FACTA UNIVERSITATIS
Series: **Electronics and Energetics** Vol. 35, Nº 2, June 2022, pp. 155-186
https://doi.org/10.2298/FUEE2202155N

Review paper

FIFTY YEARS OF MICROPROCESSOR EVOLUTION: FROM SINGLE CPU TO MULTICORE AND MANYCORE SYSTEMS

Goran Nikolić, Bojan Dimitrijević, Tatjana Nikolić, Mile Stojčev

University of Niš, Faculty of Electronic Engineering, Niš, Serbia

Abstract. *Nowadays microprocessors are among the most complex electronic systems that man has ever designed. One small silicon chip can contain the complete processor, large memory and logic needed to connect it to the input-output devices. The performance of today's processors implemented on a single chip surpasses the performance of a room-sized supercomputer from just 50 years ago, which cost over $ 10 million [1]. Even the embedded processors found in everyday devices such as mobile phones are far more powerful than computer developers once imagined. The main components of a modern microprocessor are a number of general-purpose cores, a graphics processing unit, a shared cache, memory and input-output interface and a network on a chip to interconnect all these components [2]. The speed of the microprocessor is determined by its clock frequency and cannot exceed a certain limit. Namely, as the frequency increases, the power dissipation increases too, and consequently the amount of heating becomes critical. So, silicon manufacturers decided to design new processor architecture, called multicore processors [3]. With aim to increase performance and efficiency these multiple cores execute multiple instructions simultaneously. In this way, the amount of parallel computing or parallelism is increased [4]. In spite of mentioned advantages, numerous challenges must be addressed carefully when more cores and parallelism are used.*

This paper presents a review of microprocessor microarchitectures, discussing their generations over the past 50 years. Then, it describes the currently used implementations of the microarchitecture of modern microprocessors, pointing out the specifics of parallel computing in heterogeneous microprocessor systems. To use efficiently the possibility of multi-core technology, software applications must be multithreaded. The program execution must be distributed among the multi-core processors so they can operate simultaneously. To use multi-threading, it is imperative for programmer to understand the basic principles of parallel computing and parallel hardware. Finally, the paper provides details how to implement hardware parallelism in multicore systems.

Key words: *Microprocessor, Pipelining, Superscalar, Multicore, Multithreading*

Received April 13, 2022
Corresponding author: Goran Nikolić
University of Niš, Faculty of Electronic Engineering, 18106 Niš, Aleksandra Medvedeva 14, Serbia
E-mail: goran.nikolic@elfak.ni.ac.rs

1. INTRODUCTION

A microprocessor (processor implemented in a single chip) is one of the most inventive technological innovations in electronics since the discovery of the transistor in 1948. This amazing device has involved many innovations in the field of digital electronics, and became a part of everyday life of people. The microprocessor is the central processing unit (CPU) and it is an essential component of the computer [5]. Nowadays, it is a silicon chip that is composed from millions up to billions of transistors and other electronic components. The CPU can execute several hundred millions/billions of instructions per second. A microprocessor is preprogrammed to execute software in conjunction with memory and special-purpose chips. It accepts digital data as input and processes it according to the instructions stored in the memory [6]. The microprocessor performs numerous functions including data storage, interaction with input-output devices, time-critical execution and other. Applications of microprocessors range from very complex process controllers to simple devices and even toys. Therefore, it is necessary for every electronics engineer to have a solid knowledge of microprocessors.

This article discusses the types and 50 years' evolution period of microprocessor. The evolution of microprocessors throughout history has been turbulent. The first microprocessor called Intel 4004 was designed by Intel in 1971. It was composed of about 2,300 transistors, was clocked at 740 kHz and delivered 92,000 instructions per second while dissipating around 0.5 watts. After that, almost every year a new microprocessor, with significant performance improvements in respect to previous ones, was launched. The growth in performance was exponential, of the order of 50% per year, resulting in a cumulative growth of over three orders of magnitude over a two-decade period [7]. These improvements have been driven by advances in the semiconductor manufacturing process and innovations in processor architecture [8]. Multicore processing has posed new challenges for both hardware designers and application developers. Parallel applications place new demands on the processing system. Although a multicore architecture designed for a specific target problem gives excellent results, it should be borne in mind that the main goal in computer system design should be to provide the ability to efficiently handle different types of problems. However, a single architecture "one size fits all", which is able to effectively solve all challenges, has not been found so far, and many are convinced that it will never be [9].

This article presents a review of the microarchitecture of contemporary microprocessors. The discussion starts with 50 years of microprocessor history and its generations. Then, it describes the currently used microarchitecture implementations of modern microprocessors. At the end it points to specifics of parallel computing in heterogeneous microprocessor systems. This article is intended for an advanced course on computer architecture, suitable for graduate students or senior undergrads in computer and electrical engineering. It can be also useful for practitioners in the industry in the area of microprocessor design.

2. DEFINITION OF MICROPROCESSOR

Central Processing Unit, also known as a processor or microprocessor, is a controlling unit of a micro-computer inside a small chip. CPU is often referred to as the brain and heart of all computer (digital) systems and is responsible for doing all the work. It performs every single action a computer does and executes programs. In essence, the CPU is capable to perform Arithmetic Logical Unit (ALU) operations and communicates

with the other input/output devices and auxiliary storage units connected with it. In modern computers, the CPU is contained on an integrated circuit chip in which several functions are combined [10]. In general, all CPUs, single-chip microprocessors or multi-chip implementations run programs by performing the following steps:

1. Read an instruction and decode it
2. Find any associated data that is needed to process the instruction
3. Process the instruction
4. Write the results out

The instruction cycle is repeated continuously until the power is turned off.
A microprocessor is built using the following three basic circuit blocks [11]:

1. Registers,
2. ALU, and
3. Control Unit (CU).

Registers can exist in two forms, either as an array of static memory elements such as flip-flops, or as a portion of a Random Access Memory (RAM) which may be of the dynamic or static type. ALU usually provides, at the minimum, facilities for addition, subtraction, OR, AND, complementation, and shift operations. The CU of the CPU regulates and integrates computer operations. It selects and retrieves instructions from main memory in the appropriate order and interprets them to activate other functional building blocks of the system at the appropriate time with aim to perform its proper operations.

3. GENERATION AND MICROPROCESSOR HISTORY

On December 23rd, 1947, the *Transistor* was invented in Bell Laboratory, whereas an *Integrated Circuit* was invented in 1958 in Texas Instruments. In 1971 Intel or INTegrated ELectronics has invented the first *Microprocessor*. The evolution of CPU can be divided into five generations such as first, second, third, fourth, and fifth generation [12], and the characteristics of these generations will be discussed in the sequel.

1st Generation: The first-generation microprocessors were introduced in the year 1971-1972 when INTEL launched the first microprocessor 4004 running at a clock speed of 740 kHz. Other microprocessors that belong to this generation are Rockwell International PPS-4, INTEL-8008, and National Semiconductors IMP-16. Instruction processing of these CPUs was serial. Namely, instruction phases, fetch, decode and execution, were performed sequentially. When the current instruction was finished, then the CPU updates the instruction pointer and fetches the consecutive one in the program sequence, and so on for each instruction in turn.

***2nd Generation*:** This was the period from 1973 to 1978 in which very efficient 8-bit microprocessors were implemented like Motorola 6800 and 6801, INTEL-8085, and Zilog's-Z80, which were among the most popular ones. The second-generation of the microprocessor is characterized by overlapped fetch, decode, and execute phases. When the first instruction is processed in the execution unit, then the second instruction is decoded and the third instruction is fetched. Compared to the first-generation, the use of new semiconductor technologies for chip manufacture was a novelty in the second generation. Gains in innovation were a significant increase in instruction execution speed and chip densities.

***3rd Generation*:** The third-generation microprocessors were introduced in the year 1978, as denoted by Intel's 8086 and the Zilog Z8000. From 1979 to 1980, Intel 8086/80186/80286

and Motorola 68000 and 68010 were developed. Processors of this generation were 16-bit, four times faster than the previous generation, and with a performance like mini computers [13], [14], [15]. The development of a proprietary microprocessor architecture based on own Instruction Set Computer (ISC) was a novelty of this generation.

***4th Generation*:** Development of 32-bit microprocessors, during the period from 1981 up to 1995, characterizes the fourth-generation. Typical products were Intel-80386 and Motorola's 68020/68030. Microprocessors of this generation are characterized by higher chip density, even up to a million transistors. High-end microprocessors at the time, such as Motorola's 88100 and Intel's 80960CA, could issue and retrieve more than one instruction per clock cycle [16], [17].

***5th Generation*:** From 1995 until now, this generation has been characterized by 64-bit processors that have high performance and run at high speeds. Typical representatives are Pentium, Celeron, Dual and Quad-core processors that use superscalar processing, and their chip design exceeds 10 million transistors. The 64-bit processors became mainstream in the 2000s. Microprocessor speeds were limited by power dissipation. In order to avoid the implementation of expensive cooling systems, manufacturers were forced to use parallel computing in the form of the *multi-core* and *many-core processor*. Thus, the microprocessor has evolved through all these generations, and the fifth-generation microprocessors represent an advancement in specifications. Some of the processors from the fifth generation of processors with their specifications will be briefly discussed in the text that follows.

4. CLASSIFICATION OF PROCESSOR

Processor can be classified along several orthogonal dimensions. Here we will point briefly to some of the most commonly used. The first classification is based on microarchitecture specifics, second one to the market segment, the third on type of processing, e.tc. In this article, we will focus on the first classification scheme. For more details about this problematic the readers can consult Reference [10].

4.1. Classification of microarchitecture specifics

In general, we distinguish the following classifications:

4.1.1. Pipelined vs Non-Pipelined Processors

A Non-Pipelined processor executes only a single instruction at a given time. The start of the next instruction is delayed until the current ends, not based on hazards but unconditionally. The CPU scheduler chooses the instruction from the pool of waiting instructions, when it is free.

Pipelining is a technique where multiple instructions are overlapped during execution. The Pipelined processor is divided into several processing stages (segments). The stages are mutually connected in a form of a pipe structure. Constituents of each stage are an input register and a combinational circuit. The role of the register is to hold data and of combinational circuit to process it. The combinational circuit outputs processed data to the input register of the next segment.

The pipeline technique is divided into two categories:
a) Arithmetic Pipelines are mainly used for floating point operations, multiplication of fixed-point numbers, etc.;
b) In Instruction Pipeline instructions are executed by overlapping fetch, decode and execute phases. Pipeline technique increases Instruction Level Parallelism (ILP) and is used by all processors nowadays [18].

Table 1 Difference between Pipelining and Non-Pipelining Systems

Pipelining System	Non-Pipelining System
Multiple instructions are overlapped during execution	Phases fetching, decoding, execution and writing memory are merged into a single unit (step)
Several instructions are executed at the same time	Only one instruction is executed at the same time
The CPU scheduler design determines efficiency	The efficiency is not dependent on the CPU scheduler
Execution time is less (in a fewer cycle)	Execution takes more time (a greater number of cycles)

In addition to the fact that pipelining increases the overall system performance, there are several factors that cause conflicts and degrade performance. Among the most important factors are the following:
1. *Timing Variations* - The processing time of instructions is not the same, because different instructions may require different operands (constants, registers, memory). Accordingly, pipeline stages do not always consume the same amount of time.
2. *Data Hazards* - The problem arises when several instructions are partially executed in the pipeline system and in doing so, two or more of them refer to the same data. In that case, it must be ensured that the next instruction stall until the current instruction has finished processing that data, because otherwise an incorrect result will occur.
3. *Branching* - The next instruction is fetched during the execution of the current one. However, if the current instruction is conditional branching, then the next instruction will not be known until this current one completes data processing and determines the branching outcome.
4. *Interrupts* - Interrupts have an impact on the execution of instructions by inserting unwanted instructions into the current instruction stream.
5. *Data Dependency* - This problem occurs when the result of the previous instruction is not yet available, and it is already needed as data for the current instruction.

