited by the bandwidth of the central PU. Also in the case of a failure of this PU the whole system fails.

The system with a common bus has the star topology if we assume that the central node in Fig. 5.1, f is a bus. Therefore, this system has all the inherent disadvantages of the system with the star topology. Also, the conflicts frequently occur when several PUs are trying to capture the common bus.

The multiprocessor topology is mainly determined by the structure of a switch that connects the PUs with the Memory Units (MUs) of the shared RAM (see. Fig. 3.2).

The ***crossbar switch*** allows the system to achieve the complete connectivity of PUs and MUs with the minimized cost (Fig. 5.2, a). The vertical lines of the cross switch are connected to q MUs, and the horizontal lines are connected to P PUs. The switches are installed at the intersections of vertical and horizontal lines. But their number $P \cdot q$ for the complex systems becomes too large. Assuming the switches the individual nodes of the topology, the diameter of the cross switch is equal to two.

The ***multistage*** or ***mesh switch*** is widely used as a universal switching system between PUs and MUs. The multistage switch for connection of P PUs and P MUs has $\log_2 P$ stages. Each level has $P/2$ switches. Each of them passes data directly or cross.

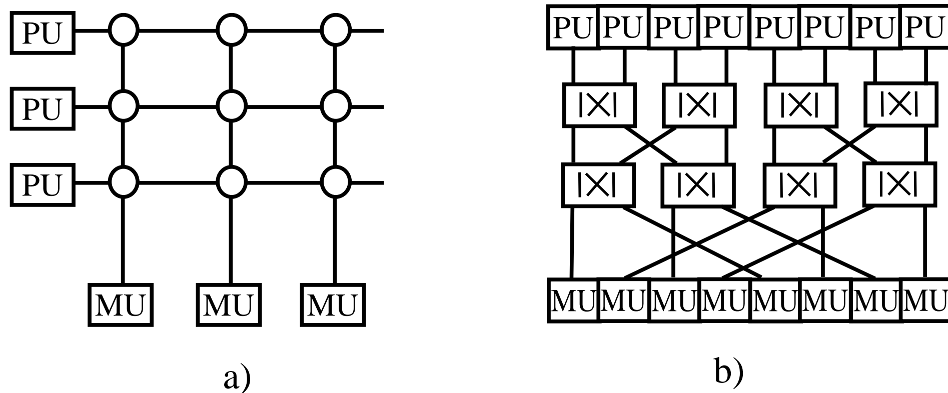

Fig. 5.2. Topology of switching systems

This switch is much simple and technologic than the crossbar switch. But the memory access time (latent delay) in it is increased to $2 \cdot \log_2 P$ delays of a switch. In addition, the conflicts arise it often by using a switch on two different routes. To increase the bandwidth, the switches have the intermediate registers. Then, the data transmission is performed by a pipeline principle using the packet switching.

The rectangular switching network is promising, which has the multi-switch topology with the two-dimensional lattice of PUs. Each node of the network is the basic processor that performs only switching function. Getting the route in this network is performed dynamically due to the principle of a worm: the packet head makes the “hole” for itself and the subsequent body of the package via free transmitting nodes to the destination as well as a worm is making its way through an apple. Because this network uses only local connections between nodes and the pipeline transfer principle, its capacity is quite high, although it has a large latent delay.

The disadvantages of certain types of communication systems are minimized in their combinations. So the hybrid interprocessor communication systems are designed. For example, the topology of the pyramid is obtained by adding the links between processors belonging to the same tree

stage, according to the two-dimensional lattice topology. Thus, in the pyramid topology, the advantages of a tree and a lattice are combined.

The structures of the computing clusters also apply to the hybrid systems. Within each cluster, PUs are connected closely to each other, and several clusters are connected to the system in accordance with different topologies, such as the lattice or hypercube. This hybrid topology formation can be continued recursively getting the clusters of clusters.

5.1.6 Problems

1) The program contains 100 procedure calls, each of them is executed for 1 ms, 90 procedures can be executed in parallel on 5 PUs. Calculate the average parallelism level, the maximum program acceleration, and the program efficiency.

2) The conditions as in the problem 1, but the real number of processors is equal to 3.

3) The program contains 1500 instructions, 200 of them are executed only in a sequence. Estimate the maximum program acceleration.

4) The computer contains 128 PUs and a common one port memory. The access time to the memory is equal to 20 ns per data. The speed of PU is 2 ns per instruction. The parallel subroutine executes the addition of two data vectors of the length of 1024, which are stored in the common memory. The share of operations, which are executed in a single PU is equal to 1%. Estimate the algorithm acceleration and its efficiency. Explain the factors, which infer the speed-up of the subroutine.

5) The conditions of task 4, but the PU number is 16.

6) The conditions of task 4, but the data transfer is executed in the pipelined mode, i.e., one data per clock cycle.

7) What are the advantages and disadvantages of the distributed shared memory MIMD architecture contra the single shared memory MIMD architecture?

8) What are the advantages and disadvantages of the messaging MIMD architecture contra the multiprocessor architecture?

9) What are the advantages and disadvantages of the NUMA architecture contra the UMA or SMP architecture?

10) What are limitations of the SMP architecture scalability?

11) What is the main feature of the PRAM architecture and why?

12) What are the differences and features of the network switching and packet switching systems?

13) Consider the computer system with 32 PUs, and with the linear topology. How much do the speed-up and the effectiveness increase if the topology becomes the ring? Note, that 1% of operations are executed in a single PU, the preparation time is equal to the time of PU operation and is proportional to the system diameter.

14) The conditions of the problem 13, but the topology becomes the grid 4x8.

15) The conditions of the problem 13, but the topology becomes the hypercube.

16) What are the advantages and disadvantages of the crossbar switch contra the mesh switch?

5.2. Processors with the SIMD parallelism

5.2.1 What is SIMD

The concept of SIMD is a method of increasing the speed of executing the algorithm with the natural parallelism. Traditionally, when one or more operations are performed on a data array, a software loop is executed, which iterations calculate all elements of the array. During each iteration, one group of data is calculated by one group of instructions. The SIMD concept can be explained by the unrolling the program cycle with the simultaneous execution of operations in its iterations in individual PUs. Thus, SIMD — Single Instruction — Multiple Data — means that an instruction from the instruction stream executes several identical operations on data in a set of PUs. Here, the data level parallelism takes place here. Accordingly, the SIMD-system is a computer system with multiple identical PUs, which are performed a single instruction at different but homogeneous data at the same time.

Sometimes the SIMD-system means a computer paradigm, in accordance with which the PUs of a multiprocessor system are loaded by the equal software modules that perform the same operations on the arrays of similar data. In this case, a system is called as Single Program — Multiple Data (**SPMD**).

Firstly the SIMD-architecture was used in the supercomputers of the first generation, such as Illiac-4. In our country, the mass-produced computer PS-2000 had an SIMD-architecture. Now, this architecture becomes the mass distribution due to the increased processing power, which is needed for the multimedia data processing. The SIMD-systems are used both in CPUs and graphics accelerators, which are considered below.

The three-dimensional graphic computing uses the SIMD-system architecture especially effectively since it requires intensive processing of

three- and four-dimensional matrices. The library Microsoft Direct3D is oriented to an architecture, which contains the SIMD-computer system with a special structure. A striking example of the SIMD-system use is the video game console. Almost every modern system for the video games, starting with the Sony PlayStation, has a built-in SIMD-system.

Consider the main features of the SIMD-architecture.

5.2.2 Data representation

The data are represented as vectors in the SIMD-architecture. But unlike the vector-pipelined processors, the vector components are processed not sequentially but in parallel. Because of this, the data memory has a characteristic structure that allows the computer to read several vector elements at once.

This memory is designed as the sliced shared memory or as the memory distributed among PUs. In the first case, the memory has P inputs and outputs connected to the corresponding PUs. In the second case, each PU has its own local memory, while each PU accesses its RAM at the same address.

In the last two decades, the SIMD architecture became widely known in relation to the multimedia applications in computers. The most common data representation in it is, so-called, ***packed vectors***. The CPUs have the corres-ponding register memory that stores these packed vectors of the respective bit width. For example, 64-bit data word stores eight bytes of data in the packed form. Last time the CPU architectures are expanded, which packed vectors have 128-bit width.

5.2.3 Control

The control signals to the SIMD-processor are propagated from a single source, named the control unit, which selects and executes instructions

from one instruction memory. Since each PU executes the same instruction simultaneously, the control hardware of the SIMD-processor is simplified, and all calculations can be performed in the pipelined mode with a maximum clock frequency. But all these PUs must execute one instruction in any circumstances. To weaken this condition, the instruction masking is used in PUs. Then, the mask code determines, which of PUs are allowed to execute the current instruction, and which of them must be idle.

If the SIMD-system is rather large, then the instruction sending from a single source lasts a significant period of time. In this case, to preserve the performance, a group of PUs and even each PU has its own control unit. Thus all these PUs perform the same program. That's what the modern graphic accelerators are designed, which are discussed below.

5.2.4 Vector instruction set

If the SIMD-system is small (no more than a dozen of PUs), then the system uses the vector instructions, which calculate the packed vectors of data. This instruction set usually involves the operations of effective reading or writing, arithmetic operations (addition, subtraction, multiplication, division, square root, etc.), logical operations (AND, OR) and comparison. The SIMD-instructions are available for programming in the assembly language. Some instructions are used for the data shuffling, i.e. they are used for different exchanges of data between PUs. For the high-level language programming, the procedure calls, described in assembly language, are usually used.

The vector instructions are found in most processors, including the following instruction subsets: IBM's AltiVec, PowerPC SPE, Intel MMX, SSE — SSE5, AMD 3DNow!, ARC ARC Video subsystem, SPARC VIS, Sun MAJC, HP MAX for PA-RISC, ARM NEON, MIPS MDMX , MIPS-3D.

5.2.5 SIMD-processor topology

In the most cases, the SIMD-system has the linear, one-dimensional topology. In the simplest case, PUs are connected through a common bus. In more complex cases, PUs are connected in a line or ring (Fig.5.1, b, c). In an extreme case, they are connected to the matrix switch (Fig.5.2). In the SIMD-MMX system, it is believed that the PUs are connected in a ring.

The two-dimensional SIMD-systems have the lattice or torus topology (Fig.5.1, d, e). So, these computer systems are often called as a matrix processor. The multidimensional SIMD-systems have the topology of lattice, hypercube, tree and others.

5.2.6 MMX architecture

The MultiMedia eXtension (MMX) coprocessor of the Pentium II architecture handles integer data type of the bit width: 8, 16, 32 and 64. These data are packaged in 64-bit registers. So, the eight bytes of data are calculated by eight parallel operations in one instruction cycle. In the case of 16-bit data, a single instruction executes four operations. To store the packet data, the MMX architecture uses eight 64-bit general purpose registers. They are mapped in the floating point coprocessor registers in the address space.

In this architecture, the conditional execution of instructions is utilized. Due to this approach, the expression of the form:

if <condition> then Ra = Rb else Ra = Rc;

is replaced by the expression:

Ra = (Rb AND Rx) OR (Rc AND NOT Rx),

where the mask Rx depends on the condition value and may determine the selection of Rb or Rc in parallel. This means that the conventional operators are replaced by the effective sequence of the unconditional operators, simplifying the algorithm parallelization.

The features of the MMX architecture are explained by its programming model. This model is illustrated by Fig. 5.3 and Fig. 5.4.