Main advantages of pipelining are higher clock frequency and increased the system throughput. However, there are disadvantages of this technique, primarily the greater complexity of the design and the increased latency of the instruction.

4.1.2. In-Order vs Out-of-Order Processors

A processor that executes instructions sequentially usually uses resources inefficiently, resulting in poor performance. Two approaches can be used to improve processor performance. The first one deals with simultaneous executing different sub-steps of consecutive instructions or even executing instructions completely simultaneously. The second one refers to out-of-order instruction execution which can be achieved by executing the instruction in a different order from the original one [1], [19]. Instructions order is determined by the compiler, but it is not necessary to execute them in that order. They may

be: a) issued in order and completed in order; b) issued in order, but completed out of order; c) issued out of order, but completed in order; and d) issued out of order and completed out of order [1].

First- and second-generation microprocessors process instructions in order. In-order processor performs the following steps:

1. Retrieves instructions from the program memory.
2. If input operands are available in the register file, it sends command to the execution unit in order to execute instruction.
3. If during the current clock cycle input operands are not available, the processor will wait for them. This case occurs when the processor retrieves data from slow memory. This implies that instructions are statically scheduled.
4. The instruction is then executed by the appropriate execution unit.
5. After that, the result is entered back into the destination register.

Out-of-order execution is an approach used in third-, fourth-, and fifth-generation microprocessors. This approach significantly reduces latency when executing instructions. The specificity is that the processor will execute instructions in the order of data or operand availability, but not in the original order of instructions generated by the compiler. In this way, the processor will avoid waiting states, because during the execution of the current instruction, it will obtain operands for the next instruction. For example, I1 and I2 are two instructions where I1 is the first and I2 is the second. In out-of-order execution, the processor may execute an I2 instruction before the I1 instruction is completed. This feature will improve CPU performance as it allows execution with less latency.

The steps required for out-of-order processor are as follows:

1. Retrieves instructions from the program memory.
2. Instructions are sent to an instruction queue (also called instruction buffer).
3. Until the input operand is available the instruction waits in the queue. The instructions will leave the queue when the operand is available. This implies that instructions are dynamically scheduled.
4. The instruction is sent to appropriate execution unit for execution.
5. Then the results are queued.
6. If all the previous instructions have their results written back to register file, then the current result is entered back to the destination register.

The main goal of out-of-order instruction execution is to increase the amount of ILP. But let note that the hardware complexity of out-of-order processors is significantly higher compared to in-order ones.

5. SCALAR VS SUPERSCALAR PROCESSORS

A scalar processor is one where instructions are executed in a pipeline, as is presented in Fig. 1a), but only a single instruction can be fetched or decoded in a single cycle. A super scalar processor on the other hand can have multiple parallel instruction pipelines [20], [21]. A 2-way super scalar processor (see Fig. 1b)) can fetch two instructions per cycle and supports two parallel pipelines. The terms "scalar" or "superscalar" are not to be confused with "single-core/multi-core". Scalars are single-core processors, while superscalars may either be single- or multi-cores. The key point is that scalars cannot perform more than one operation (i.e., carry out more than one instruction) per clock cycle, but

superscalars can perform up to two instructions in some cases. This means that if you have a CPU with three cores on it – one being an old scalar processor – and you run an application that utilizes all three cores, the old third core will be no more than half as fast as if it were completely superscalar. The main thing to remember is that certain instruction sets are suited better to certain optimizations. Superscalars can execute basic operations such as add and load on separate registers simultaneously, whereas a scalar processor would have to complete one operation before moving on to the next. For example, a scalar processor may be able to run multiple threads, but they will all share the same core and therefore only run as fast as the slowest thread. Superscalars can provide much higher performance because each thread gets its own core/execution unit.

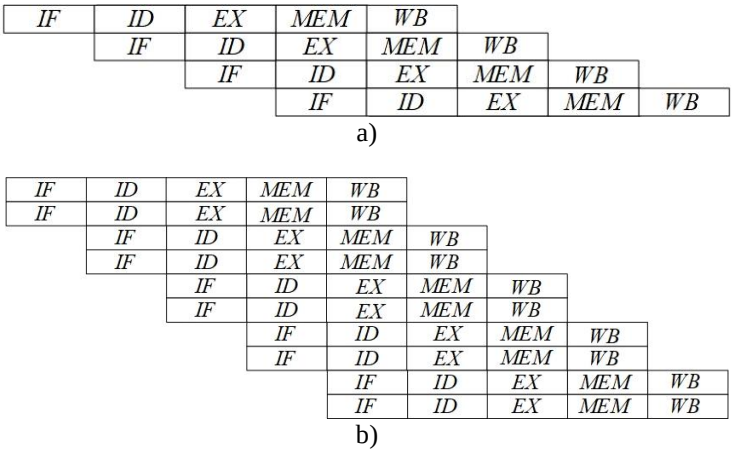


Fig. 1 Scalar processor (a), superscalar processor (b)

The terms "scalar" or "superscalar" are not to be confused with "single-core/multi-core". Scalars are single-core processors, while superscalars may either be single- or multi-cores [22]. The key point is that scalars cannot perform more than one operation (i.e., carry out more than one instruction) per clock cycle, but superscalars can perform up to two instructions in some cases. This means that if you have a CPU with three cores on it – one being an old scalar processor – and you run an application that utilizes all three cores, the old third core will be no more than half as fast as if it were completely superscalar. The main thing to remember is that certain instruction sets are suited better to certain optimizations. Superscalars can execute basic operations such as add and load on separate registers simultaneously, whereas a scalar processor would have to complete one operation before moving on to the next. For example, a scalar processor may be able to run multiple threads, but they will all share the same core and therefore only run as fast as the slowest thread. Superscalars can provide much higher performance because each thread gets its own core/execution unit.

The main challenge in superscalar processing is how many instructions can be issued per cycle. If a processor can issue k instructions per cycle, then it is called a k-degree superscalar processor. In order for a superscalar processor to take full advantage of parallelism, then k instructions must be executable in parallel. So, the key idea of a superscalar processor is that there is more instruction level parallelism (ILP) [18].

The implementation of superscalar processing requires special hardware (see Fig. 2 for more details). The data path is increased with the degree of superscalar processing. For instance, if 2-degree superscalar processor is used and the instruction size is 32 bit, then 64-bit data is fetched from the instruction memory and 2 instruction registers are required.

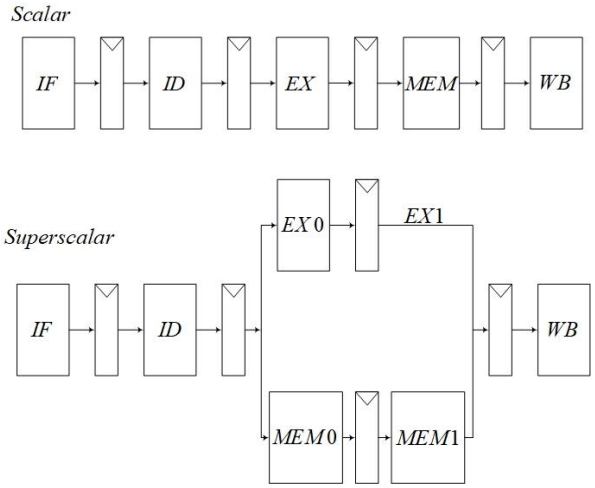


Fig. 2 Comparison between a Scalar and a Superscalar Processor.
Notice: The Superscalar Processor implements one pipeline dedicated for Memory Access and one pipeline for Arithmetic operations.

The main feature of superscalar processors is to issue more than one instruction in each cycle (usually up to 8 instructions). Let note that instructions can change the order to make better use of the processor architecture.

In order to reduce data dependency in superscalar processing, more complex parallel hardware is necessary. Hardware parallelism ensures the availability of more resources and it is one of the ways to use parallelism. An alternative way is to use ILP which can be achieved by transforming the source code using an optimization compiler.

Typical commercial superscalar processors are IBM RS/6000, DEC 21064, MIPS R4000, Power PC, Pentium, etc.

Very-Long-Instruction-Word (VLIW) processors are a variant of superscalar processors because they can process multiple instructions in all pipeline stages [23]. The VLIW processor has the following features: (a) it is an in-order processor; (b) the binary code defines which instructions will be executed in parallel. The size of the VLIW instruction word can be in hundreds of bits. The compiler forms the layout of the VLIW instruction by compacting the instruction words of the source program. The processor must have the sufficient number of hardware resources to execute all the specified operations in VLIW word simultaneously. For instance, as shown in Fig. 3, one VLIW instruction word is compacted to have L/S operation, FP addition, FP multiply, branch, and integer ALU.

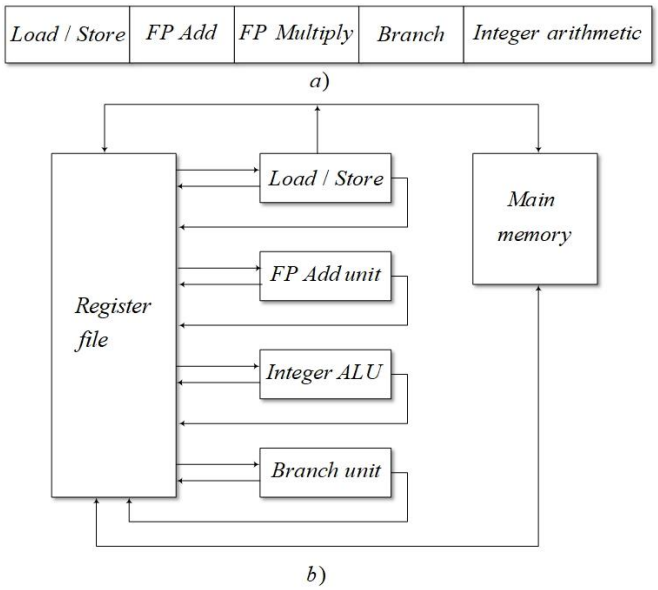


Fig. 3 a) VLIW instruction word; b) VLIW processor

All functional units (shown in Figure 3 b)) are implemented according to the VLIW instruction word (given in Figure 3 a). Large registry file is shared by all functional units in the processor. The parallelism in instructions and data flow is specified at compile time. Trace scheduling is used for handling branch instructions. It is based on the prediction of branch decisions at compile time, while prediction is based on some heuristic methods.

In Table 2 a comparison between VLIW and Superscalar processors from aspect of ILP implementation is given.

As conclusion, when we compare VLIW and Superscalar Processors, we can say that VLIW differs from superscalar machine in the following: a) instruction decoding process is simpler; b) ILP is higher but code density is lower; and c) object-code compatibility with a larger family of nonparallel machines is lower.

Table 2 Instruction-Level Parallelism: VLIW vs Superscalar

Superscalar	VLIW
Instruction scheduling mechanism is implemented with complex hardware Out-of-order execution ▪ There is a logic that checks the dependencies between parallel instructions and checks the hazards when working functional units Longer execution time and higher power consumption are potential consequences More efficiently execution of pipeline-dependent code	More functional units are needed Instruction code word is larger Complex compiler is needed If a compiler that performs efficient code optimization is not implemented, then more effort is needed to create executable code Hardware is simpler due to the use of predicted execution to avoid branching

Simple hardware structure and instruction set are the crucial advantages of VLIW architecture. The VLIW processor is suitable for scientific applications where the program behavior is more predictable.

Super-Pipelining is an alternative performance method to superscalar. In this approach, pipeline stages can be segmented into *n* distinct non-overlapping parts each of which can execute in 1/*n* of a clock cycle, i.e., Super-Pipelining is based on dividing the stages of a pipeline into sub-stages and thus increasing the number of instructions which are active in the pipeline at a given moment. By dividing each stage into two, the cycle period τ is reduced to the half, τ/2 => at maximum capacity, a result is produced every τ/2 s (see Fig. 4). For a given architecture and the corresponding instruction set there is an optimal number of pipeline stages; increasing the number of stages over this limit reduces the overall performance [24].

By analyzing Fig. 4 we can observe the following:

1. Base pipeline: *i*) Issues one instruction per clock cycle; *ii*) Can perform one pipeline stage per clock cycle; *iii*) Several instructions are executing concurrently; *iv*) Only one instruction is in its execution stage at any one time; and *vi*) Total time to execute 6 instructions is 10 cycles.
2. Super-Pipelined implementation: *j*) Capable of performing two pipeline stages per clock cycle; *jj*) Each stage can be split into two non-overlapping parts: *jjj*) Each executing in half a clock cycle; *jiv*) Total time to execute 6 instructions is 7.5 cycles; *jv*) Theoretical speedup is equal to $1 - 7.5 / 10 \approx 25\%$.
3. *Superscalar* implementation: *k*) Capable of executing two instances of each stage in parallel; *kk*) Total time to execute 6 instructions is 7 cycles; and *kkk*) Theoretical speedup: $1 - 7/10 \approx 30\%$.

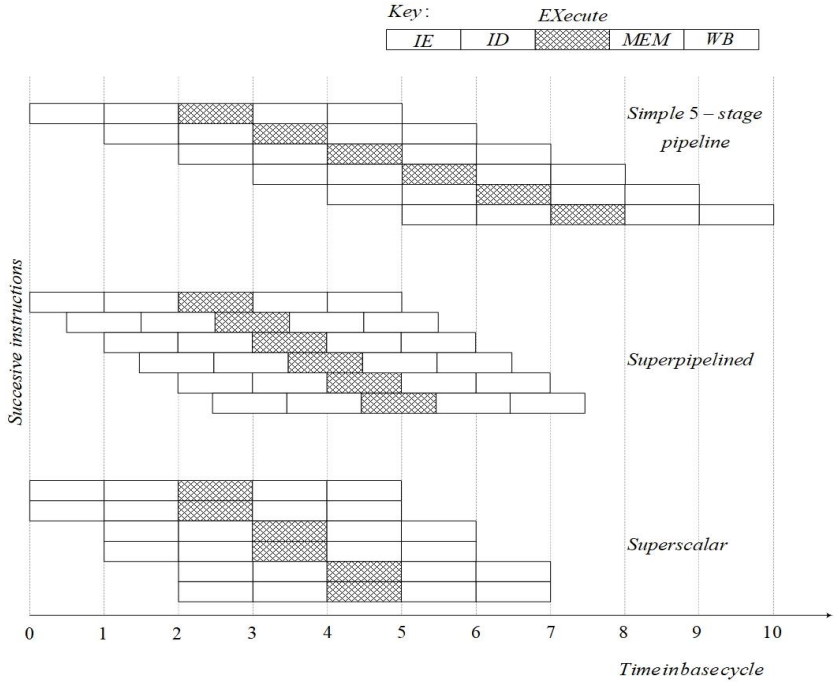


Fig. 4 Comparison of superscalar and super-pipeline approaches

From the Fig. 4 we can notice that both the Super-Pipeline and the Superscalar implementations: *a*) Have the same number of instructions executing at the same time; *b*) However, Super-Pipelined processor falls behind the superscalar processor; and *c*) Parallelism empowers greater performance. So, a better solution to further improve speed is the Superscalar architecture.

6. VECTOR PROCESSOR

A vector is an ordered set of the same type of scalar data items that can be of type a floating-point number, an integer, or a logical value. Vector processing is the arithmetic, or logical computation, applied on vectors whereas in scalar processing only one or pair of data is processed. Therefore, vector processing is faster compared to scalar processing. When the scalar code is converted to vector form then it is called vectorization. A vector processor is a special accelerator building block, which is designed to handle the vector computations [25], [26].

There are the following types of vector instructions:

a) *Vector-Vector Instructions*: Vector operands are fetched from the vector register and after processing generated results are stored in another vector register. These instructions are marked with the following function mappings:

$$P1: V1 \Rightarrow V2$$

$$P2: V1 \times V2 \Rightarrow V3$$

For example, P1 type denotes vector square root, and P2 addition (or multiplication) of two vectors.

b) *Vector-Scalar Instructions*: Scalar and vector operands are fetched and stored in vector register. These instructions are denoted with the following function mappings:

$$P3: S \times V1 \Rightarrow V2 \quad ; \text{where S is the scalar item}$$

For example, P3 type denotes vector-scalar subtraction or divisions.

c) *Vector-Reduction Instructions*: This type of instructions is used when operations on vector are being reduced to scalar items as the result. These instructions are presented with the following function mappings:

$$P4: V1 \Rightarrow S1$$

$$P5: V1 \times V2 \Rightarrow S2$$

For example, P4 type corresponds to finding the maximum, minimum and summation of all the elements of vector, while P5 is used for the dot product of two vectors.

d) *Vector-Memory Instructions*: This type of instructions is used when vector operations with memory M are performed. These instructions are marked with the following function mappings:

$$P6: M1 \Rightarrow V1$$

$$P7: V1 \Rightarrow M2$$

For example, P6 type corresponds to vector load and P7 to vector store operation.

Typical examples of vector operations are the following:

1. $V2 \Leftarrow V1$ *; Complement all elements*
2. $S \Leftarrow V1$ *; Min, Max, Sum*
3. $V3 \Leftarrow V2 \times V1$ *; Vector addition, multiplication, division*
4. $V2 \Leftarrow V1 \times S$ *; Multiply or add a scalar to a vector*
5. $S \Leftarrow V2 \times V1$ *; Calculate an element of a matrix*

Vector Processing with Pipelining: Due to the repetition of the same computation on different operands, vector processing is very suitable for pipelining. A vector processor performs better if length of vector is larger, but it causes the problem in storage and manipulating of vectors.

Efficiency of Vector Processing over Scalar Processing: As we have already mentioned, a sequential computer processes vector item by item. Therefore, with aim to process a vector of length *n* through the sequential computer then the vector must be divided into *n* scalar steps and executed one by one.

For example, consider the following example which is used for addition of two vectors of length 1000:

$$\boldsymbol{A}+\boldsymbol{B}\Rightarrow\boldsymbol{C}$$

The sequential computer implements this operation by 1000 add instructions in the following way:

$$C[1]=A[1]+B[1]$$
$$C[2]=A[2]+B[2]$$
$$\vdots$$
$$C[1000]=A[1000]+B[1000]$$

A vector processor does not divide the vectors in 1000 add statements to perform identical operation, because it has the set of vector instructions that allow the operations to be specified in single vector instruction as:

$$\boldsymbol{A}(1:1000)+\boldsymbol{B}(1:1000)\Rightarrow\boldsymbol{C}(1:1000)$$

Comparative execution of addition instruction by scalar and vector processor is presented in Fig. 5.

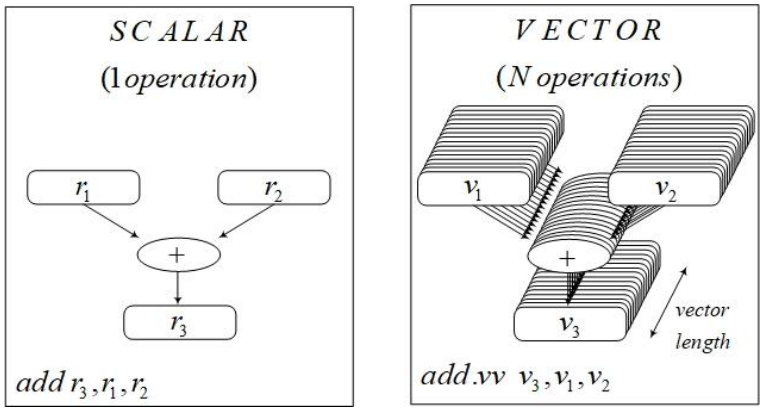


Fig. 5 Scalar vs Vector operations execution

Thus, the main advantage of using vector in respect to scalar processing is reflected in the elimination of overhead caused by the loop control.

Properties of Vector Instructions:

a) Single Instruction implies lot of operations: Hence reduce the number of instruction's fetch and decode.
b) Each operation is independent of each other: i) Simple design; ii) Multiple Operations can be run in parallel.
c) Data hazards has to be checked for each vector operation and not each operation.
d) Reduces Control hazards by reducing branches.
e) Knows memory access pattern.

Nowadays, large number of microprocessors contain a set of instructions that manipulate with relatively small vectors (e.g., up to 8 single-precision FP elements in the Intel AVX extensions [27]). These instructions are often referred to as SIMD (single instruction, multiple data) instructions.

Table 3 shows the comparative properties (advantages vs disadvantages) of vector processors.

Table 3 Comparative properties of vector processors

Advantages of Vector Processors	*Disadvantages of Vector Processors*
▪ Instruction bandwidth is lower ▪ Fetch and decode phases are reduced ▪ Main memory addressing is easier ▪ Load/Store units use known patterns for memory access ▪ Memory wastage is eliminated – no cache misses, latency only occurs during vector loading ▪ Control hazards logic is simple – loop-related control hazards are eliminated ▪ Scalable platform – larger number of hardware resources increases performance ▪ Code size is reduced – N operations are described by single instruction	▪ Works (only) if parallelism is regular (data/SIMD parallelism). ▪ Very inefficient if parallelism is irregular. ▪ Memory (bandwidth) can easily become a bottleneck especially if: a) Compute/memory operation balance is not maintained; b) Data is not mapped appropriately to memory banks

Vector Processing Applications include problems that can be efficiently formulated in terms of vectors such as: a) Long-range weather forecasting; b) Petroleum explorations; c) Seismic data analysis; d) Medical diagnosis; e) Aerodynamics and space flight simulations; f) Artificial intelligence and expert systems; g) Mapping the human genome; and h) Image processing.

7. MULTICORE PROCESSORS

Nowadays, large uniprocessors no longer scale in performance, because conventional superscalar techniques for instruction issue allow only a limited amount of parallelism to be extracted from the instruction flow. In addition, it is not possible to further increase the clock speed, because the power dissipation will become prohibitive.