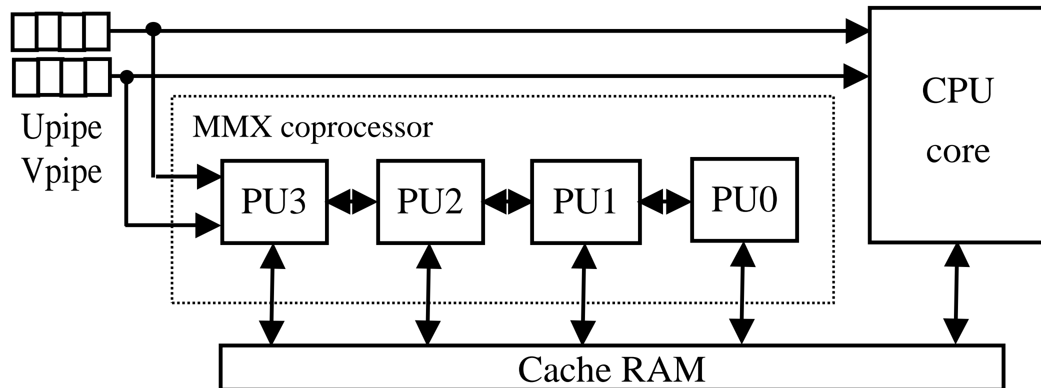

Fig. 5.3. Structure of the Pentium II CPU with the MMX coprocessor

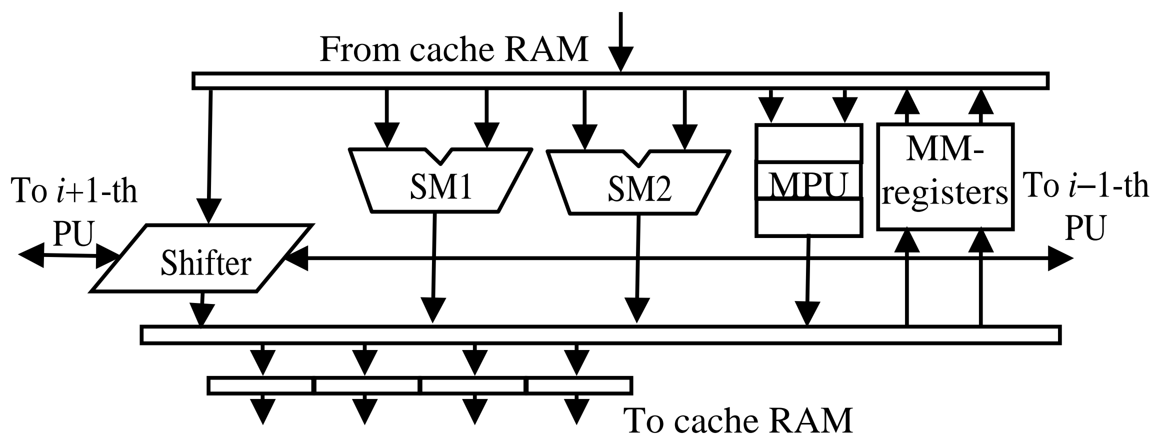

Fig. 5.4. Structure of a single MMX PU

Fig. 5.3 shows the interaction of the main CPU and the MMX coprocessor. Fig. 5.4 shows the structure of one PU of the MMX coprocessor. The last one contains four PUs in the case of 16-bit data or eight PUs considering the byte operands. In this structure, in accordance with the principle of the superscalar processor, each PU executes either one or two instructions coming from the instruction pipelines U and V. The processor core executes the control algorithm and calculates the address flow for the cache RAM in parallel with the MMX coprocessor.

The MMX-instructions, except multiplication, are executed for a single clock cycle. The multiplication is performed by three cycles in the pipeline mode with one clock period. The data exchange between PUs is implemented by the shift instruction.

5.2.7 SIMD-system advantages

The advantages of the SIMD-system occur if the system topology corresponds to the executable algorithm. These are applications where one operation is executed over an array of data. Such applications are signal processsing, multimedia algorithms, linear algebra problem solving. For example, to change the image brightness, for each pixel the same value is added.

Consider the example of the 8-byte packed vector data. Its processing is performed eight times faster than when processing the individual bytes. And three MMX instructions exchange the brightness of eight pixels in parallel. Thus, the benefits of SIMD-systems appear to accelerate calculations compared to the superscalar processors.

To design and program an SIMD-system, often a method of systolic processor development is used. The ***systolic processor*** is an array processor, in which the calculations are organized in the pipelined mode. And the arrays of input and intermediate results are rhythmically pushed through the layers of PUs as through large pipeline stages, recalling the systolic blood waves coming through the tissue.

Under this method, the algorithm, described by the cycle nest, can be automatically mapped into the systolic processor structure and further be loaded in the structure of the SIMD-system. The advantage of this method is that the PUs of the system are fully loaded by calculations and the cost of the data transfers is minimized. Therefore, the optimum efficiency of the parallel algorithm execution is achieved in these systems.

5.2.8 SIMD-system disadvantages

The main drawback of the SIMD-system is that it effectively performs a limited set of algorithms, in which parallelism is specified explicitly or in the form of program cycles. It is desirable that the number of PUs corresponds to the size of data arrays, the interprocessor transfers are minimized, and more. In other cases, the SIMD-system programming must be performed manually, which is very expensive and time-consuming. In cases of complex data relationships, the data transfer costs can "eat" all the benefits of the computing acceleration.

As for SIMD-architecture in the modern microprocessors, their advantages show themselves only when they are programmed by the assembly language. In addition, in the most architectures, the vector registers are shared between the multimedia coprocessor and the floating point coprocessor. This feature reduces the speed if the algorithm uses both floating point and packed vector data because the switching the integer and floating point modes take a lot of time overheads. Besides, some overheads are needed to pack the data into the vectors and to unpack them back.

5.2.8 Problems

1) What are the limitations of the SIMD architecture, i.e. in PU number, network diameter, instructions per second, etc.?

2) What are the differences between SIMD and SPMD architectures?

3) What is the limitation of the packed vector length?

4) For which problem solving are the SIMD systems intended, which have the 3-d lattice topology?

5) For which problem solving are the SIMD systems intended, which have the hypercube topology?

6) For which problem solving are the SIMD systems intended, which have the tree topology?

7) The SPMD system has the linear topology of $n = p^2$ PUs. Consider the problem of the solid modeling by the finite particle method. The solid is represented by the 2-dimensional grid of dimensions $n \cdot n$. One node of this grid is modeled by 100 PU instructions. A single modeling iteration consists of receiving data from neighboring nodes, performing mathematical computations and sending the results to the neighbors. How much the computation speed increases when the system has the 2-D grid topology of $p*p$ PUs. Note, that all the instructions including the communication ones have the same execution time.

8) The MMX instruction set has the instruction PADDB for packed byte vector addition and the instruction PSLLQ for shifting left the quad word, stored in the MMX register, to the bit number, which is the second operand. Propose an assembly language sketch, which performs the addition of 16 bytes, stored in the MMX registers mm1 and mm2. The result must be stored in the highest byte of the register mm0. What is the speed-up factor of this program comparing to the program, written using the scalar instructions?

9) The conditions as in the problem 7, but the task is to shuffle the packed bytes that the 0-th byte occurs in the 7-th place, the 1-st byte is in 6-th place and so on.

10) The program contains 100 RISC-type instructions and 100 MMX instructions, which execute the packed byte vectors. What is the algorithm speed-up factor comparing to the program, which consists only of the RISC-type instructions, not taking into account the instructions of packing and unpacking the vectors?

5.3 Multiprocessor Architectures

5.3.1 Problem of the sequential consistency

The multiprocessor MIMD-architecture is ***sequentially consistent*** if the results of any parallel computations in it is the same as the results of the corresponding sequential program, executed by a single processor.

The issues of the memory access consistency were discussed in Section 3.6. It was also shown, that if a computer system is subject to sequential consistency, then this leads to significant execution delays. This problem can be partially solved by the application of a computational model with less stringent requirements for the consistency.

The most of the MIMD-processors are often use the weak consistency model of the access to the shared memory. It means that the read operation does not have to wait until all previous operations have finished the writings. This leads to the danger that the read data is outdated. But there are important classes of applications, such as solving large systems of equations, where this case leads to the correct final result.

In general, the weak consistency means that real consistency is not implemented at all times, but only at certain synchronization points, which are located by a programmer. Then the complex procedure of the consistent memory accesses is replaced by the ordered bypass of synchronization points. A common method of interrupting the successive program threads is the ***barrier synchronization***.

So, the programmer is responsible for putting the synchronization points into his program, ensuring the accuracy of the program execution. The use of weak consistency model to improve the efficiency of the program requires extensive knowledge of the programmer. The programmer must explicitly imagine the meaning of his actions and explicitly coordinate the

accesses to the shared resources. If the programmer can not divide the problem to subtasks without breaking the semantics of a sequential program, then he should require a consistent system performance, regardless of the losses.

5.3.2 Message passing model

The message passing model is simple for the implementation and effective for computing in the shared memory architectures. In the messaging paradigm, the parallel computing is realized at the level of the communicated processes. That the problem, solved with messaging, belongs to the coarse-grained parallelism. Under this paradigm, each PU of the parallel system has its own address space, and the parallel software modules can be large computing processes, threads or processes of medium size.

The programmer must ensure the exchanges between the processes by the inserting in the program of the programming constructions: “send”, “receive” and “reply” in accordance with the interprocessor exchange protocol. These constructions are the synchronization points to access the shared memory. But the manual solving the synchronization problem makes this model difficult to use. Just as in the shared memory model, the programmer is responsible for proper insertion of the task synchronization points in order to ensure the correct parallel computing.

Currently, there are common standard software interfaces for the message passing model. They are ***Parallel Virtual Machine*** (**PVM**) and ***Message Passing Interface*** (**MPI**). These interfaces are the library functions such as “send”, “receive” and “reply”, which are inserted in the parallel program in the synchronization points. These function calls are implemented at the operation system level and usually use the interprocessor communication protocol stack.

When using the message passing paradigm, the parallel programming is significantly simplified, if the solved problem allows the use of the SPMD (Single Process-Multiple data) model. In the SPMD model, a single sequential program code is duplicated in the processor nodes and is applied to different data sets, as in the SIMD architecture.

Thus, the SPMD combines in itself the global homogeneity of the system with the local automatic computing processes on different data. In the SPMD mode, each process, running under the MPI control, cyclically goes through three successive phases: 1) a new data portion is received from other processes, including those from other PUs, 2) calculating new results, 3) sending the results to other processes.

There are important classes of problems that can be solved in a simple and straight-forward SPMD model. A typical example is a problem, which has a schema in the form of a grid mesh. Such problems are linear algebra iterative methods for the modeling of fluid dynamics, weather forecast, strength test of constructions, etc.

5.3.3 Latent delay minimization in the MIMD systems

The critical problem of the large MIMD architecture is the latent delay of the data transfer in the interprocessor communications. If the processors are idle during this communication, the latent delay may become a serious obstacle that limits the performance of a computer system, because it is part of the time of the data preparation in the parallel algorithm acceleration formula (5.2).

There are two approaches to solving the latent delay problem. This delay can be either minimised or hidden. The latent delay minimizing means the shortening the data transmission delays to the allowable value by the structural and technological methods. This is achieved as by minimizing the

interprocessor switching system delays, including the reduction of the message forming and optimizing the frequency of data exchange.

The ***latent delay hiding*** means the organizational scheme, in which the processors perform the useful work during the data transfers. By such a system behavior, the system performance is reduced, due to the data transfer delays is not felt by the system users.

The usual approach to minimize the frequency of data exchanges is equipment of PUs by the additional amount of cache memory. This memory stores the data, which need to be frequently accessed if they were placed in the memory of other PUs. But this creates a problem of the cache memory consistency, which is considered in section 3.6.

The interprocessor communication system contains a hardware and software levels, which make different contributions to the latent delay. In the parallel systems with the physically separated memory, the latent delay is equal to a sum of the interprocessor communication delay and the software delay. The last is necessary for the implementation of a given memory access model.