For more than thirty years (time period between 1972-2003 year, often called as time intensive microarchitecture processor design), a variety of modifications have been conducted to perform one of two goals: 1) increasing the number of instructions that can be issued per cycle; and 2) increasing the clock frequency faster than Moore's law and

Denard's rule would normally allow [28]. Pipelining and super-pipelining of individual instruction execution into a sequence of stages has allowed designers to increase clock rates. Superscalar processors were designed to execute multiple instructions from an instruction stream on each cycle. These function by dynamically examining sets of instructions from the stream to find one's capable candidates for parallel execution on each cycle. These can be often executed in out-of-order manner with respect to the original sequence. This concept is referred as *instruction-level parallelism* (ILP). Typical instruction streams have only a limited amount of usable parallelism among instructions [1], [29], so superscalar processors that can issue more than about four instructions per cycle achieve very little additional benefit on most applications.

Today, advances in processor core development have slowed dramatically because of a simple physical limit: *power dissipation*. In modern pipelined and superscalar processors, typical high-end power exceeds 100 W. In order to bypass the mentioned design constraints, processor manufacturers are now switching to a new microprocessor design paradigm: multicore (also called chip multiprocessor, or CMP for short) and many-core.

A *multi-core processor* is a single computing component with two or more independent actual processing units (called "cores" - made up of computation units and caches [30]), which are functional units that read and execute program instructions. Multiple cores can run multiple instructions (ordinary CPU instructions) at the same time, increasing overall speed for programs suitable to parallel computing. Coupling multiple cores on a single chip should achieve the performance of a single faster processor. The individual cores on a multi-core processor are not necessary to run as fast as the highest performing single-core processors, but in general they improve overall performance by executing more tasks in parallel [31]. The increase in performance can be seen by considering the way single-core and multi-core processors execute programs. Single-core processors that run multiple programs will assign different time slices to all programs, and they will run sequentially. If one of the processes lasts longer, then all the other processes start to lag behind. However, with multi-core processors, if there are multiple tasks that can run in parallel at the same time, then each of them will be executed by a separate core in parallel. This improves performance.

Depending on the application requirements, multi-core processors can be implemented in different ways. It can be a group of heterogeneous cores or a group of homogeneous cores or a combination of both. In a homogeneous core architecture, all cores in the processor are identical [32] and in order to improve overall performance they break down a computationally intensive application into less intensive applications and run them in parallel [4]. Significant advantages of a homogeneous multi-core processor are reduced design complexity, reusability, and reduced verification effort [33]. Heterogeneous cores, on the other hand, consist of dedicated application specific processor cores that would run various applications [34].

Cores in multi-core systems, as well as single-processor systems, can implement architectures such as VLIW, superscalar, vector, or multithreading. Multicore processors are used in many application domains, such as general purpose, embedded, multimedia, network, digital signal processing (DSP) and graphics (GPU). They can be harnessed as complex cores that address computationally intensive applications, or a remedial core that deals with less computationally intensive applications [24].

Software algorithms and their implementations greatly influence the performance improvement obtained by using multi-core processors. In particular, possible gains are limited by the fraction of the software that can run in parallel simultaneously on multiple

cores. At best, parallel problems can achieve acceleration factors close to the number of cores, or even more if the problem is sufficiently split to fit in the local core cache(s). In this way the number of accesses to much slower main system memory are reduced. However, most applications are not so fast without the effort of programmers to reshape the whole problem. Currently, software parallelization is a significant ongoing research topic [4].

A comparison between single and multiple-core processor is given in Table 4.

Table 4 Comparation of Single-Core processor and Multi-Core processor

Parameter	Single-Core Processor	Multi-Core Processor
Number of cores on a die	Single	Multiple
Instruction Execution	Single instruction is executed at a time	Multiple instructions are executed by using multiple cores
Gain	Speed up every program	Speed up the programs intended for multi-core processor
Performance	Depend on the clock frequency	Depend on the clock frequency, number of cores and program
Examples	80386, 80486, AMD 29000, AMD K6, Pentium I, II, III etc.	Core-2-Duo, Athlon 64 X2, I3, I5, I7 etc.

7.1. Multicore topologies

In the sequel we will point out to four types of multicore topologies: symmetric (or homogeneous), asymmetric, dynamic, and composed (alternatively referred as "fused" or "heterogeneous") [20], [35].

The symmetric multicore topology is composed of multiple copies of the same core that functioning at the same frequency and voltage. In this topology, the resources such as the power and the area budget, are evenly distributed on all cores. In Figure 6a) symmetric multicore processor is presented where each block is a Basic Core Equivalent (BCE) and contains L1 and L2 caches as constituents. L3 cache and on-chip network are not presented.

The asymmetric multicore topology is composed of one large monolithic core and a number of identical small cores. This topology uses a large high-performance core that performs the serial part of the code and uses a number of small cores as well as the large core to take advantage of the parallel part of the code. In Figures 6b) and 6c) asymmetric multicore processors are presented with: b) one complex core and 12 BCEs; c) two complex cores and 8 BCEs.

The dynamic multicore topology is a modification of the asymmetric topology. Parallel parts of the code are executed by small cores while the large core is off, and the serial part of the code is executed only on the large core, while small cores are inoperative. In Figures 6d) and 6e) dynamic multicore processors are presented with: d) 16 BCEs or one large core; e) four cores and frequency scaling using power budget of 8 BCEs (currently one core is at full core thermal design point (TDP), two cores are at 0.5 core TDP, and one core is switched off).

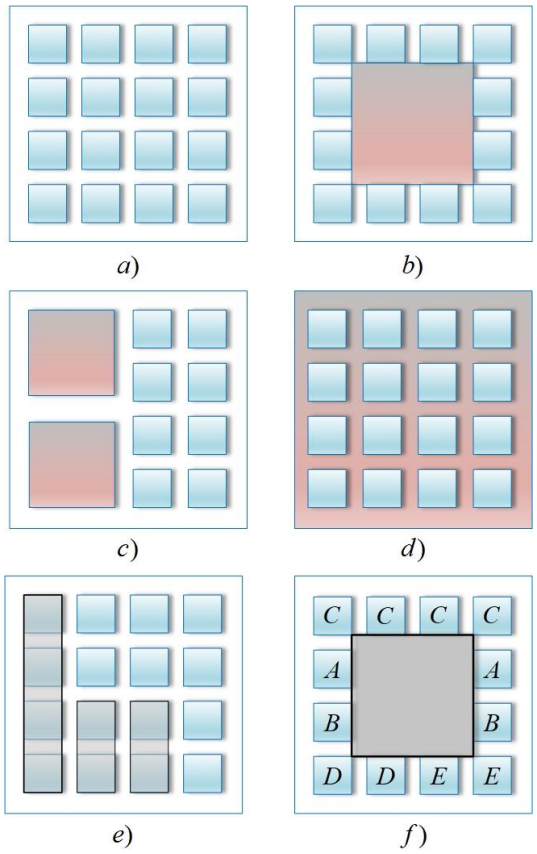


Fig. 6 a) Symmetric multicore processor; b) and c) Asymmetric multicore processors; d) and e) Dynamic multicore processors; f) Heterogeneous multicore

The heterogeneous (composed) multicore topology is composed of a set of small cores that are logically combined to assemble a large high-performance core for serial code execution. In serial or parallel cases, exclusively large or small cores are used. In Figure 6f) heterogeneous multicore is presented with large core, four BCEs, two accelerators or co-processors of type A, B, D, E each.

Some of the limits of multicore processors are the following [36]:

1. Present days CMPs are designed to exploit both instruction-level parallelism (ILP) and thread-level parallelism (TLP). In such solutions, the number of processors and the complexity of each processor are fixed at design time.
2. Performance improvement mainly achieved by increasing the number of cores cannot always lead to effective design solution due to: a) Dark silicon problem (all the cores cannot be powered at the same time); and b) Declining yield in TLP.

Nowadays, we have multicore processors all over the place, single thread programs are no longer an option. In essence, we moved from single core to multicore not because the software community was ready for concurrency but because the hardware community could not afford to neglect the power issue.

Today, multi-core technology has become commonly used in most personal electronic devices that contain multiple cores. Therefore, in order to take advantage of multiple cores on such machines, creating parallel programs is crucial to achieving high performance and enabling large-scale data processing.

In addition to multicore technology (mainly realized as shared memory systems) [37], parallel computing can be in the form of distributed systems. Unlike multicore shared memory systems, distributed systems can solve problems that do not fit in the memory of a single machine. In contrast to multicores with shared memory, communication and data replication in distributed systems causes high additional overheads. Compared to distributed memory systems, multicores with shared memory are more efficient for programs that can fit in memory. Efficiency is reflected in reduced hardware, cost, and power consumption [3].

Today's multicore CPUs use most of their transistors on processing logic and cache memory. During operation most of the power is consumed by non-computational units. Alternative strategy are heterogeneous architectures, i.e. multi-core architectures in combination with accelerator cores. Accelerators are specialized hardware cores designed with fewer transistors, operating at lower frequencies than traditional CPUs, and enabling increased system performance.

8. MULTITHREADED PROCESSORS

As hardware complexity of modern processor and capabilities have increased, so demands related to higher performance increased too. This requirement has led to an increase in CPU resource efficiency to the same extent. The main idea is that the time while the processor is waiting to perform certain tasks, i.e. it is in idle state, is used to perform another activities. To achieve this goal, software designers involved new approach in possibilities of the operating system that support running pieces of programs, called threads. Threads are small tasks that can run independently [38]. During execution, each thread gets its own time period. As a consequence, the processor time is efficiently utilized. Fig. 7 shows multithreading execution on single processor and two-way superscalar processor.

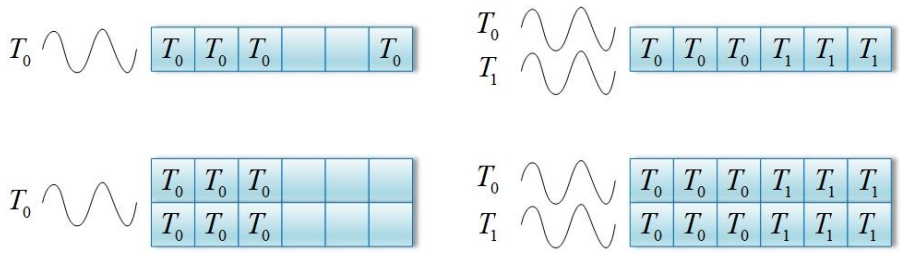


Fig. 7 Multithreading in a CPU: (a) Single processor running a single thread. (b) Single processor running several threads. (c) Two-way superscalar processor running a single thread. (d) Two-way superscalar processor running multiple (two) threads.

8.1. Difference between Multitasking, Multiprocessing and Multithreading

Several threads make up one process (task), and share access to processor resources. This new concept of operating systems, known as multi treading, has ensured the run of one thread while the other is in a state of waiting for an event. Contemporary commercially available PC machines and servers, mainly based on Intel or AMD processors, that run Microsoft Windows, support multithreading [28].

Each program requires resources that are occupied by the process (task). The process is assigned a virtual address space, executable code, system object manipulation, a security context, a unique process identifier, environment variables, a priority type, working set sizes, and at least one thread of execution. A thread is a single entity within a process that can be planned for execution. All threads that are part of a process share its previously mentioned resources. In addition, each thread maintains code for manipulation with exceptions, a planning priority, a local thread memory, a thread identifier, and structures that the system will use in order to preserve the context of the thread. The thread context consists of a set of machine registers, a kernel stack, an environment block, and a user thread stack. Each thread is characterized by: 1) thread ID; 2) register state, including PC and SP; 3) stack; 4) signal mask; 5) priority; and 6) thread-private memory. Threads share instructions and data of the process to which they belong. All threads in the process can see changes in the shared data of any thread. Threads in the same process can interact with each other without involving the operating environment.

Multitasking is a mode of operation where the CPU performs multiple tasks at the same time (see Fig. 8). It is characterized by CPU switching between multiple tasks so that users can work together with each program. Unlike multithreading, in multitasking, processes share separate memory and resources. In multitasking, CPU switching between tasks is relatively fast.

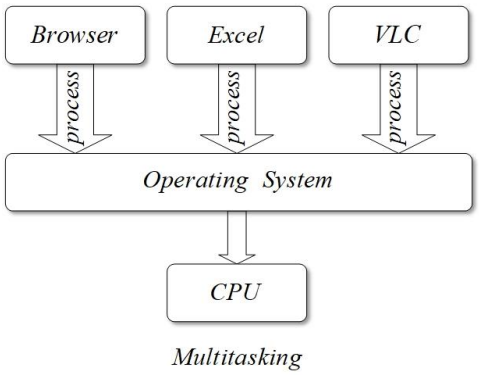


Fig. 8 Multitasking operating system for single processor

Multithreading is an operating mode in which during process execution many threads are active. In this manner, higher computer power is achieved. In multithreading (see Fig. 9), CPU executes many threads that are part of a process at a time. Processes share the same memory and resources. Property of multithreading is that two or more threads can run concurrently. Therefore, multithreading is also referred as concurrency [39].

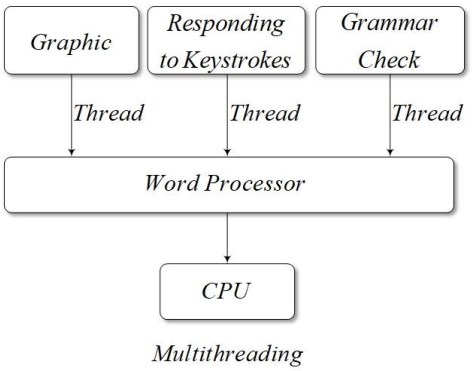


Fig. 9 Multithreading system for single processor

The difference between multitasking and multithreading [40] is presented in Table 5.

Table 5 Difference between Multitasking and Multithreading

No.	Multitasking	Multithreading
1.	CPU performs many user tasks	Within a process many threads are created
2.	CPU switching among the tasks	CPU switching among the threads
3.	Processes share separate memory	Processes share same memory
4.	Multiprocessing can be involved	Multiprocessing cannot be involved
5.	CPU executes many tasks at a time	CPU executes many threads at a time
6.	Each process has separate resources	Each process shares same resources
7.	Multitasking is slower	Multithreading is faster
8.	Termination of process is longer	Termination of thread is shorter

A computer system composed of two or more processors is called a multiprocessing system (see Fig. 10). In this way, the computing speed of the system is increased. In such systems, each processor has its own registers and main memory. The division of processes and resources among processors is done dynamically.

The main characteristics of multiprocessing are the following: i) The organization of memory determines the type of multiprocessing; ii) System reliability is improved, and iii) Decomposing programs into parallel executable tasks leads to performance increase.

Advantages of multiprocessing are the following: a) more activity can be performed in a shorter time; b) code is simple; c) system is composed of multiple CPU and cores; d) synchronization is simplified; e) child processes are interruptible/killable; and f) cost-efficient because processors share resources.

Disadvantages of multiprocessing are: a) inter-process communication involves time overhead; and b) larger memory is needed.

The main characteristics of multithreading are the following: j) each thread is executed parallel with other; and jj) program performance is increased since threads share the same memory area.

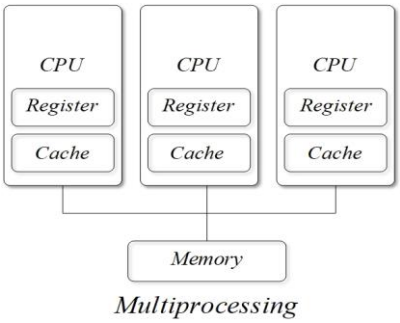


Fig. 10 Multiprocessing system

Advantages of multithreading are the following: a) the address space is shared for all threads; b) lower amount of memory is needed; c) cost-efficient and fast communication between threads; d) fast context switching; e) suitable for input/output-oriented applications; and f) switching time between two threads is short.

Disadvantages of multithreading are the following: a) not interruptible/killable; b) manual synchronization is often necessary; and c) program code is harder to understand and testing and debugging is harder due to race conditions.

Both multiprocessing and multithreading as operating modes increase a computing power [41]. A multiprocessing system is composed of multiple processors where a multithreading comprises multiple threads.

Table 6 Multiprocessing vs Multithreading

Multiprocessing	Multithreading
Multiple CPUs increase computing power	Multiple threads of a single process increase computing power
Multiple processes are executed concurrently	Multiple threads of a single process are executed concurrently

9. MULTITHREADING: EXECUTION MODEL

One decade later in respect to software architects, hardware architects designed a multithreaded processor which can run more than one thread on some of its cores at the same time. A multithreaded architecture is one in which a single processor has the ability to follow multiple streams of execution without the aid of software context switches. In order for a conventional processor to stop executing one thread and start executing instructions from another thread, it requires special software. The role of this software is to transfer the state of the running thread to memory (usually to stack memory) and then load the state of the selected other thread into the processor. This process usually requires hundreds (or thousands) of cycles, especially if an operating system was introduced. A multithreaded architecture, on the other hand, can access the state of multiple threads in, or near, the processor core. This allows the multithreaded architecture to quickly switch between threads, and potentially more efficiently and effectively use processor resources [42] (see for illustration Fig. 11).

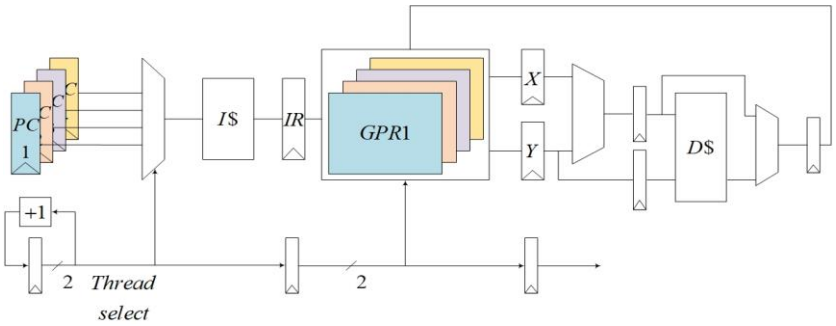


Fig. 11 Multithreaded pipeline example

Multicore and multithreading can be used simultaneously because they are two orthogonal concepts. For instance, the Intel Core i7 processor has multiple cores, and each core is two-way multithreaded [43]. In the case where multiple threads are executed simultaneously, then those threads use mostly different hardware resources in the multicore, while they share most of the hardware resources in the multithreaded processor.

In order to achieve this, a multithreaded architecture must be able to store the state of multiple threads in hardware - this storage is referred to as hardware contexts, where the number of supported hardware contexts defines the level of multithreading (the number of threads that can share the processor without software intervention). The state of a thread is primarily composed of the program counter (PC), the contents of general-purpose registers, and special purpose and program status registers. It does not include memory (because that remains in place), or dynamic state that can be rebuilt or retained between thread invocations (branch predictor, cache, or TLB contents).

9.1. Instructions issue

Multithreaded processors are divided into two groups depending on how many threads can issue instructions in a given cycle. When instructions can be issued only from a single thread in a given cycle, explicit multithreading is used. In that case, the following two main techniques can be applied [44] (see Fig. 12): *i*) Coarse-grain multithreading (CGMT) or Blocked multithreading (BMT); and *ii*) Fine-grain multithreading (FGMT) or Interleaved multithreading (IMT). When instructions can be issued from multiple threads in a given cycle, Simultaneous multithreading (SMT) is used.

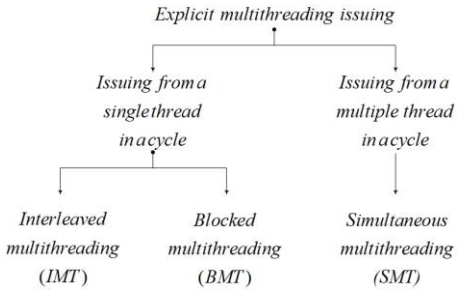


Fig. 12 Explicit multithreading

Coarse-grain multithreading, also called blocked multithreading or switch-on-event multithreading, has multiple hardware contexts associated with each processor core [45]. The instructions of a thread are executed successively, but when an event occurs that may cause latency, then it produces a context switch. Instructions of one thread continue to be executed until there is a long delay such as a branch or no cache data is found (see Fig. 13). When such a delay is achieved, it is switched to another thread, and this thread is also executed until a long delay occurs. This process is constantly repeated. The strategy of this technique makes it possible to hide long delays, but omits shorter delays where the cost of switching is higher than the cost of tolerating delays. A hardware context is the program counter, register file, and other data required to enable a software thread to execute on a core. A coarse-grain multithreaded processor operates similarly to a software time-shared system, but with hardware support for fast context switch, allowing it to switch within a small number of cycles (e.g., less than 10) rather than thousands or tens of thousands.

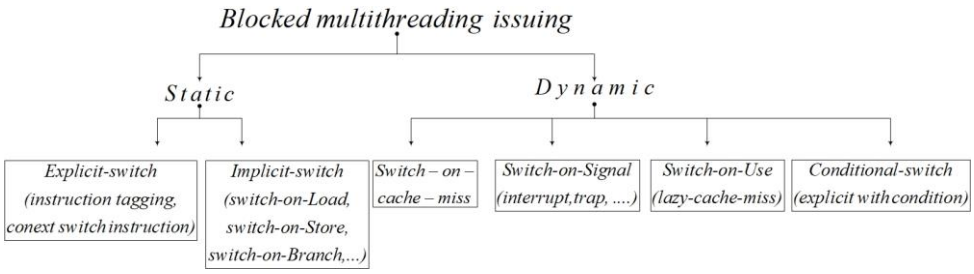


Fig. 13 Coarse-grain multithreading

Fine-grain multithreading, also called interleaved multithreading, also has multiple hardware contexts associated with each core, but can switch between them with no additional delay. An instruction of another thread is fetched and entered into the execution pipeline at each cycle, and therefore the processor can execute an instruction or instructions from different thread in each cycle. Unlike coarse-grain multithreading, then, a fine-grain multithreaded processor has instructions from different threads active in the processor at once, within different pipeline stages [46]. But within a single pipeline stage (or given our particular definitions, within the issue stage) there is only one thread represented. In this approach, the CPU executes one instruction of each thread in succession (one after the other) before going back (in a circular way) to execute the next instruction of the first thread. During execution, the CPU skips the instruction of any thread that is waiting for an event to occur and has a long delay (stalled). In this manner, the processor is busy because the pipeline system is almost always full. Such a processor has significantly complex hardware structure because for each thread it needs a separate copy of register file and program counter.