In the message passing architecture, the hardware latency is equal to a delay of the message transfer from one processor to another one through the communication system. A software component delay is a delay of the operating system kernel, which is necessary for forming a header of the message, its transmission in source PU, and receiving in destination PU. Therefore, the latent delay minimizing requires the combined optimization of both hardware and system software.

5.3.4 Architecture with remote memory access and message passing

In the architecture with the distributed memory, two different methods for the interprocessor communications may be involved. The first one is that the PU node can have access not only to its memory but to the memory of other PUs. The second one is that PU is connected to other PUs using the messages, as shown above. So there are two different architectural categories: the architecture with the remote access to the memory and the message passing architecture.

Among the architectures with the remote memory access, the Distributed Shared Memory architecture (DSM) and the Multi-Threading Architecture (MTA) are distinguished. DSM and MTA architectures are distinguished in the different mode of the latent delay minimization. DSM uses the hardware delay minimization. In MTA, the latent delay hiding is performed due to the parallel execution of multiple software threads or branches. Both architectures use the medium-grained parallelism.

The message passing architectures and remote memory access architectures to its data exchange latency differ by one or two orders of magnitude (see. Fig. 5.5.).

The delay of a data exchange in the message passing architecture is determined by the duration of an object (message) transmission from the address space of one PU to the address space of another PU using the corresponding functions of the operational system. As already mentioned, this delay can be reduced both in hardware and in the program part and can be reduced through its hiding. But even at best, the latent delay in the message passing systems is relatively large. Consequently, the intervals of calculations between message transfers should also be significantly higher in order to

cover the cost of data transfers. Therefore, the messaging architectures should use the coarse-grained parallelism.

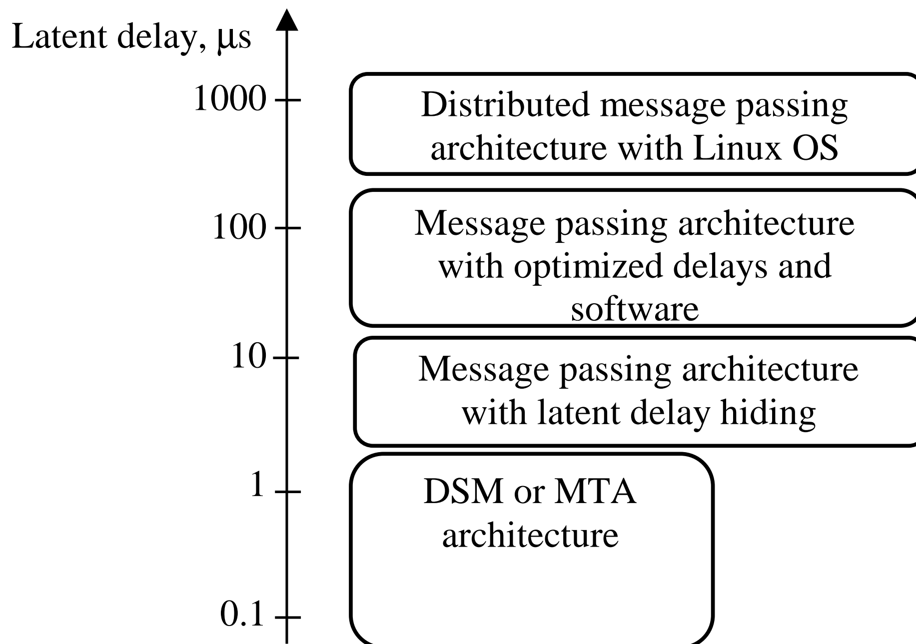

Fig. 5.5. Data transfer delays in different MIMD-architectures

5.3.5 DSM architecture

In the MIMD systems with the distributed shared memory, the total memory bandwidth grows proportionally to the number P of processor vertices. This makes the architecture scalable in general, but with certain restrictions, as is shown below. In order to understand, how the remote access is done in such a system, it should be noted that all memory blocks are placed in a single global address space. The global logical address is mapped into the corresponding physical address (<node number>, <cell address>) using the attached addressing.

The remote access is performed along the route, which passes through the interprocessor communications. As a result, the latent delay of the remote access is significantly more than one of the local access. Moreover, this delay

is proportional to the distance between the nodes of the system (see. Sec. 5.1). This delay dependence complicates the problem of the sequential consistency.

In the DSM architectures, the latent delay of the remote memory access is minimized through the installation of the secondary cache memory for each PU. Because of this, the global objects are in the cache lines of different PUs at a time. But this approach complicates the problem of cache coherency, which means that if there is a request to write a global object, then all PUs, which save a copy of the object, must clarify that these copies are invalid. Thus, each cache RAM should pass the signal of each writing access to all cache RAMs of other PUs. That is, each PU must pass the addresses of their write operations in the rest of PUs. As a result, we obtain a very large flow of messages.

In accordance with the coherence protocol of data cancellation, the write operation to cache RAM of PU must declare all the copies of this object to be invalid in the rest of PUs. Because of this, PU, which has made a writing to an object, sends an invalidation signal to all PUs, which store the object copies. The next read access to the invalid copy leads to a cache miss. Consequently, a copy of the modified object is read from the memory of PU, that has performed its modification. The information, on where to send these signals, is stored in a central directory. To get this information, PU, making a writing, has to send a request to this directory and receive a response. That is, the data about the accessed global objects are sent twice.

A bottleneck of the centralized directory method can be circumvented in a distributed scheme. Each PU in it stores a local directory with addresses of all copies of global objects. But this increases the memory volume of each PU. The required increase in the catalog volume is proportional to P^2. This factor is a significant constraint of increasing the number P of PUs in the system. Thus, the DSM concept is not suitable for the systems with the

massively parallel computing. And the system with thousands of PUs could not be realized as DSMA.

5.3.6 MTA Architecture

The principle of parallel software threads implies a large number of parallel branches in the algorithm for solving the problem. This principle is involved in the MTA architecture. It is worth mentioning that the program thread is the relatively independent computational process, such that OS provides to several related threads of a single application the shared resources. A thread may have the states of pending, standby and execution. The thread execution is usually controlled by the data flow. That is, a thread is suspended until it receives all needed operands, after that it falls in the standby state, and finally it is executed when the computing resources are available.

The MTA and DSM architectures have one thing in common — they both work in the global address space. The difference between them is that MTA does not need the high-cost and fast data transfer hardware, as the MTA architecture is based on the principle of the ***latent delay hiding***.

The request of the remote memory access automatically leads to the suspend of the thread execution, while this access is finished. At this time, the another thread, which was in a state of standby, goes into execution. So, time to spend on the remote access is used efficiently (i.e., it is "hidden"). This scheme leads to the intense context switching between the threads. Therefore, this method is effective only when the context switching delay is quite small. There are two ways to achieve the low costs for the context switching:

1) The algorithm of the thread is executed only from one remote access to another one, i.e., before the interrupt, the thread finishes its calculations. So, when the context is changed, the intermediate data need not be saved.

2) A set of register banks is built in PU. So, the contexts of several threads may be stored simultaneously in the respective register banks. In this case, the context switching consists only in exchange of the register bank pointer.

The first approach leads to a large number of small threads. The second approach enables large threads, but this requires the hardware support in the form of a set of register banks.

The thread execution under the data flow control requires the use of a synchronization system in order to meet the requirement: the thread execution begins only after it becomes available all its operands. The thread synchronizing is generally supported by the attachment of a flag "empty" — "full" to each memory cell of the input or result data, indicating whether the cell can be read ("full"), or written ("empty"). Thus, a set of flags indicates the moment of time when the thread can be performed in an environment of existing data dependencies.

5.3.7 Virtual shared memory architecture

I the Virtual Shared Memory Architecture (**VSMA**), each processor doesn't have the direct access to the memory of other processors, but only to its own memory. So, if the processor needs an access to the global object that is not in its memory, the object must firstly be rewritten in its local memory. This can be done through a mechanism, that is similar to the system behavior when the cache miss occurs. Therefore, a synonym of this architecture is the Cache-Only Memory Architecture (**COMA**).

In this architecture, if one copy is modified, then the rest of copies should be corrected. Therefore, the existence of a set of object copies in the system memory is a potential problem of the access consistency. Here an analogy with a multiprocessor with the cache blocks in its PU can be seen.

Then the memory of PU in VSMA acts as a cache memory and the common virtual memory serves as the shared memory. So, here the addressing memory mechanisms are similar to those, that arise when the cache addressing coherence is implemented.

If the addressed object is not in the PU memory, this PU sends a request to another PU, which sends this global object to that PU, where it is needed. So, this architecture is similar to the message passing architecture. In this case, each PU is equipped with additional storage to store the addresses of the global objects. But unlike DSM, in this architecture, the amount of additional memory increases linearly with the increase of the PU number. That is, the VSMA architecture is scalable as well as the message passing architecture is.

The following rules ensure the strict consistency implementation of accesses to the global objects.

1) An object can be read by any number of PUs in the random moment of time, and only one PU has the ability to writing him.

2) Before writing the global object, all other copies of it must be canceled.

This coherence model is called Multiple Readers — Single Writer (MRSW) model. Unlike the remote memory access architecture, the virtual address in the VSMA architecture does not specify where to find the object, i.e., there is no fixed place of its storage. But each object has its own host — the computational process that has permission to write in it. The program, which must refer to an object, is looking for it in the local memory of PU. If it's not there, that the cache miss happens, the memory management unit of PU should rewrite the object from another PU.

The compliance with strict consistency can lead to a phenomenon called "page threshing". A memory page begins to be "threshed" when two or more PUs perform writing to the same page and therefore, this page is sent

between those PUs too often. This situation can be done around by giving a permission for multiple writings, i.e., when the MRSW model is replaced to the Multiple Readers — Multiple Writers model (MRMW). But unlike the limited MRSW model, the MRMW model cannot prevent the emergence of the inconsistent copies of the object.

5.3.8 Problems

1) Name examples of the problems, which can be solved correctly in the MIMD system with the memory access consistency violations.

2) Consider the problem of the solid modeling by the finite particle method. The solid is represented by the 3-dimensional grid. One node of this grid is modeled by 100 PU instructions. A single modeling iteration consists of receiving data from neighboring nodes, performing mathematical computations and sending the results to the neighbors. Estimate, how many nodes are to be modeled in a single program thread, when the problem is solved in the MPI architecture.

3) The conditions as in the problem 2, but the grid is 2-dimensional.

4) The conditions as in the problem 2. Select the best MIMD topology, which is fitted for this problem solving and explain why.

5) The conditions as in the problem 2, but the MTA architecture is used.

6) Consider the processor operating in the MTA mode, which performs the same task, which is described in the problem 2. Estimate the optimum number of nodes, which are computed in a single thread in the conditions, when the data between threads are transferred through HDD. Note, that a single HDD access lasts 10^5 clock cycles.

5.4 Graphics accelerator architecture

The graphics accelerator (Graphic Processing Unit, **GPU**) was established in the nineties to accelerate the realistic scene building in a computer display screen. The initial data for this problem is the skeletons of the geometric bodies and the texture library. These textures should cover the spaces between the ribs of the skeleton on the basis of the light scattering effects, shadows, reflection and mutual places of the bodies. The solving such a problem requires intensive floating point calculations.

A decade ago, it became clear that the graphic accelerators can be used not only to accelerate the graphics and multimedia applications but also for the many tasks that require the intensive computing in the areas of linear algebra, cryptography, modeling, image processing and so on. Therefore, some graphics accelerator architectures became to be open and began to be accompanied by the appropriate means for the application design. Thus, the Nvidia company has proposed the programmer architectural platform Compute Unified Device Architecture (**CUDA**), which is widespread now in the world. The system for the heterogeneous architecture programming and the respective Open Computing Language (**OpenCL**) are designed for the mutual use for the application design based on GPUs of the companies Nvidia and AMD.

The basic unit of computing in the CUDA architecture is a thread. The thread represents both an elementary program and a group of data to it. When a thread is started, the flow processor resources, memory up to 256 registers, addresses generator for data vector reading from the external memory and the shared memory are assigned to it.

The instructions in the thread are RISC-type instructions. They implement the different operations with the register data, data exchanging

between registers and memory, instruction flow control including the conditional transfer instructions and synchronization.

When a task is formed, each graphical model element, for example, the skeleton polygon that appears on the screen, is mapped to a thread. As this model calculation is performed iteratively, i.e., frame by frame, the computation task periodically runs tens of thousands of threads on the GPU hardware.

To simplify the thread control, up to 32 threads are sampled in a group, called a warp. The ***warp*** is a basic program unit, which is a subject to planning and scheduling. The warp starts its execution in the Streaming Multiprocessor (**SM**) of GPU, which has from eight to 192 of PUs, the first level cache, and shared memory. Usually, all the threads in a warp have the same sequence of instructions. So, SM has the architecture of SPMD, although Nvidia company has called its architecture as Single Instruction — Multiple Threads (**SIMT**). Because the warp and thread contexts are stored in the register banks, the context switching has almost no delay. One SM executes in the sequential-parallel mode hundreds of threads that switch their context.

The threads in a warp can simultaneously select the data from different pages of the shared memory or from a single cell of this memory, When the data are stored in a single page, then the access to it is performed in a sequence. To align the start events of similar instructions in the parallel threads, usually, the barrier synchronization is executed. Thus, the threads from a single warp can easily communicate with each other. But the data transfer from one warp to another one can be implemented only through the external shared memory.

During the execution of the conditional instructions, the threads execute the instructions in both alternative branches to align the moments of starting the respective instructions in the parallel threads. The instruction in a thread can read the register immediately, but the access to the local memory

last ca. 10 – 22 clock cycles. The total volume of the outer DRAM can be equal up to several gigabytes. Although the data flow between the GPU chip and DRAM reaches hundreds of gigabytes per second and is pipelined, the latent delay of the DRAM access lasts from 400 to 800 clock cycles.

But the user does not feel such latent delays because they are hidden due to the fact that the warps are permanently switched by a dispatcher as in the MTA architecture. If the warp is waiting for the data, it is postponed and the warp is running, for which the data are ready.

Modern GPUs are equipped with thousands of PUs, which operate at a clock frequency of about one gigahertz. Due to its high peak throughput, that achieves teraflops, the most of the modern supercomputers are based on the graphic accelerators. They are frequently used for the linear algebra problem solving, simulation, cryptology. But the problem programming for such machines must be conscious, that GPU architecture is adapted to the problems with the natural parallelism and with the coarse-grained parallel tasks.

In an example, GPU performs the fast Fourier transform algorithm for 262k points for only 9 microseconds, which is a record. But the input-output overhead is equal to ca. 60,000 microseconds. Suppose P = 1024 PUs are used for this algorithm. Then, the duration of its execution on a single PU lasts T_1 = 9.216 ms, the preparation time is equal t_d = 60 ms. Then, the real algorithm acceleration due to the formula (5.2) is equal only S_P = 0.15, that is, it is better to execute this algorithm on a single processor.

GPU is worth to use for such algorithms, in which the number of computation operations is significantly greater than the amount of the shared memory accesses. In the previous example, the fast Fourier transform algorithm has a number of operations, other words, an algorithm complexity, which is estimated as $O(N\log_2 N)$. So, it would be better to select the algorithms like solving the equation system, which has the complexity $O(N^3)$.

6 APPLICATION SPECIFIC PROCESSORS

6.1 Introduction

The application specific processors are designed and manufactured when the extreme parameters are necessary to obtain at the cost of limiting a set of the implemented algorithms. These parameters are low cost, low power consumption, high performance, high reliability, low weight, dimensions and so on. Moreover, usually, only one or a very limited set these parameters reach the effective values in these processors. For example, it is impossible to imagine a high-performance computer with small dimensions and price. I.e., the application specific processors obey the principle of balancing the versatility and specialization.

The first electronic computer by J. Atanasov was specialized, as it has executed only one algorithm. The same computer was Colossus. Eniac was an application specific processor, which was adapted to the simple dataflow algorithms. But such universal computers like Univac or MECM, due to a significant reduction in performance, compared to the above computers, could perform a large set of algorithms. All military, aerospace computers were always specialized because special requirements of reliability, weight and size parameters are assigned to them.

Now the specialized computers are basic computing, control, and service units. So to speak, they are the workhorses of the transport, communication, and infrastructure, which are serving the humankind. Now a universal computer accounts for tens and hundreds of different application specific processors in the smart cards, vending machines, robotic complexes, home appliances, communications and navigation, monitoring and control systems. Because the supercomputers can effectively perform only a limited set of

algorithms with extremely high throughput and they are often manufactured in only one instance, they can also be attributed to the application specific computers.

But for the application specific processor, the architecture concept has a slightly different meaning than for the universal processors. Through its various specialty, these processors typically do not have the same architecture. When a new application specific processor is developed, its architecture has some meaning only to the developer, if the algorithm is implemented in hardware. Otherwise, the architecture is common for a hardware developer and for its programmer, if the processor is programmable.

6.2 Microcontrollers

6.2.1 Microcontroller purposes

According to the classification, given in section 1.2, the microcontroller is considered as a single-chip microprocessor, which performs only one pre-prepared program. Since the microcontroller has the minimized amount of RAM, it is not able to perform the program compilations. Therefore, this program is usually prepared, compiled and simulated in the external universal computer. And the compiler, and the development tools, which are used for this, are called as ***cross-compiler*** and ***cross-tools***, respectively.

The first microprocessors, such as i4004, i8008 and i8080, were used as microcontrollers. Their appearance has enabled to significantly minimize the hardware costs of developing various devices for automation, control, measurement, and so on. Thus it was possible to create, build and upgrade these devices according to the uniform methodology. This methodology is based on programming, that many people understood, not on the creation of complex automata networks, which were usually applied before the

microcontroller success. For these reasons, the microcontrollers are still widely used and are involved in the future.

Microcontrollers are classified by the bit width of data, that are processed by them. There are 8-, 16- and 32-bit microcontrollers. This bit width determines not only speed but also the address space of the microcontroller and thus its potential to perform the complex algorithms.

Earlier, the microcontroller cost was determined by its complexity and bit width. Now, the chip integration scale is quite large. And the cost of the hardware peripherals, interfaces, and built-in memory blocks greatly exceed the cost of the processor core. This hardware contains millions of transistors.

Due to this, the microcontroller price is determined mainly by the cost of its packaging. Therefore, when a microcontroller application is designed, its cost consists, in general, of the built-in software cost, which, in turn, is determined by its architecture, implemented algorithm complexity and software reuse. Usually, the built-in software is named as a ***firmware***, because it depends on the hardware features of concrete microcontrollers, and could not be so flexible as usual software can.

Because of this, the main volume of the software for the microcontrollers is described by the C language, and not by the assembly language, as it was two-four decades ago. Moreover, to speed-up the application development and pushing it to the market, many of them are programmed in such high-level languages as Java and Python.

6.2.2 Microcontroller features

Although the microcontrollers inherently is a kind of microprocessors, they have a number of distinctive architectural differences that are looking on.

— The program memory is separated from the data memory. If the executable program is prepared in advance, it is usually written in the

program ROM. The instructions are read from it regardless of reading or writing the data, stored in RAM. But this factor does not increases the processor speed as in the Manchester architecture. It is explained by the fact, that the period of the access to ROM in the modern microcontrollers is an order of magnitude greater than the access time to RAM.

The instruction ROM is usually made by the flash technology. To speed-up this memory reading, several neighboring instructions are read simultaneously from it. Also, the instruction execution frequency is increased by using a cache memory of small volume. A separate RAM block is often used, to which the program is rewritten from ROM before the execution or during it in the overlay mode.

— The memory management unit is missing or is primitive. The memory management unit (MMU) is used in the microprocessors to implement the virtual memory and to protect it. If the program is a single one, then its placement in the memory is known at compile time. Respectively, it is needed not to be protected from other computing processes and therefore, MMU is not required for the microcontroller. But some microcontrollers have the primitive MMU to protect the selected memory segments. For example, MMU protects ROM from writing to it. It prohibits writing to some pages, which is important during the debugging process and to improve the reliability.

In the absence of MMU, the microcontrollers are unable to run the multitask operational systems, such as Linux. But for them, the simplified operating systems are developed, such as uCLinux, FreeRTOS, which are compact, and high-speed OSes. They allow the programmer to arrange the execution of multiple processes in parallel in the time-sliced or switched-on-event multitasking mode (see chapter 3.5).

Recent microcontrollers are programmed by such parallel programming languages like Java, Ada. This makes it necessary to introduce the virtual memory and complicated MMU.

— Optimized interrupt system. As noted in Section 1.8, the CPU speed is mainly determined by the speed of its interrupt system. The microcontrollers are working in real time, and so they use the architectural measures to improve the interrupt system.

The number of interrupt signal sources in the microcontrollers is usually not more than a few dozen. Therefore, the interrupt source decoding and its priority check are usually performed by the hardware. To speed up the context switching, several register banks are built in many microcontrollers, that are switched during an interruption. To reduce the time costs of the interrupt vectoring and service, the small interrupt subroutines are placed directly in the interrupt vector table, such as in the i8051 microcontroller.

The usual processor, when the interrupt request comes while executing another interrupt, finishes the last one, changes the context to the context of the main program, then performs the service of a new interrupt with a new context switching. In the microcontroller with the discipline of the ***nested vectored interrupts***, such as ARM Cortex-M, in such a situation, the context of one interruption is immediately changed to the context of a new interruption. Due to this, the context switching overhead is reduced from 42 to six clock cycles.

— The instruction pipeline or its length is minimized. The microcontroller architecture design is a search for compromise between the performance and hardware costs, ease of use, power consumption, price. This often gives priority to the parameters, that are listed as the last ones in this row of parameters. The most of the microcontrollers are built on the Manchester architecture, in which each instruction is executed consistently for several

clock cycles. This provides several advantages, such as easy programming, simple implementation of the interrupt system, no problems with the memory access consistency, ability to calculate the program delays in advance.

If the microcontroller is built on the RISC architecture, the length of its instruction pipeline is usually limited by three or five stages. In this architecture, the pipeline flushing and interruptions overheads are acceptable to process the data in real time. But because of this architectural feature, the clock frequency is equal only to a few tens of megahertz, and only in some models, it reaches one gigahertz.

— Specific instructions in the instruction set. Each microcontroller has a certain degree of specialization. To perform certain functions effectively, the specific instructions are usually added to a universal set of instructions, which speed-up the calculation of the specific functions.

The most of the specific instructions are ones, that simplify the manipulation of individual bits in words. Thus, the read-modify-write instruction reads a bit, such as a bit in the chip pin, and changes its value. The microcontrollers, designed for solving the telecommunication problems, have an instruction, that performs the iteration of the code polynomial convolution when calculating the Cyclic Redundancy Check (**CRC**) code. Many specialized instructions are used in the DSP microprocessors, which are discussed below.

— A set of interfaces, that are specific to the microcontrollers. The microcontroller has to communicate in a uniform manner to various external devices, from which it gets the information, and which it controls. Here are the main standard interfaces, that are used for this purpose.

General Purpose Input-Output (**GPIO**) interface is an interface, which is inherent in all microcontrollers. It consists of any number of parallel inputs-outputs. Its signals have the standard logic level and are tolerant to arbitrary

external peripherals such as memory, the analog-to-digital converter (**ADC**), and more.