Since the next instruction of a thread is fed into the pipeline after the withdrawal of the previous instruction of this thread, control and data dependencies between instructions do not occur in FGMT. The pipeline system is simple and potentially very fast because there is no need for complex hardware hazard detection. In addition, the context switching time between threads is zero cycles. Memory latency is compensated by not scheduling a thread until memory access is completed. In this model, the number of CPU pipeline stages determines the number of threads that can be executed. The processing power available to one thread is limited by the instruction interleaving from other threads.

In simultaneous multithreading (SMT) instructions are simultaneously initiated from multiple threads to the execution units of a superscalar CPU. In this way, the initiation of several superscalar instructions is linked with hardware resources for multiple-context approach. The CPU can issue multiple instructions from multiple threads each cycle. In this way, both unused cycles in the case of latencies and unused issue slots within one cycle can be filled by instructions of alternative threads. From one hand, TLP exists as a consequence of multithreading, parallel programs or multiple independent programs in a multiprogramming workload. From the other hand, ILP is based on execution of individual threads. SMT processor achieves better throughput and speedup in respect to single threaded superscalar processor for multithreaded workloads because it efficiently uses coarse- and fine-grain parallelism, at cost of more complex hardware architecture.

SMT, CGMT and FGMT are approaches which are often used in RISC or VLIW processors [39], [47]. Intel Pentium 4 implements SMT from 2002, starting from the 3.06 GHz model. Intel calls SMT technique as Hyper-Threading. Other processors that use SMT are Alpha AXP 21464, IBM Power5, and Intel Nehalem i7 [48]. One simple SMT architecture is presented in Fig. 14.

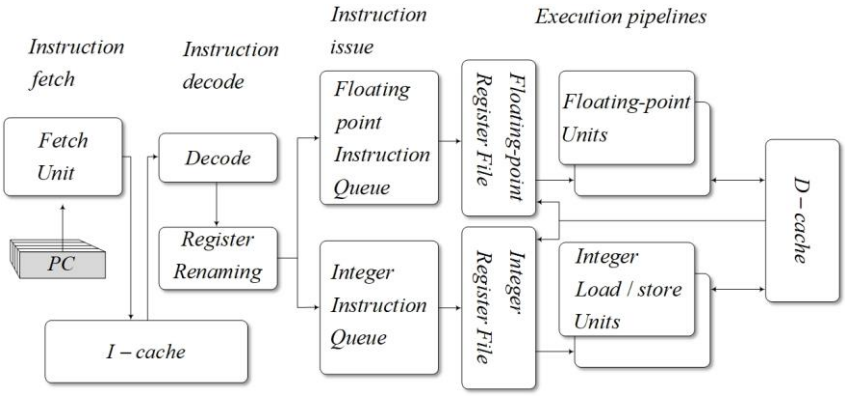


Fig. 14 SMT processor architecture

Much like pipelining, superscalar architecture (presented in Fig. 14) also extends very naturally the possibility to support multiple threads of instructions. A multi-threaded superscalar processor executes instructions from multiple threads. Each thread executes its logical instruction stream and uses separate registers, etc., but shares most of the available physical resources. The additional hardware required to support multiple thread execution is minor, but performance is significantly improved.

Short remarks related to explicit multithreading: Coarse-grain multithreaded processors directly execute one thread at a time, but can switch contexts relatively quickly, in a matter of a few cycles. This allows them to switch to the execution of a new thread to hide long latencies (such as memory accesses), but they are less effective at hiding short latencies. *Fine-grain multithreaded* processors can context switch every cycle with no delay. This allows them to hide even short latencies by interleaving instructions from different threads while one thread is stalled. However, this processor cannot hide single-cycle latencies. A *simultaneous multithreaded* processor can issue instructions from multiple threads in the same cycle,

allowing it to fill out the full issue width of the processor, even when one thread does not have sufficient ILP to use the entire issue bandwidth.

Illustration purpose only, two different approaches that are possible with single-issue (scalar) processors and multiple-issue processors are given in Fig. 15 and Fig. 16, respectively [37]. Fig. 17 presents two cases of issuing multiple threads in a cycle [38].

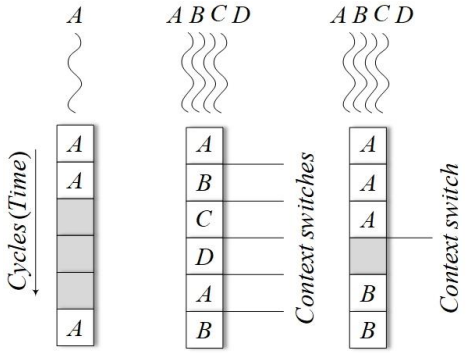


Fig. 15 Different approaches possible with single-issue (scalar) processors: a) single-threaded scalar, b) interleaved multithreading scalar, c) blocked multithreading scalar

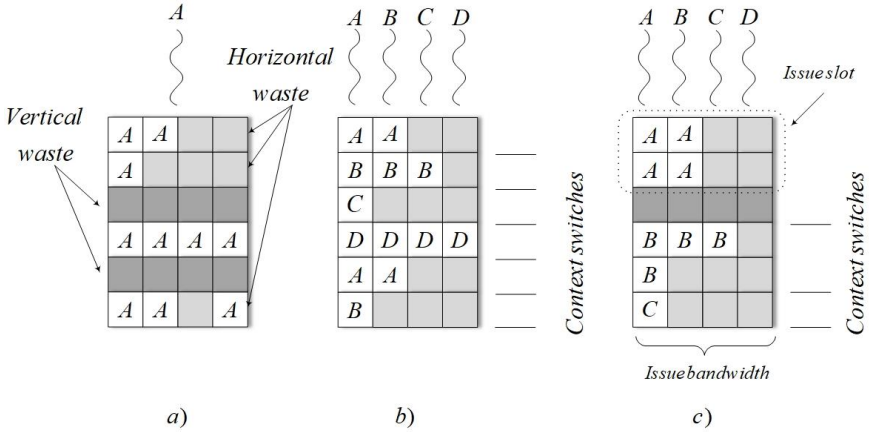
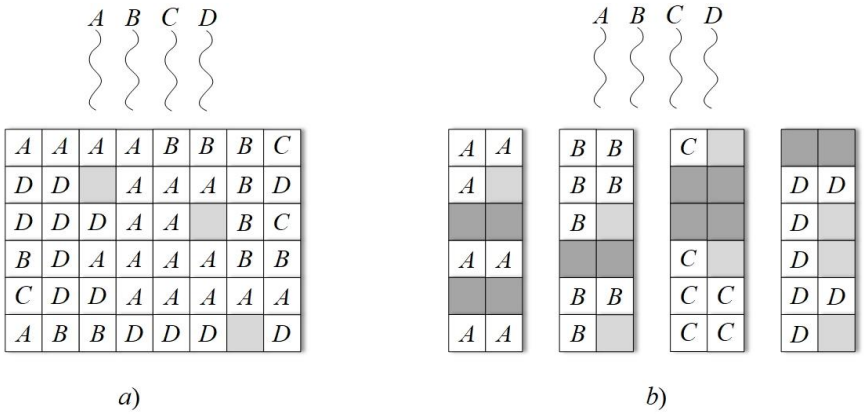


Fig. 16 Different approaches possible with multiple-issue processors: (a) single-threaded four-wide superscalar, (b) interleaved multithreading four-wide superscalar, (c) blocked multithreading four-wide superscalar

Notice: Vertical waste corresponds to darker marked box, while horizontal waste corresponds to lighter marked box



a) b)

Fig. 17 Issuing from multiple threads in a cycle: a) simultaneous multithreading, b) chip multiprocessor

10. PARALLEL VS SERIAL COMPUTING

What is Serial Computing: Computer software is conventionally created for serial execution, where the algorithm divides the problem into smaller parts, i.e. instructions. These instructions are then serially executed on the CPU of the computer one by one [18]. After completing the current instruction, the next one begins. So, in short, Serial (sequential) Computing is following (see Fig. 18):

- A problem is broken into a discrete series of instructions
- Instructions are executed sequentially one after another
- Executed on a single processor
- Only one instruction may execute at any moment in time.

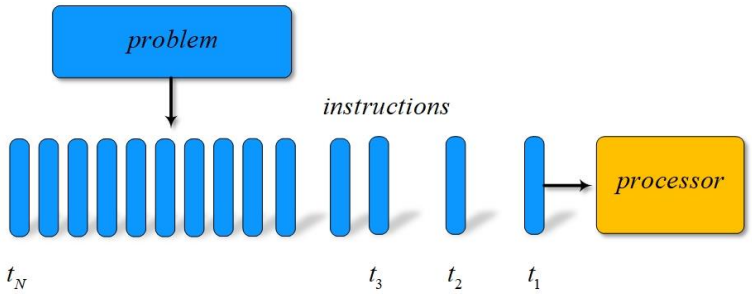


Fig. 18 Serial Computing generic example

What is Parallel Computing: Contrary to the serial approach, parallelism can be defined as an approach of dividing big problems into smaller ones. After that, smaller problems are simultaneously solved by multiple processors. The terms parallelism and concurrency are often confused. Parallelism means that two or more program sequences

are executed independently of each other by a number of processors, while in concurrent execution there are dependencies between program sequences so that the execution of one program sequence must wait for the execution of another to continue. Every parallel processing is not needed to be considered as concurrent. For example, bit-level parallelism is not concurrent.

As can be seen from Fig. 19, to solve a computational problem Parallel Computing involves the simultaneous usage of multiple computing resources [49]:

- A problem is decomposed into several parts that can be solved concurrently
- Each part is further decomposed down to a series of instructions
- Instructions from each part execute simultaneously on different processors
- A general control/coordination mechanism is implemented.

Concurrency vs Parallelism: We can see how concurrency and parallelism work with the below example. As shown in Fig. 20, there are two cores and two tasks. In a concurrent approach, each core is executing both tasks by switching among them over time. In contrast, the parallel approach doesn't switch among tasks, but instead executes them in parallel over time [50]. This simple example for concurrent processing can be any user-interactive program, like a text editor. In such a program, there can be some IO operations that waste CPU cycles. When we save a file or print it, the user can concurrently type. The main thread launches many threads for typing, saving, and similar activities concurrently. They may run in the same time period; however, they aren't actually running in parallel.

Types of Parallelism: In essence, the parallelism can be implemented at two levels, hardware and software, respectively [51].

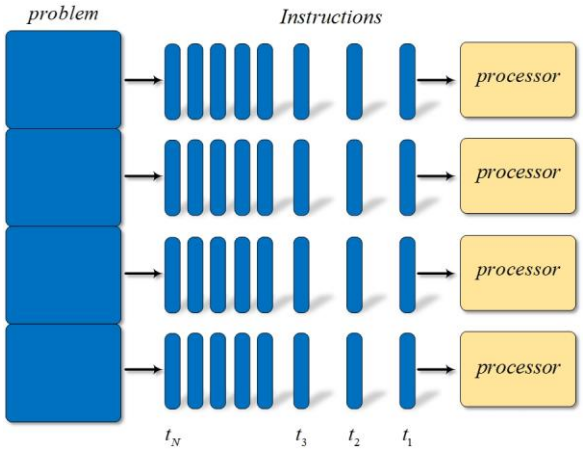


Fig. 19 Parallel Computing generic example

Parallelism at hardware level is built into machines architecture and hardware multiplicity, so it is also known as machine parallelism. This type of parallelism is a function of cost and performance trade off. It also displays resource utilization patterns of simultaneously executable operations and indicates the peak performance of processor resources. It is characterized by number of instruction issues per machine cycle [1].