Universal Asynchronous Receiver-Transceiver (**UART**) is analogous to the COM interface and respond to the RS-232 or RS-485 standard interfaces. It is designed for the data transfer at a distance of several hundred meters at a speed of up to 115 thousand baud.

Serial Peripheral Interface (**SPI**) is a popular synchronous serial interface with four signals. It has one master and a few slaves, it is quite fast (up to 12 – 100 Mbaud), but it operates at short distances (typically up to a meter).

Joint Test Action Group (**JTAG**) interface responds to the IEEE 1149.1 Boundary-Scan architecture standard. By its action, this interface is similar to SPI. The majority of VLSI chips are equipped with the JTAG interface, including microcontrollers. The interface is designed for the verification of the chips in the fab, testing the chips assembled on boards, programming the internal memory, attaching the debug equipment.

Inter-Integrated Circuit (**I2C**) interface is a simple two-wire serial interface based on the bus with the open collector/source drivers. It may have several masters and an arbiter. The interface bandwidth is equal to 100 or 400 kbit/s. The external nonvolatile memory, external device configuration registers are usually accessed through this interface.

Recently, the universal and high-speed interfaces like USB and Ethernet, as well as the display interfaces like DVI are added to the microcontrollers.

— A wide range of peripherals. While the microcontrollers are somewhat specialized computer tools, their manufacturing has always aimed to meet the maximum range of users with different requirements. For this purpose, the microcontrollers of the first generation were released as families,

numbering hundreds of varieties, which differ among themselves due to the internal memory capacity and a set of peripherals.

Now, the cost of hardware peripherals is taken into account at least. And the most of the microcontrollers has the redundant set of peripherals. Therefore, according to statistics, more than a half of the microcontroller chip equipment is not used at all in the real applications.

For example, let us list a set of peripheral units, that are part of the microcontroller ARM Cortex-M. They are system timer, general purpose timers, real time clock, multi-channel DMA, interfaces GPIO, USB, UART, I2C, Ethernet, JTAG, multi-input 12-bit ADC, digital-to-analog converter (**DAC**), electric motor control unit.

The separate safety timer, named as a ***watchdog,*** is involved in many microcontrollers to prevent their hang-up. During the microcontroller operation, its program periodically reloads this timer. If the program hangs up, then this timer is not reloaded. And after a certain period of time (about a second) it causes a timer overflow interrupt, which forces a resetting of the microcontroller.

— Programmable power consumption. Most microcontrollers are used in devices that are critical to the energy consumption. Therefore, they have the program option to reduce the energy consumption in several ways. For this, the clock frequency is programmable, corresponding to the optimum ratio: speed/power. In the energy saving mode, the microcontroller enters a "sleep" state, in which the voltage is disconnected from the most of the units. Only the interrupt circuit is powered in this mode, which is designed with a minimum supply current. When a “wake” signal causes an interrupt, then the microcontroller is switched to the normal mode.

— Many microcontroller architectures are modernized microprocessor architectures. There are many applications of microcontrollers, that

require large amounts of the software. This, for example, the applications, which perform access to the standard file system, Ethernet. In this case, it is useful to involve the software, which was designed for the microprocessors with a common platform.

Because of this, many microcontrollers have the common microprocessor architectures, such as i80x86, MIPS, ARM, PowerPC, SPARC. Besides, the software development for these microcontrollers is much easier, since there is the widespread and comfortable ***ecosystem*** (a set of libraries, programming and debugging tools, maintenance, education, FAQ systems, etc.) for them. There are many developers with the relevant programming experience as well.

6.3 DSP microprocessors

It is difficult to name the science and technology, which has not involve the Digital Signal Processing (**DSP**). DSP is used in transport, electrical engineering, telecommunications, measuring and household appliances the most actively. This involves the mass production and a wide range of DSP tools.

The **DSP *microprocessor*** is a kind of microcontroller, which is adapted to implement the DSP algorithms.

The digital filters have a number of indisputable advantages over analog filters like the high quality of the magnitude-frequency characteristics, no manual adjustment, high characteristic stability, low noise level. The finite impulse response (**FIR**) filter algorithm can be represented as follows:

$$y(n) = \sum_{r=0}^{M} b_r x(n-r) \tag{6.1}$$

where b_r is the r-th filter coefficient, $x(n–r)$ is the n-th sample of the input signal delayed for r cycles, $y(n)$ is the resulting sample. Thus, the filter

calculation is the repeated iteration which consists of a multiplication by a factor, and adding the product to the result accumulator. Most DSP algorithms can be performed using a multiplication by a factor and addition to the accumulator: $s = ax + y$.

The microcontroller Intel i2920, which has appeared in 1979, has input ADC, output DAC, ALU, small RAM and primitive software control unit. The multiply by a factor, as in the formula (6.1), is carried out in it through a series of shifts and additions of a multiplicand. The appearance of this micro-controller had a striking effect. It was the first fact, that only a single chip could replace the analog filter with tens of amplifiers and large unstable capacitors.

Since the beginning of the eighties, the hardware multiplication blocks were implemented in VLSI circuits. Since then, the attempts to save the multiplications in the algorithms is stopped, and the operation $s = ax + y$ has become a major operation in the microcontrollers, which are adapted to DSP. These microcontrollers have formed a set of DSP microprocessors. TI TMS32010 microprocessor, which has appeared in 1992, is considered the first generation DSP microprocessor.

The pipelined processing is organized in the multiply-and-accumulate unit of the signal microprocessor for the maximizing its bandwidth. For this purpose, the microprocessors are equipped with many port memory blocks and the address sequence generators to read-write the operands. Thus, one special instruction, which runs in a loop, calculates the assignment $s = ax + y$, and increments the address counters, checking them to prevent them from going beyond the limits.

Also through the sum of products the basic operation of the fast Fourier transform (FFT) algorithm is fulfilled. For its support, a special instruction of the bit-reversal addressing is used.

The high-speed processing is usually supported by the software pipelining technique, which is considered in section 3.1.4. Through the pipelined computations and the use of special instructions, signal filtering and FFT is performed in the DSP microprocessors with great performance and minimized ALU stalls (about 10 – 40%). For its architectural features, the DSP microprocessor reminiscents the vector-pipelined processor, described in section 3.2.

For three decades, four signal generation microprocessors have changed. New microprocessors have the architecture which is still with separate instruction RAM and data RAM. Its data ALU performs the operation $s = ax + y$ in the pipelined mode. The changes occurred only in increasing the clock frequency (in 10 – 100 times), the volume of internal memory (in 100 times), expanding the range of peripherals (including ADC, DAC), decreasing prices. DSP microprocessors of series TI TMS3205x, AD ADSP21xx, Motorola/Freescale/NXP DSP56xx are widely spread because they do not change their architecture for decades. Therefore, they are considered as the microprocessor platforms.

The DSP microprocessors with multiple identical processing units of the architecture MIMD appeared as well. The trend is to spread the DSP microprocessors with the VLIW-architecture.

The first signal microprocessors had the 16-bit architecture. The high-quality sound processing and vocoder construction have pledged to implement the 24-bit, and then 32-bit DSP microprocessors. In addition, the floating point DSP microprocessors are common now to permit the sophisticated DSP algorithms with small errors.

For this purpose, the RISC-microprocessors came successfully, for which the 32-bit width is natural and which are pipelined. The most popular platform for this DSP is the ARM architecture. Also, the instruction that

calculates $s = ax + y$ with the automatic address increment has been added to it. In addition, the ARM platform provides the attachment of the specialized processors, that can perform effectively separate DSP algorithms like FFT, signal and image compression, data encryption. Something about such an architecture is described in the next section.

The signal microprocessors gradually lose its independent value. The 32-bit microcontrollers take their functions increasingly. They also become the components of the systems on the chip.

6.4 Processors with hardware control. Configurable computers

6.4.1 Field programmable gate arrays

It is obviously that any computing task is best performed using ASIC, which is custom designed for it. But the development of a new ASIC for the arbitrary task is economically inefficient and burdensome. Therefore, FPGAs which were used for the ASIC prototyping, began to be used as a programmable computing environment, that can withstand any amount of reprogrammings (see. Sec. 1.2). This, in turn, has accelerated the development of the FPGA industry.

In the last two decades, there is a phenomenon of a sharp increase in the number of programmable gates in FPGA, their clock frequency, and trace capacity. This led to a sharp increase in computing power of FPGA (Fig. 6.1).

Modern FPGA contains hundreds of thousands of configurable logic cells (LCs), thousands of multiplication blocks and thousands of memory blocks that ensure peak performance of more than a trillion operations per second. High-capacity FPGAs are widely used in telecommunications, to compute the multimedia data streams, solving the problems of bioinformatics,

process the databases, etc. These calculations are performed typically in the pipeline mode at the high frequency data streams.

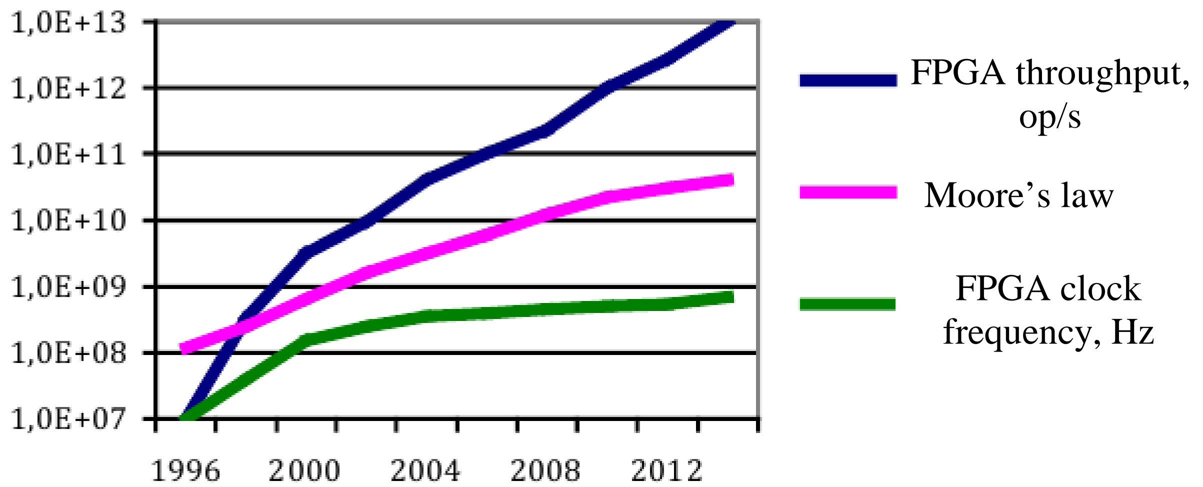

Fig.6.1. Growth of the FPGA potential performance

But FPGAs as a medium to accelerate the parallel computations in computers are used still rarely. This is due to the fact that FPGAs have never seen as a computing environment that can be programmed by the usual programmer. It has no effective programming language, in which not a specialist could program his tasks. Thus, the main obstacle for this is that FPGA is programmed by the programmable hardware design specialists who possess such languages as VHDL or Verilog instead of conventional programmers. On the other hand, the development of FPGA programs, i.e., its bitstreams is quite long because the compiler-synthesizer for it is very slow. Moreover, current development technology is focused on the description of the hardware at the register transfer level.

6.4.2 Configurable computer architecture

The configurable computer architecture is considered to be an alternative to advanced architectures. The ***configurable computer*** is such

that changes its structure during its operation at the logical level and the register transfer level.

The most common model of a configurable computer is a structure consisting of a general-purpose processor, which is connected to the computer, which is configured to accelerate the computations (Fig.6.2).