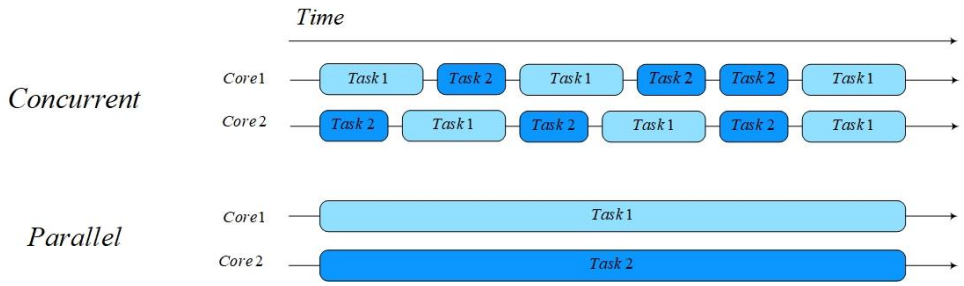


Fig. 20 Concurrent vs Parallel execution

In summary, we distinguish the following types of hardware parallelism:

1. Parallelism in a Uniprocessor level implemented as: *i*) Pipelining, Super-pipelining; *ii*) Superscalar, VLIW etc.
2. Parallelism implemented with SIMD instructions, Vector processors, GPUs
3. Parallelism at Multiprocessor level: *j*) Symmetric shared-memory multiprocessors; *jj*) Distributed-memory multiprocessors; *jjj*) Chip-multiprocessors a.k.a. multi-cores; *ivj*) Multicomputer a.k.a. clusters.

Parallelism at software level is exploited by the concurrent execution of machine language instructions in a program. This type of parallelism is a function of algorithm, programming style and compiler optimization. It also displays patterns of simultaneously executable operations using the program flow graph [52].

We distinguish the following two types of software parallelism:

1. Control parallelism – allows two or more operations to be performed simultaneously;
2. Data parallelism – at most same operation is performed over many data elements by many processors simultaneously.

Data level parallelism (DLP) arises from executing essentially the same code on a large number of objects [53], while control level parallelism (CLP) arises from executing different threads of control concurrently [54].

Parallelism in software level can be implemented at instruction, task, data or transaction level parallelism [55].

10.1. Implementations of the Most Common Type of Parallelism

Bit-level Parallelism: This type of parallelism (implemented at hardware level) uses doubling the processor word size. It provides faster execution of arithmetic operations for large numbers. For instance, an 8-bit CPU executes 16-bit addition for two cycles, whereas a 16-bit processor needs just one cycle for the same activity. This level of parallelism is also used in 64-bit processors.

Instruction-level parallelism (ILP): This type of parallelism (implemented at hardware level) exploits the potential overlap between instructions in a program. In most cases, ILP is implemented on each processor's hardware as: *i)* Instruction pipelining; *ii*) Superscalar processing; *iii*) Out-of-order execution; and *iv*) Speculative execution/Branch prediction.

Most processors use a combination of the aforementioned ILP techniques to achieve higher performance. Very Long Instruction Word (VLIW) processors use specialized compilers to achieve static ILP parallelism at the software level. Compilers prepare parallel instruction streams for VLIW processors so that they take full advantage of a number of executive units organized in multiple pipelines.

Task/Thread-level parallelism (TLP): This type of high-level parallel computing (implemented at software level) is based on partitioning the application in distinct task or threads, that can be then executed simultaneously. Threads are executed on different computer units and can work on independent data or share data. Until recently, programming was done sequentially, with a single thread representing the entire application. Today, it is necessary to use the new paradigm of multi-threaded programming in order to take full advantage of the available multicore processors. In that sense, modern operating systems (OSs) provide scheduling of different processes on different cores. However, in the case of complex applications such as bioinformatics, the OS cannot efficiently distribute the computational load of each process to available cores. In order to improve performance, these applications need to be redeveloped to achieve thread-level parallelism.

Data parallelism: This form of high-level parallelism partitions data into various available computing units. Data is assigned to cores that independently execute the same task code on each fragment of data. Therefore, this type of parallelism requires advanced code development skills and can only be applied to specific problems.

Computer graphics is an important area of application of high-level data parallelism. The design of graphic processor units (GPUs) enables efficient execution of every graphics processing task. First, each frame is divided into regions, and then, based on the command, hundreds of processor units perform the task independently on each data region.

Many incarnations of DLP architectures over decades are the following [49]: a) Old vector processors (Cray processors: Cray-1, Cray-2, …, Cray X1); b) SIMD extensions (Intel SSE and AVX units, Alpha Tarantula (didn't see light of day)); c) Old massively parallel computers (Connection Machines, MasPar machines); and d) Modern GPUs (NVIDIA, AMD, Qualcomm, ...). In general, DLP focus of throughput rather than latency.

In Fig. 21 a classification scheme of parallel computer architectures based on type of instruction processing is given.

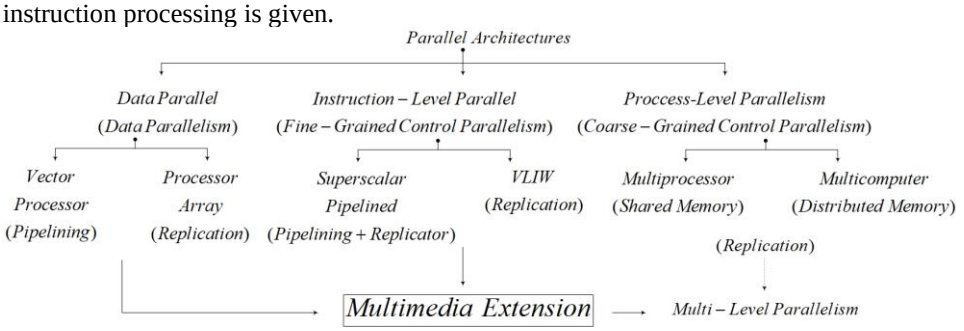


Fig. 21 Classification of Parallel Architectures based on type of instruction processing

The difference among the three major categories ILP, TLP and DLP that are nowadays mainly used in computer systems to exploit parallelism is sketched in Fig. 22, and that is [56]:

- *Instruction-Level Parallelism* (ILP) - Multiple instructions from one instruction stream are executed simultaneously;
- *Thread-Level Parallelism* (TLP) - Multiple instruction streams are executed simultaneously;
- *Vector Data Parallelism* (VDP) - The same operation is performed simultaneously on arrays of elements.

There are many reasons to use parallel computing [4]. First of all, the whole real world has a dynamic nature, i.e. many things happen at a certain time, but in different places at the same time, so this data is very huge to manage and requires more dynamic simulation and modeling. It is parallel computing that ensures the concurrency and organization of complex, large datasets and their management while saving money and time. It also provides efficient use of hardware resources and real-time system implementation.

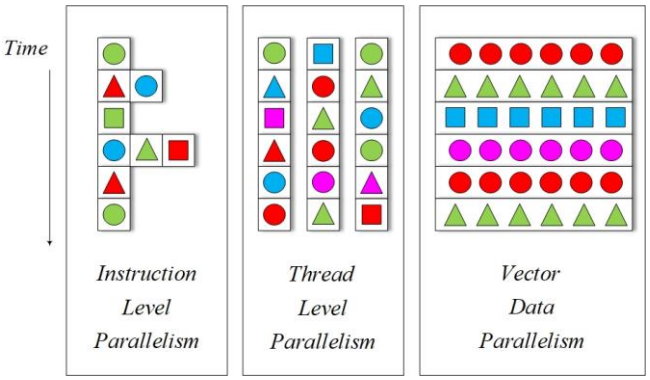


Fig. 22 Differences in execution among ILP, TLP and VDP

Parallel computing finds application in many areas of science and engineering, then in databases and data mining, in real-time system simulations, as well as in advanced graphics, augmented reality and virtual reality.

In addition to a number of advantages, there are some limitations of parallel computing. The main problem is the difficulty in achieving communication and synchronization between multiple subtasks and processes. Also, algorithms or programs must be provided with low coupling and high cohesion as well as the possibility that they can be handled in a parallel mechanism. Developers must be experts and technically skilled in order to be able to effectively code a program based on parallelism.

11. CONCLUSION

Nowadays, the microprocessor represents one of the most complex applications of the transistor, with well over 10 billion transistors of the most powerful microprocessor. In fact, throughout its 50 years of evolution period, the microprocessor has always used the technology of the day. The intention to permanently increase performance has led to rapid technological improvements that have made it possible to build more complex microprocessors. Advances in semiconductor fabrication processes, computer architecture and organization, as well as CMOS IC VLSI design methodologies, were all needed to create today's microprocessor. The development of microprocessors since 1971 has been aimed at (*a*) improving architecture, (*b*) improving instruction set, (*c*) increasing speeds, (*d*) simplifying power requirements [57], [58] and (*e*) embedding more and more memory space and I/O facilities in the same chip (using single chip computers).

This paper discusses first, fifty years of microprocessor history and its generations. Then it describes the benefits of switching from non-pipelined processor to single core

pipelined processor, and switching from single core pipelined and superscalar processor to multicore pipelined and superscalar processor. Finally, it presents the design of a multicore processor. The transition from single-core to multi-core is inevitable because past techniques for accelerating processor architectures that do not modify the basic Von Neumann computer model, such as pipelining and superscalar, encounter strong limits. The question is, why have multi-core machines become so widespread in the last decade? According to Moore's Law, the density of transistors doubles approximately every 18 months [59], and according to Dennard's scaling, the power density of transistors is constant [60]. This has historically corresponded to increase in clock speed of single core machines of approximately 30% per year since the mid-1970s [61]. However, since the mid-2000s, Dennard scaling has no longer been maintained due to physical hardware limitations, and therefore, there has been a need for new mechanisms. To improve performance, hardware vendors have focused on developing processors with multiple cores [62]. As a result, the microprocessor industry is moving towards multicore architectures. However, the full potential of these architectures will not be exploited until the software industry fully accepts parallel programming. Multiprocessor programming is much more complex than programming single processor machines and requires an understanding of new algorithms, computational principles, and software tools. Only a small number of developers currently master these skills. There are many techniques that can be used to facilitate the transition to multicore processors, but to take full advantage of the potential offered by such systems, some form of parallel programming will always be needed [4]. Multicore technology has become ubiquitous today with most personal computers and even mobile phones [63], so writing parallel programs is crucial to achieving scalable performance and enabling large-scale data processing. In addition, to take full advantage of multicore technology, software applications must be multithreaded. The total work to be performed must be able to be distributed among the execution units of a multicore processor in such a way that they can execute at the same time. In order to consider multithreading in more detail, it is first necessary to understand parallel hardware and parallel computing [44]. Finally, this paper provides some details on how to implement hardware parallelism in multicore systems.