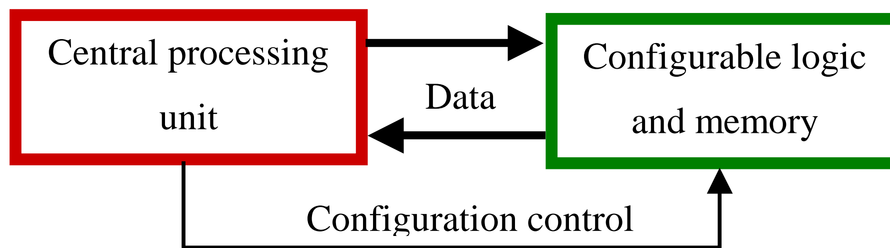

Fig. 6.2. The overall structure of the configurable computer

The opportunity to configure the connections between the LCs in FPGA is paid by the significant overheads. Therefore, the clock frequency of FPGA is in 3 to 10 times lower than it is in the microprocessors (see Fig.6.1). One transistor, which implements the equivalent gate in FPGA, accounts for up to 40 transistors of the infrastructure, that ensures the configuration. The infrastructure consists of configuration RAM, interconnection switches, nets for the clock and reset signal propagation, configurable clock management units, programmable multiplexers, communication transceivers, etc. So, in terms of rational use of transistors, FPGAs are much less effective than the microprocessors.

Although FPGA provides a high productivity increase and reduced energy consumption, its internal structure is irrational, and this should be considered. This fact is called the configurable computer paradox.

According to the von Neumann paradigm, (see. Sec. 1.3), the algorithm of the sequential nature is stored in program RAM, and the instructions are read sequentially in the instruction register except the branch instructions. As

a result of following this paradigm, the computers have a small performance, which is limited by the flow of instructions from RAM. This restriction is critical because the ratio of the on-chip memory speed to the external memory speed increases by 50% every year. Furthermore, even if the data is stored separately from the program as in the Harvard architecture, the overheads are needed for calculating the data addresses, performing data movings between ALU and outer memory. As a result, in the von Neumann computer, the delay shares to instruction fetching, its decoding, data reading and writing are accounted for respectively 33%, 10%, and 32% of the total instruction execution delay. But the ALU delay is only 5% of the total instruction delay.

The energy consumption is defined in discrete circuits as the energy dissipation in the conductor networks by the capacity recharging and therefore, it is proportional to the clock frequency and average switching activity (1.1). The other hand, the network delays are proportional to the capacitance of their wires and transistor gates. Thus, according to the delays of control and information signals in the von Neumann architecture, at least 95% of all switching circuits in it are irrational and are subject of minimization.

The effective solution is to embed in FPGA a microprocessor core, which has an instruction set, that expands, as such, ARM or MIPS instruction sets. Then the special instructions can be added to the instruction set, which perform the computing intensive operations by processing the data streams.

Instead of execution of an instruction sequence, which calculates the data addresses and counters, the parallel hardware sequencer units can be used in the advanced architecture. Also, data may be processed using the pipelined ALU using the principle of the pipeline concatenation. Then, both the data movings with the outer memory and ALU stalls can be minimized. (see Sec. 3.2).

The data flow processor can be effectively implemented in the configurable processors. This architecture paradigm was firstly implemented in the Dennis computer three decades ago. But in that time, the scalar parallelism was utilized, i.e., the operation granularity was too fine. And this led to the complex and slow networks of the processor.

But at present, the computed data are the arrays of different volume, which are computed by complex algorithms like matrix operations, data compression, feature extraction, etc., i.e., here is a middle- and coarse-grained computations.

6.4.3 Configurable processor in a multichip module

Due to the processor structure, illustrated by Fig. 6.2, it is obvious, that the configurable part of the system has to be placed as near as possible to the CPU part. In this situation, the throughput of the lines, connecting both parts, is the highest.

This strategy is taken into account by the Intel company, which introduced a MultiChip Module (**MCM**), which contains both Xeon microprocessor and Stratix-X FPGA. The chips of the microprocessor and FPGA are connected through an Embedded Multi-die Interconnect Bridge (**EMIB**) in a single MCM, which provide both high-speed data interchange (2 Gbits/s per via) and small energy consumption (ca. 1.2 pJ/bit). Further, the DRAM chips are attached to the system through the EMIBs.

CPU is programmed to execute both software and hardware functions. If a hardware function call occurs, then CPU starts this function in FPGA. Then the respectively configured module in FPGA reads the data from the shared memory, processes them and writes back the results, sending to CPU the interrupt signal about the function finishing.

The system programming is implemented using the library OpenCL of parallel computations. I.e., the hardware functions are prepared, in general, manually and collected to the library.

6.4.4 Cluster configurable processor

Xilinx company has proposed the configurable processor as a cluster, which is allocated in a cloud like Amazon. The FPGAs in the cluster are connected to each other and with the cloud processors through the PCIe and Ethernet interfaces. Here, the configurable processor is considered as a powerful resource, which is shared among the virtual processors-customers of the cloud.

The programming of the configurable processor is implemented using the High-Level Synthesis (**HLS**) tools. The input languages of the HLS compiler are C and C++. Therefore, the configurable processor can be programmed by the usual programmers — the cloud customers.

Before the hardware function execution, all the needed data are loaded into the configurable processor. After the task finishing, the system writes the results from this processor back.

6.4.5 Heterogeneous system architecture

The Heterogeneous System Architecture (**HSA**) is the standard architecture platform, which is supported by many companies and universities and by the HSA Foundation consortium. The goal of this consortium is to propagate the methods of simple programming of the heterogeneous computer systems. HSA gives to the system designers the possibility to easy and effectively use the specific hardware resources including CPU, GPU, DSP, and FPGA.

The HSA requirements are:

— security and quality of service to provide the robust system operation and be immune to the malicious and bad software;

— shared virtual memory with the respective MMU;

— atom operations at the platform level, which provides the memory assess consistency;

— the data types are signals, which contain both executed data and synchronization functions;

— user queues and dispatching them;

— coherent cache memories.

To put in operation the configured hardware function, the calling process sends to FPGA the function number and the reference address of the data in the shared memory of the system. Then the respective module in FPGA loads the needed data from the shared RAM using their cashing and computes them, storing the results to the shared RAM.

This order provides easy to understand and effective connection of the parallel processes which are executed both in CPU and in FPGA. The disadvantage of this architecture is the large overheads to arrange MMUs and cache coherency.

6.5 System on chip in FPGA

System on a Chip (**SoC**) is commonly referred us as a chip, which integrates the main functional elements of a whole hardware product. SoC usually contains in itself a processor, memory, peripherals, and application specific modules. Unlike SoC, manufactured as ASIC or ASSP, SoC in FPGA can be modified by the user through the partial or complete reconfiguration.

The processor provides basic computing and control functions in SoC. In addition, it initializes the peripherals, application specific modules and manages the memory.

The most common processors are used in Xilinx and Altera FPGA because these companies control the bulk of the market of programmable chips. Altera and Xilinx offer configurable processors Nios II and MicroBlaze, respectively. Both companies produce FPGAs with the built-in two-core system with the ARM Cortex-A9 architecture. It is also possible to use the license-free configurable processors, such as architecture LEON SPARC V8, Lattice Mico-32, and others. These processors have 32-bit RISC arhitectures with the 3 – 7 staged instruction pipeline.

The FPGA contains up to thousands of memory blocks with a capacity of several kilobytes. Since they are two-port blocks, they are effectively used to build the cache memory and FIFO buffers, which arrange the data stream transmissions between application specific modules, peripherals, and processors. The external dynamic memory and flash memory are used for applications that require the large amounts of memory. To ensure the high bandwidth of the dynamic memory interface, the appropriate memory control units are built in FPGA.

Application specific units in SoC are those units that perform certain functions with high speed. They usually have the pipelined structure and intended for processing the data streams. For the great performance, the bandwidth of these units should meet the bandwidth of the interfaces that connect them together. To do this, often these units are connected via FIFO buffers based on the two-port memory blocks.

So, the SoC efficiency significantly depends on the internal switching system, i.e., on the communication infrastructure, which construction is selected depending on the nature of the algorithms. In a typical SoC, this

infrastructure is fixed during its production and should be multi purposed. FPGA enables flexible programming of the infrastructure according to the nature of the application.

For example, consider the architecture of Xilinx Zynq-7000 FPGA, which is intended to implement the SoC, and which is widespread now. Its other name is All Programmable SoC says that this FPGA is a platform, which is optimized for the implementation of a broad class of the application specific systems, that it is possible to integrate to it both the processor with the configured modules and the equipment for DSP, telecommunications, various interfaces.

This FPGA incorporates the dual-core ARM Cortex-A9 with the cache RAM and MMU, with a wide range of embedded peripherals, multiple high-speed serial interfaces that are able to be configured in PCI-E, SATA, Gigabit Ethernet, HDMI interfaces, as well as two multi-input ADCs. The processor cores are connected to the internal memory controller, flash memory, external dynamic RAM and numerous peripherals via Advanced eXtensible Interface (AXI), which is adapted to the high-speed applications in SoC.

This Zynq platform is focused on the applications, where not only speed, low power consumption but also reliability and security are important. To protect the information, after power on, OS kernel is initially loaded in the processors. Then the configuration is loaded in FPGA. In this case, the operating system kernel and FPGA firmware can be encrypted using the standards of RSA, AES-256 and authentication protocol HMAC. This prevents any attempt of intrusion Trojan's software in SoC. Also for the information security, SoC uses the hardware encryption accelerator implementing the algorithms AES and SHA.

To prevent the tampering attacks, SoC in FPGA uses the Anti-Tamper technology. The Security Monitor via ADC samples data from the internal

temperature sensors, pressure sensors, and other ones, and resets the security key if it determines the tamper situation. In addition, the ADC inputs can be connected to the external sensors that are able to respond to the abnormal situations.

For the SoC project development, the Zynq platform has an effective ecosystem, i.e., a set of software and hardware tools, libraries, operating systems, training programs, consulting centers and other measures that facilitate the use of the platform and its study.

It also includes a high-level synthesis system. This synthesis consists in describing the algorithms by languages such as C, C++ or SystemC, their compilation into the FPGA configuration and processor firmware.

7 COMPUTER ARCHITECTURE PROSPECTS

For seven decades of the computer architecture history, several generations of computers have changed. Each year, performance was increased by increasing their clock frequency and architecture modernizations. In addition, the varieties of architectures with a wide range of their productivity was growing. Extrapolating the graphics of the architecture performance growth we can to some extent predict their development in the future (Fig.7.1).