Acknowledgement: *This work was supported by the Serbian Ministry of Education and Science, Project No TR-32009 – "Low power reconfigurable fault-tolerant platforms".*

REFERENCES

[1] D. Patterson and J. Hennessy, *Computer Architecture: A Quantitative Approach*, 6th Ed., Morgan Kaufmann, 2017.
[2] J.-L. Baer, *Microprocessor Architecture: From Simple Pipelines to Chip Multiprocessors*, Cambridge University Press, 2009.
[3] Y. Solihin, *Fundamentals of Parallel Multicore Architecture*, Chapman & Hall/CRC, 2015.
[4] R. Kuhn and D. Padua, *Parallel Processing*, 1980 to 2020", Morgan & Claypool, 2021.
[5] M. Stojčev, *Microprocessor architectures I part*, in serbian, Elektronski fakultet Niš, 2004
[6] B. Parhami, *Computer Architecture: From Microprocessors to Supercomputers*, Oxford University Press, 2005
[7] Semiconductor Industry Association, *International Technology Roadmap for Semiconductors (ITRS), 2013 edition*, 2013
[8] K. Olukotun, L. Hammond, and J. Laudon, *Chip Multiprocessor Architecture: Techniques to Improve Throughput and Latency*, Morgan & Claypool, 2007

[9] R. V. Mehta, K. R. Bhatt and V. V. Dwivedi, "Multicore Processor Challenges – Design Aspects", J. Emerg. Technol. Innov. Res. (JETIR), vol. 8, no. 5, pp. C171-C174, May 2021.
[10] A. Gonzalez, F. Latorre and G. Magklis, *Processor Microarchitecture: An Implementation Perspective*, Morgan & Claypool, 2011.
[11] M. Stojčev and P. Krtolica, *Computer systems: Principle of digital systems*, in serbian, Elektronski fakultet Niš i Prirodno-matematički fakultet Niš, 2005.
[12] "Microprocessor Chronology", av.at. https://en.wikipedia.org/wiki/Microprocessor_chronology, last access 28.03.2022.
[13] M. Stojčev, *Contemporary 16-bit microprocessors, Vol. I*, in serbian, Naučna knjiga, Beograd, 1988.
[14] M. Stojčev, *Contemporary 16-bit microprocessors, Vol. II*, in serbian, Naučna knjiga, Beograd, 1988.
[15] M. Stojčev, *Contemporary 16-bit microprocessors, Vol. III*, in serbian, Naučna knjiga, Beograd, 1988.
[16] M. Stojčev, *RISC, CISC and DSP processors*, in serbian, Elektronski fakultet Niš, 1997.
[17] M. Stojčev, Branislav Petrović, *Architectures and programming microcomputer systems based on processor family 80x86*, in serbian, Elektronski fakultet Niš, 1999.
[18] D. Patterson and J. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*, 5th Ed., Morgan Kaufmann, 2014.
[19] Y. Etsion, "Computer Architecture Out-of-order Execution", av.at. https://iis-people.ee.ethz.ch/~gmichi/asocd/addinfo/Out-of-Order_execution.pdf, last access 28.03.2022.
[20] "Superscalar Processors", av. at. https://www.cambridge.org/core/terms, last access. 28.03.2022.
[21] M. Stojčev and T. Nikolić, *Pipeline processing and scalar RISC processor*, in serbian, Elektronski fakultet Niš, 2012.
[22] M. Stojčev and T. Nikolić, *Superscalar and VLIW processors*, in serbian, Elektronski fakultet Niš, 2012 .
[23] Philips Semiconductors, *Introduction to VLIW Computer Architecture*, av. at. https://www.isi.edu/~youngcho/cse560m/vliw.pdf. Last access 28.03.2022
[24] N. P. Jouppi and D. W. Wall, "Available Instruction Level Parallelism for Superscalar and Superpipelined Machines", WRL Research Report 89/7, av. at. https://www.hpl.hp.com/techreports/Compaq-DEC/WRL-89-7.pdf, last access 28.03.2022
[25] C. E. Kozyrakis and D.A. Patterson, "Scalable Vector Processors for Embedded System", *IEEE Micro*, vol. 23, no. 6, pp. 36– 45, Nov.-Dec. 2003.
[26] E. Aldakheel, G. Chandrasekaran and A. Kshemkalyani, "Vector Procesors", av. at. https://www.cs.uic.edu/~ajayk/c566/VectorProcessors.pdf, last access 29.03.2022
[27] C. Lomont, "Introduction to Intel® Advanced Vector Extensions", av. at. https://hpc.llnl.gov/sites/default/files/intelAVXintro.pdf, last access 29.03.2022.
[28] M. Stojčev, E. Milovanović and T. Nikolić, *Multiprocessor systems on chip*, in serbian, Elektronski fakultet Niš, 2012.
[29] J. L. Lo and S. J. Eggers, "Improving Balanced Scheduling with Compiler Optimizations that Increase Instruction-Level Parallelism", av. at. https://homes.cs.washington.edu/~eggers/Research/bsOpt.pdf, last access 29.03.2022.
[30] S. Akhter and J. Roberts, *Multi-Core Programming*, Intel Press, 2006.
[31] G. Koch, "Intel's Road to Multi-Core Chip Architecture", av. at. http://www.intel.com/cd/ids/developer/asmo-na/eng/220997.htm
[32] G. Koch, "Transitioning to multi-core architecture", av.at. www.intel.com/cd/ids/developer/asmo-na/eng/recent /221170.htm, last access 29.03.2022.
[33] M. Brorsson, "Multi-core and Many-core Processor Architectures", Chapter 2 in *Programming Many-Core Chips*, Ed. A. Vajda, Springer, 2011.
[34] M. Zahran, *Heterogeneous Computing: Hardware and Software Perspectives*, ACM Books #26, 2019.
[35] M. Mitić, M. Stojčev and Z. Stamenković, "An Overview of SoC Buses", in *Embedded Systems Handbook, Digital Systems and Aplications*, Ed. V. Oklobdzija, Chapter 7, 7.1- 7.16, CRC Press, Boca Raton, 2008.
[36] J. Rehman, "Advantages and disadvantages of multi-core processors", av. at https://www.itrelease.com/2020/07/advantages-and-disadvantages-of-multi-core-processors/, last access 29.03.2022
[37] J. Shun, *Shared-Memory Parallelism Can Be Simple, Fast, and Scalable*, Morgan & Claypool Pub., 2017
[38] T. Ungerer, B. Rogic and J. Silc, "Multithreaded Processors", *Comput J.*, vol. 45, no. 3, pp. 320–348, 2002.
[39] A. Silberschatz, G. Gagne and P. B. Galvin, "Multithreaded Programming", Chapter 4 in *Operating System Concepts*, 8th Ed., John Wiley, 2009.
[40] DifferenceBetween.com, "Difference Between Multithreading and Multitasking", av.at. https://www.differencebetween.com/difference-between-multithreading-and-vs-multitasking/, last access 29.03.2022.

[41] TechDifferences, "Difference Between Multitasking and Multithreading in OS", av. at. https://techdifferences.com/difference-between-multitasking-and-multithreading-in-os.html, last access 29.03.2022
[42] tutorialspoint, "Multi-Threading Models", av.at https://www.tutorialspoint.com/multi-threading-models, last access 22.03.2022
[43] Wikipedia, "List of Intel Core i7 Processors", av. at https://en.wikipedia.org/wiki/List_of_Intel_Core_i7_processors, last access 29.03.2022
[44] M. Nemirovsky and D. M. Tullsen, *Multithreading Architecture*, Morgan & Claypool, 2013.
[45] O. Mutlu, "Computer Architecture: Multithreading", av. at. https://rmd.ac.in/dept/ece/Supporting_Online_%20Materials/5/CAO/unit5.pdf, last access 22.03.2022
[46] N. Manjikian, "Implementation of Hardware Multithreading in a Pipelined Processor", In Proceedings of the IEEE North-East Workshop on Circuits and Systems, 2006, pp. 145–148.
[47] P. Manadhata, and V. Sekar, "Simultaneous Multithreading", av. at https://www.cs.cmu.edu/afs/cs/academic/class/15740-f03/www/lectures/smt.pdf, last access 29.03.2022
[48] Intel, "Products formerly Nehalem EP", av. at
[49] https://ark.intel.com/content/www/us/en/ark/products/codename/54499/products-formerly-nehalem-ep.html, last access 29.03.2022
[50] K. Hwang and Z. Xu, *Scalable Parallel Computing: Technology, Architecture, Programming*, McGraw-Hill, 1998.
[51] D. Malkhi, *Concurrency: The Works of Leslie Lamport*, ACM Books #29, 2019.
[52] CS4/MSc Parallel Architectures, "Lect. 2: Types of Parallelism", av. at https://www.inf.ed.ac.uk/teaching/courses/pa/Notes/lecture02-types.pdf, last access 29.03.2022
[53] Chapter 3: "Understanding Parallelism", av. at https://courses.cs.washington.edu/courses/cse590o/06au/LNLCh-3-4.pdf, last access 29.03.2022
[54] J. Owens, "Data Level Parallelism", av. at https://www.ece.ucdavis.edu/~jowens/171/lectures/dlp3.pdf. Last access 29.03.2022
[55] A. A. Freitas, S. H. Lavington, "Data Parallelism, Control Parallelism, and Related Issues", in *Mining Very Large Databases with Parallel Processing*, Springer, 2000.
[56] E. I. Milovanović, T. R. Nikolić, M. K. Stojčev and I. Ž. Milovanović, "Multi-functional Systolic Array with Reconfigurable Micro-Power Processing Elements", *Microelectron. Reliab.*, vol. 49, no. 7, pp. 813–820, July 2009.
[57] C. Severance and K. Dowd, *High Performance Computing*, Connexions, Rice University, Houston, Texas, 2012.
[58] G. Nikolić, M. Stojčev, Z. Stamenković, G. Panić and B. Petrović, "Wireless Sensor Node with Low-Power Sensing", *Facta Univ. Ser.: Elec. Energ.*, vol. 27, no 3, pp. 435–453, Sept. 2014.
[59] T. Nikolić, M. Stojčev, G. Nikolić and G. Jovanović, "Energy Harvesting Techniques In Wireless Sensor Networks", *Facta Univ. Ser.: Aut. Cont. Rob.*, vol. 17, no. 2, pp. 117-142, Dec. 2018.
[60] G. E. Moore. "Cramming More Components onto Integrated Circuits", *Electronics*, vol. 38, no. 8, pp. 114–117, April 1965.
[61] R. H. Dennard, F. H. Gaensslen and K. Mai, "Design of Ion-Implanted MOSFET's with Very Small Physical Dimensions", *IEEE J. Solid-State Circuits*, vol. 9, no. 5, pp. 256–268, Oct. 1974.
[62] S. Naffziger, J. Warnock and H. Knapp. "When Processors Hit the Power Wall", In Proceedings of the IEEE International Solid-State Circuits Conference (ISSCC), 2005, pp. 16–17.
[63] S. Borkar and A. A. Chien, "The Future of Microprocessors", *Commun. ACM*, vol. 54, no. 5, pp.67–77, May 2011.
[64] M. D. Hill and M. R. Marty, "Amdahl's Law in the Multicore Era", *IEEE Comput. Mag.*, vol. 41, no.7, pp. 33–38, July 2008.