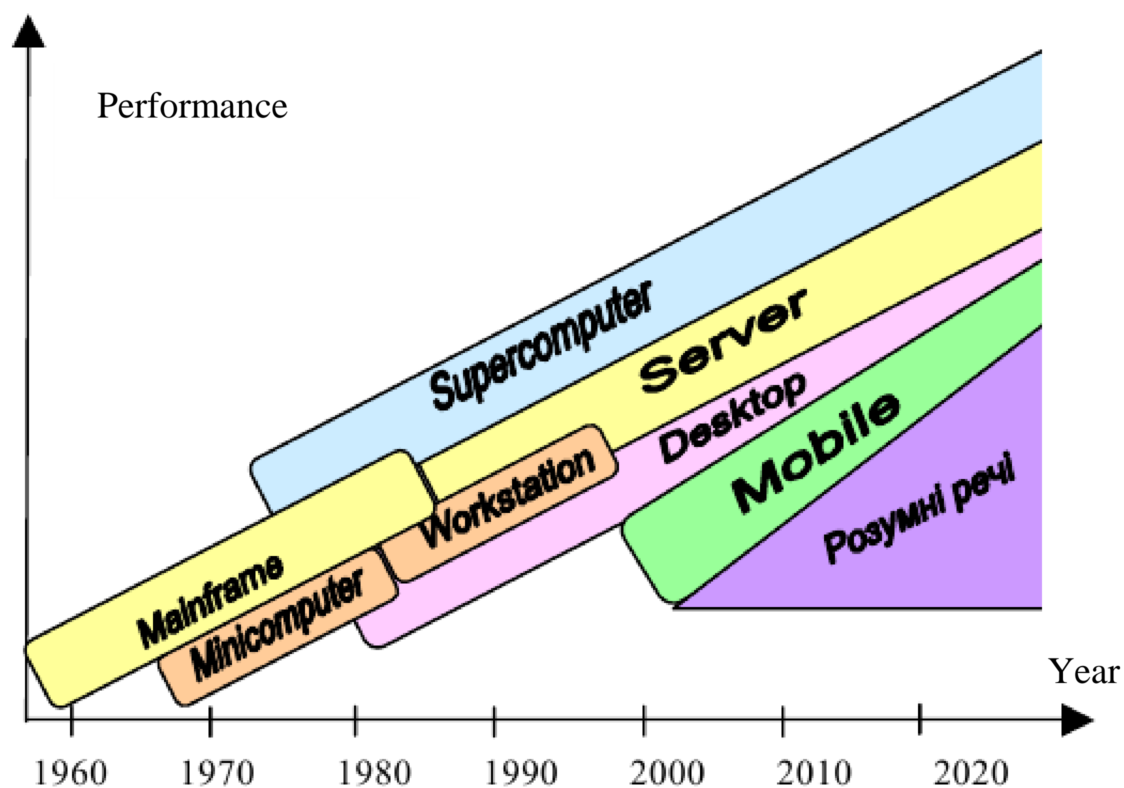

Fig. 7.1. Computer performance growth over time (in logarithmic scale)

At the beginning of the millennium, the significant changes in the evolution of the chip manufacturing technology, and the processor architectures based on them have taken place. Here are these changes.

The integral circuits have reached the limits of growth of power consumption. The critical power consumption is equal to ca. 200 W per square centimeter of the chip surface. An urgent need to save the processor power consumption appeared in all possible ways, including effective architecture selection and program optimization.

The processor clock speed increase is stopped in the limit of about 5 GHz. The gate delays in the chips become smaller than the delays in the lines. To transfer the signal between remote units in a chip without its distortions, it must go through several registers and buffers. The cost of design of new processor chips is significantly increased because of the complexity of the technology with the design rules, which are less than 65 nm.

The increase of the processor interface bandwidth is slowed down with a further increase of the gate number and memory capacity. The high bandwidth is supported only by increasing the number of chip pins and the deep pipelining of the signal routes.

The speed increase of a single CPU due to the optimizing of its internal structure (microarchitecture) is exhausted. The last architecture improvements are the introduction of trace cache and multithreaded processing mode (multithreading, hyperthreading), and the branch prediction by the adaptive algorithms.

The rapid spread of Internet technology, which is accompanied by the increased channel bandwidth.

The emergence of new types of computers that are expanded dynamically, such as mobile computers and "smart things" (Fig. 7.1). The last ones are the various household, transportation, medical, technological devices. They contain the built-in specialized computer tools that have sufficiently high performance, machine intelligence elements, and capable of

transmitting data over the Internet. Therefore, this industrial sector becomes a name of ***Internet of Things*** (**IoT**).

Based on these changes, we can predict the next likely directions of the computer architecture evolution.

The microprocessor performance will grow by increasing the number of processor cores on a chip to hundreds of pieces, as well as through optimization of operation in the multithreaded mode. To prevent the chip overheating, the CPU cores will be dynamically switched off. For this purpose, the supply voltage and clock speed of individual cores will also be adaptively regulated. The cores will be specialized and adapted to perform a certain class of algorithms as well.

Meeting the conditions of Moore's law, i.e., doubling the number of transistors on a chip every two years will be realized both by reducing the design rules and through the introduction of 2,5D- and 3D-ICs. This is actually the introduction of the technology of the multichip modules, which are fully made of silicon. In some 2,5D-IC, the chips are mounted on a larger chip, as in miniature PCB, and in 3D-IC, these chips are placed one above the other.

Due to this, a microprocessor, its memory, and peripherals can be performed in a single IC. This allows a increased bandwidth of the interface between the processor and memory, and peripherals.

One of the chips in 2,5D- or 3D-IC will be performed as the optical transmitter. This will allow for 1-3 orders of magnitude increase of the interprocessor communication bandwidth in the MIMD-architecture with the messaging. The formation, reception, and routing of messages in this architecture will be carried out by a specialized hardware, not through the protocol stack subroutines. It is also a prerequisite to increasing the number of the processors in the servers and supercomputers.

Do not wait for the emergence and spread of a new processor architecture with the unique instruction set, even if it potentially prevail the famous architectures. Moreover, some architectural platforms will disappear that were common till now. So, the advanced architectures of the nineties Alpha, PARISC were already disappeared. Now the Itanium architecture, which characteristics are much better than that of the IA-32/64 architecture, is gradually disappearing. The reason for this is that, on the one hand, the income from the use of new microprocessor can not justify increasing the costs of its development in the modern integrated technology. On the other hand, the manufacturers of advanced constantly growing software through their overloading are no longer able to produce new software versions for several different platforms.

The exception is the ARM platform. Its 64-bit version will come in the spread in the computer servers. And its 32-bit versions are popular in the mobile devices by allowing the substantial energy savings.

The virtual machine paradigm will gain further spread. According to it, the program is created for a virtual processor with the instruction set, which corresponds to the problem that is solved. This, for example, the Java program, which is compiled and runs in the Java virtual machine. This virtual processor is modeled by a physical processor with the instruction set, that, for example, is optimized to minimize the power consumption. This conversion of one instruction set to another one can be performed in hardware, such as it is done in the trace cache.

The architectures are expanded, which are distributed geographically. These apply the GRID systems, Cloud services and the systems for collecting and processing information from the remote sensors (IoT). In developing and using such architectures, it is important to consider the long latent delays of the messages, that are propagated via the telecommunication channels.

Many applications that have proliferated in the last decade, such as electronic money, cloud services, intellectualization of vehicles (Advanced Driver Assistance Systems, or **ADAS**) and medical devices require the implementation of special security measures, data security, and reliability. In addition, with the increasing degree of integration of ICs, the reliability of chips began to decrease significantly. Therefore, in the new architectures, the hardware devices and hardware-software systems will be built-in to support both security and reliability increase. The architectural changes will also be introduced to prevent the piracy stealing and the intrusion of the malicious software.

LIST OF RECOMMENDED LITERATURE

1. Гук М.Ю. Аппаратные средства IBM PC. Энциклопедия. — 3-е изд. — СПб.: Питер. — 2006. — 1072 с.

2. Гук М., Юров В. Процессоры Pentium 4, Athlon и Duron. — СПб.:Питер. — 2001. — 512 с.

3. Корнеев В., Киселев А.В. Современные микропроцессоры. — 3-е изд., перераб. и доп. — ВС. — 2003. — 448 с.

4. Мельник А.О. Архітектура комп'ютера. — Луцьк: волинська обл. друк. — 2008. — 470с.

5. Ортега Дж. Введение в параллельные и векторные методы решения линейных систем. — М.: Мир. — 1991. — 365 с.

6. Организация ЭВМ. 5-е изд./ К. Хамахер и др. — СПб.:Питер. — 2003. — 848 с.

7. Процюк Р.О., Корнейчук В.И., Кузьменко П.В., Тарасенко В.П. Компьютерная схемотехника (краткий курс). — К.:Корнійчук. — 2006. — 433 с.

8. Самофалов К.Г. и др. Цифровые ЭВМ: Теория и проектирование — 3-е изд.- К.:ВШ. — 1989. — 424 с.

9. Столлингс В. Структурная организация и архитектура компьютерных систем, 5-е изд.: Пер. с англ. — ВИ. — 2002. — 896 с.

10. Таненбаум Э. Архитектура компьютера. — СПб.: Питер. — 2002. — 704 с.

11. Гуров В. В. Архитектура микропроцессоров. — Интуит. — 2016. — 328 с.

12. Bobda C. Introduction to Reconfigurable Computing Architectures, Algorithms, and Applications. — Springer. — 2007. — 359 p.

13. Bryant R.E., O'Hallaron D.R. Computer Systems. A Programmer's Perspective. — Prentice Hall. — 2011. — 1043 p.

14. El-Rewini H., Abd-El-Barr M. Advanced Computer Architecture and Parallel Processing. — USA, New Jersey, Canada: Wiley. — 2005. — 272 p.

15. Giloi W. K. Rechnerarchitektur / Springer-Lehrbuch. — Springer. — 1993. — 485 p.

16. Patterson D.A., Hennesy J.L. Computer Organization Design. The Hardware/ Software Interface. — Elsevier, Morgan Kaufmann Publ. — 2005. — 684 p.

17. Wolf W. Computers as Components. Principles of Embedded Computing System Design. — Elsevier Inc. , Morgan Kaufmann Publ. — 2008. — 507 p.

18. Yiu J. The Definitive Guide to the ARM Cortex-M3. — Elsevier Inc., Newnes. — 2007. — 359 p.

19. Tannenbaum A. S., Austin T. Structured Computer Organization. 6-th ed. — Pearson. — 2013. — 769 p.

20. Gonzalez A., Latorre A. F., Magklis G. Processor Microarchitecture: An Implementation Perspective. — Morgan & Claypool. — 2011. — 106 p.

ANNEX 1

I8051 instruction set

Data moving instructions

Instruction name	Assembly code	OPC	B	Operation
Moving from register(n = 0 — 7) in accumulator	MOV A, Rn	11101r r r	1	(A) = (Rn)
Moving from address in accumulator	MOV A, ad	11100101	2	(A) = (ad)
Moving byte from DATA (i = 0, 1) in accumulator	MOV A, @Ri	1110011 i	1	(A) = ((Ri))
Loading of a constant in accumulator	MOV A, #d	01110100	2	(A) = #d
Moving accumulator to register	MOV Rn, A	11111r r r	1	(Rn) = (A)
Moving byte with address в register	MOV Rn, ad	10101r r r	2	(Rn) = (ad)
Loading of a constant in register	MOV Rn, #d	01111r r r	2	(Rn) = #d
Moving accumulator to direct address	MOV ad, A	11110101	2	(ad) = (A)
Moving register to direct address	MOV ad, Rn	10001r r r	2	(ad) = (Rn)
Moving byte in direct address to direct address	MOV add, ads	10000101	3	(add) = (ads)
Moving byte from DATA to direct address	MOV ad, @Ri	1000011 i	2	(ad) = ((Ri))
Moving a constant to direct address	MOV ad, #d	01110101	3	(ad) = #d
Moving accumulator to DATA	MOV @Ri, A	1111011 i	1	((Ri)) = (A)
Moving byte in address to DATA	MOV @Ri, ad	0110011 i	2	((Ri)) = (ad)
Moving of a constant to DATA	MOV @Ri, #d	0111011 i	2	((Ri)) = #d
Loading in data pointer	MOV DPTR, #d16	10010000	3	(DPTR) = #d16
Moving in accumulator a byte from CODE	MOVC A, @A + DPTR	10010011	1	(A) = ((A) + (DPTR))
Moving in accumulator a byte from CODE	MOVC A, @A + PC	10000011	1	(PC) = (PC) + 1 (A) = ((A)+(PC))
Moving in accumulator a byte from XDATA	MOVX A, @Ri	1110001 i	1	(A) = ((Ri))
Moving in accumulator a byte from XDATA	MOVX A, @DPTR	11100000	1	(A) = ((DPTR))
Moving in XDATA from accumulator	MOVX @Ri, A	1111001 i	1	((Ri)) = (A)
Moving in XDATA from accumulator	MOVX @DPTR, A	11110000	1	((DPTR)) = (A)
Loading in stack	PUSH ad	11000000	2	(SP) = (SP) + 1 ((SP)) = (ad)
Fetching from stack	POP ad	11010000	2	(ad) = (SP) (SP) = (SP) – 1
Exchange accumulator from регістром	XCH A, Rn	11001rrr	1	(A) <—> (Rn)
Exchange accumulator from байтом with address	XCH A, ad	11000101	2	(A) <—> (ad)

Arithmetic instructions

Instruction name	Assembly code	OPC	B	Operation
Adding register to accumulator (n = 0 — 7)	ADD A, Rn	00101r r r	1	(A) = (A) + (Rn)
Adding byte with address to accumulator	ADD A, ad	00100101	2	(A) = (A) + (ad)
Adding byte from DATA to accumulator (i = 0, 1)	ADD A, @Ri	0010011 i	1	(A) = (A) + ((Ri))
Adding of a constant to accumulator	ADD A, #d	00100100	2	(A) = (A) + #d
Adding register and carry to accumulator	ADDC A, Rn	00111r r r	1	(A) = (A) + (Rn) + (C)
Adding byte with address and carry to accumulator	ADDC A, ad	00110101	2	(A) = (A) + (ad) + (C)
Adding byte from DATA and carry to accumulator	ADDC A, @Ri	0011011 i	1	(A) = (A) + ((Ri)) + (C)
Adding of a constant and carry to accumulator	ADDC A, #d	00110100	2	(A) = (A) + #d + (C)
Decimal correction of accumulator	DA A	11010100	1	If $A_{0—3}>9 \vee ((AC)=1)$, then $A_{0—3}=A_{0—3}+6$, $A_{4—7}>9 \vee ((C)=1)$, then $A_{4—7}=A_{4—7}+6$
Subtraction register and borrow from accumulator	SUBB A, Rn	10011r r r	1	(A) = (A) — (C) — (Rn)
Subtraction byte in address and borrow from accumulator	SUBB A, ad	10010101	2	(A) = (A) — (C) — ((ad))
Subtraction byte DATA and borrow from accumulator	SUBB A, @Ri	1001011 i	1	(A) = (A) — (C) — ((Ri))
Subtraction of a constant and borrow from accumulator	SUBB A, #d	10010100	2	(A) = (A) — (C) — #d
Incrementing accumulator	INC A	00000100	1	(A) = (A) + 1
Incrementing register	INC Rn	00001r r r	1	(Rn) = (Rn) + 1
Incrementing byte with address	INC ad	00000101	2	(ad) = (ad) + 1
Incrementing byte in DATA	INC @Ri	0000011i	1	((Ri)) = ((Ri)) +1
Incrementing data pointer	INC DPTR	10100011	1	(DPTR) = (DPTR) + 1
Decrementing accumulator	DEC A	00010100	1	(A) = (A) – 1
Decrementing register	DEC Rn	00011r r r	1	(Rn) = (Rn) — 1
Decrementing byte with address	DEC ad	00010101	2	(ad) = (ad) — 1
Decrementing byte in DATA	DEC @Ri	0001011 i	1	((Ri)) = ((Ri)) —1
Multiplication accumulator to register B	MUL AB	10100100	1	(B)(A) = (A)*(B)
Division accumulator to register B	DIV AB	10000100	1	(A).(B) = (A)/(B)

OPC – OPeration Code
B – instruction length, bytes

Logic instructions

Instruction name	Assembly code	OPC	B	Operation
Logic AND accumulator and register	ANL A, Rn	01011r r r	1	(A) = (A) ∧ (Rn)
Logic AND accumulator and byte with address	ANL A, ad	01010101	2	(A) = (A) ∧ (ad)
Logic AND accumulator and byte from DATA	ANL A, @Ri	0101011 i	1	(A) = (A) ∧ ((Ri))
Logic AND accumulator and constant	ANL A, #d	01010100	2	(A) = (A) ∧ #d
Logic AND byte with address accumulator	ANL ad, A	01010010	2	(ad) = (ad) ∧ (A)
Logic AND byte with address and a constant	ANL ad, #d	01010011	3	(ad) = (ad) ∧ #d
Logic OR accumulator and register	ORL A, Rn	01001r r r	1	(A) = (A) ∨ (Rn)
Logic OR accumulator and byte with address	ORL A, ad	01000101	2	(A) = (A) ∨ (ad)
Logic OR accumulator and byte from DATA	ORL A, @Ri	0100011 i	1	(A) = (A) ∨ ((Ri))
Logic OR accumulator and a constant	ORL A, #d	01000100	2	(A) = (A) ∨ #d
Logic OR byte with address and accumulator	ORL ad, A	01000010	2	(ad) = (ad) ∨ (A)
Logic OR byte with address and a constant	ORL ad, #d	01000011	3	(ad) = (ad) ∨ #d
Exclusive OR accumulator and register	XRL A, Rn	01101r r r	1	(A) = (A) ⊕ (Rn)
Exclusive OR accumulator and byte with address	XRL A, ad	01100101	2	(A) = (A) ⊕ (ad)
Exclusive OR accumulator and byte from DATA	XRL A, @Ri	0110011 i	1	(A) = (A) ⊕ ((Ri))
Exclusive OR accumulator and a constant	XRL A, #d	01100100	2	(A) = (A) ⊕ #d
Exclusive OR byte with address and accumulator	XRL ad, A	01100010	2	(ad) = (ad) ⊕ (A)
Exclusive OR byte with address and a constant	XRL ad, #d	01100011	3	(ad) = (ad) ⊕ #d
Resetting accumulator	CLR A	11100100	1	(A) = 0
Inversion of accumulator	CPL A	11110100	1	(A) = (NOT A)
Shifting accumulator left cyclically	RL A	00100011	1	(An+1) = (An), n = 0 ... 6, (A0) = (A7)
Shifting accumulator left through carry bit	RLC A	00110011	1	(An+1) = (An), n = 0... 6, (A0) = (C), (C)=(A7)
Shifting accumulator right cyclically	RR A	00000011	1	(A n) = (A n+1), n = 0...6, (A7) = (A0)
Shifting accumulator right through carry bit	RRC A	00010011	1	(A n)=(A n+1), n = 0...6, (A7)=(C), (C) = (A0)
Swapping nibbles in accumulator	SWAP A	11000100	1	(A0—3) <—> (A4—7)

Control flow instructions

Instruction name	Assembly code	OPC	B	Operation
Long jump in PRAM	LJMP ad16	00000010	3	(PC) = ad16
Absolute jump in range of 2 Kbytes	AJMP ad11	$a_{10}a_9a_8$00001	2	(PC) = (PC) + 2 (PC0—10) = ad11
Short relative jump in range of 256 bytes	SJMP rel	10000000	2	(PC) = (PC) + 2 (PC) = (PC) + rel
Indirect relative jump	JMP @A+DPTR	01110011	1	(PC) = (A) + (DPTR) (PC) = (PC) + 2,
Jump, if acc. – is zero	JZ rel	01100000	2	if (A) = 0, then (PC) = (PC) + rel
Jump, if acc. Is not zero	JNZ rel	01110000	2	(PC) = (PC) + 2, if (A) ? 0, then (PC) = (PC) + rel
Jump, if carry =1	JC rel	01000000	2	(PC) = (PC) + 2, if (C) = 1, then (PC) = (PC) + rel
Jump, if carry =0	JNC rel	01010000	2	(PC) = (PC) + 2, if (C) = 0, then (PC) = (PC) + rel
Jump, if bit =1	JB bit, rel	00100000	3	(PC) = (PC) + 3, if (b) = 1, then (PC) = (PC) + rel
Jump, if bit =0	JNB bit, rel	00110000	3	(PC) = (PC) + 3, if (b) = 0, then (PC) = (PC) + rel
Jump, if bit =1, then bit:= 0	JBC bit, rel	00010000	3	(PC) = (PC) + 3, if (b) = 1, then (b) = 0 and (PC) = (PC) + rel
Decrementing register and jump, if not 0	DJNZ Rn, rel	11011r r r	2	(PC) = (PC)+2, (Rn)=(Rn)—, if (Rn)≠0, then (PC)=(PC)+rel
Decrementing byte with address and jump, if not zero	DJNZ ad, rel	11010101	3	(PC) = (PC) + 2, (ad) = (ad) —1, if (ad) ? 0, then (PC) = (PC) + rel
Compare acc. with a byte with address and jump, if not equal	CJNE A, ad, rel	10110101	3	(PC) = (PC) + 3, if (A) ? (ad), then (PC) = (PC) + rel, if (A) < (ad), then (C) = 1, otherwise (C) = 0
Compare accumulator with a constant and jump, if not equal	CJNE A, #d, rel	10110100	3	(PC) = (PC) + 3, if (A) ? #d, then (PC) = (PC) + rel, if (A) < #d, then (C) = 1, otherwise (C) = 0
Compare register with a constant and jump, if not equal	CJNE Rn, #d, rel	10111r r r	3	(PC) = (PC) + 3, if (Rn) ? #d, then (PC) = (PC) + rel, if (Rn) < #d, then (C) = 1, otherwise (C) = 0
Compare byte in DATA with a constant and jump, if not equal	CJNE @Ri, #d, rel	1011011 i	3	(PC) = (PC) + 3, if ((Ri)) ? #d, then (PC) = (PC) + rel, if ((Ri)) < #d, then (C) = 1, otherwise (C) = 0
Long subprogram call	LCALL ad16	00010010	3	(PC) = (PC) + 3, (SP) = (SP) + 1, ((SP)) = ($PC_{0—7}$), (SP) = (SP) + 1, ((SP)) = ($PC_{8—15}$), (PC) = ad16
Absolute subprogram call in range 2 Kbytes	ACALL ad11	$a_{10}a_9a_8$10001	2	(PC)=(PC)+2, (SP)=(SP)+1, ((SP))=(PC0—7), (SP)=(SP)+1, ((SP)) = ($PC_{8—15}$), ($PC_{0—7}$) = ad11 ($PC_{8—15}$) = ((SP)),
Return from subprogram	RET	00100010	1	(SP) = (SP) − 1, (PC0−7) = ((SP)), (SP) = (SP)−1
Return from interrupt	RETI	00110010	1	($PC_{8—15}$)=((SP)), (SP)=(SP)−1, ($PC_{0—7}$) = ((SP)), (SP) =(SP)−1
No OPeration	NOP	00000000	1	(PC) = (PC) + 1

Bit handling instructions

Instruction name	Assembly code	OPC	B	Operation
Setting in 0 carry	CLR C	11000011	1	(C) = 0
Setting in 0 a bit	CLR bit	11000010	2	(b) = 0
Setting in 1 carry	SETB C	11010011	1	(C) = 1
Setting in 1 bit	SETB bit	11010010	2	(b) = 1
Complement of carry	CPL C	10110011	1	(C) = ($\overline{C}$)
Complement of bit	CPL bit	10110010	2	(b) = ($\overline{b}$)
Logic AND bit and carry	ANL C, bit	10000010	2	(C) = (C) ∧ (b)
Logic AND of negated bit and carry	ANL C, /bit	10110000	2	(C) = (C) ∧ ($\overline{b}$)
Logic OR of bit and carry	ORL C, bit	01110010	2	(C) = (C) ∨ (b)
Logic OR of negated bit and carry	ORL C, /bit	10100000	2	(C) = (C) ∨ ($\overline{b}$)
Moving bit to carry	MOV C, bit	10100010	2	(C) = (b)
Moving carry to bit	MOV bit, C	10010010	2	(b) = (C)