Decisive Aspects in the Evolution of Microprocessors

DEZSÖ SIMA, MEMBER, IEEE

Contributed Paper

The incessant market demand for higher and higher processor performance called for a continuous increase of clock frequencies as well as an impressive evolution of the microarchitecture. In this paper, we focus on the latter, highlighting major microarchitectural improvements that were introduced to more effectively utilize instruction level parallelism (ILP) in commercial performance-oriented microprocessors. We will show that designers increased the throughput of the microarchitecture at the ILP level basically by subsequently introducing temporal, issue, and intrainstruction parallelism in such a way that exploiting parallelism along one dimension compelled to introduce parallelism along a new dimension as well to further increase performance. In addition, each basic technique used to implement parallel operation along a certain dimension inevitably caused processing bottlenecks in the microarchitecture, whose elimination gave birth to the introduction of innovative auxiliary techniques. On the other hand, the auxiliary techniques applied allow the basic technique of parallel operation to reach its limits, evoking the debut of a new dimension of parallel operation in the microarchitecture. The sequence of basic and auxiliary techniques coined to increase the efficiency of microarchitectures constitutes a fascinating framework for the evolution of microarchitectures, as presented in our paper.

Keywords*—Instruction level parallelism (ILP), intrainstruction parallelism, issue parallelism, microarchitecture, processor performance, temporal parallelism.*

I. INTRODUCTION

Since the birth of microprocessors in 1971, the IC industry has successfully maintained an incredibly rapid increase in performance. For example, as Fig. 1 indicates, the integer performance of the Intel family of microprocessors has been raised over the last quarter of century by an astonishingly high rate of about two orders of magnitude per decade [1].

This impressive development and the underlying innovative techniques have inspired a number of overview papers [2]–[7]. These reviews emphasized either the techniques introduced or the quantitative aspects of the evolution. In contrast, our paper addresses the incentives and implications of the major steps in microprocessor evolution.

Manuscript received March 15, 2004; revised September 1, 2004.

The author is with the John von Neumann Faculty of Informatics, Budapest Tech, Budapest H-1034, Hungary (e-mail: sima@bmf.hu).

Digital Object Identifier 10.1109/JPROC.2004.837627

With maturing techniques the "effective execution width" of microarchitectures [in terms of average number of executed instructions per cycle (IPC)] approaches the available parallelism (in terms of average number of executable IPC). Recently this has given rise to developments in two main directions: 1) *to utilize instruction level parallelism (ILP) more aggressively* by means of more powerful optimizing compilers, and innovative techniques, such as those discussed in Sections V and VI; 2) *to also make use of parallelism at a higher-than-instruction (i.e., thread) level*. This latter approach is marked by *simultaneous multithreading (SMT)* [8]–[10] and *chip multiprocessing (CMP)* [11]–[13]. In our paper, however, we concentrate on the progress achieved at the instruction level in high-performance commercial microprocessors[1] and do not address thread-level parallelism.[2]

The remainder of our paper is structured as follows. In Section II we discuss and reinterpret the notion of absolute processor performance in order to more accurately reflect the impact of different kinds of parallel operations on the performance of the microarchitecture. Based on this discussion, we then identify the main dimensions of the sources of processor performance. In Sections III–VI we review major techniques aimed at increasing processor performance along each of the main dimensions. From these, we point out the basic techniques that have become part of the mainstream evolution of microprocessors. We also identify the potential bottlenecks they induce and highlight the techniques brought into use to cope with these bottlenecks. Section VII summarizes the

[1]We note that computer manufacturers typically offer three product groups: 1) expensive high-performance models designed as servers and workstations; 2) basic models emphasizing both cost and performance; and finally 3) low-cost (value) models emphasizing cost over performance. For instance, Intel's Xeon line exemplifies high-performance models, the company's Klamath, Deschutes, Katmai, Coppermine, and Pentium4 (Willamette and Northwood) cores represent basic models, whereas their Celeron processors are low-cost (value) models. High-performance models are obviously expensive, since all processor and system components must provide a high enough throughput, whereas low cost systems save cost by using less ambitious and less expensive parts or subsystems.

[2]In order to avoid a large number of multiple references to superscalar processors in the text and in the figures, we give all references to superscalars only in Fig. 28.

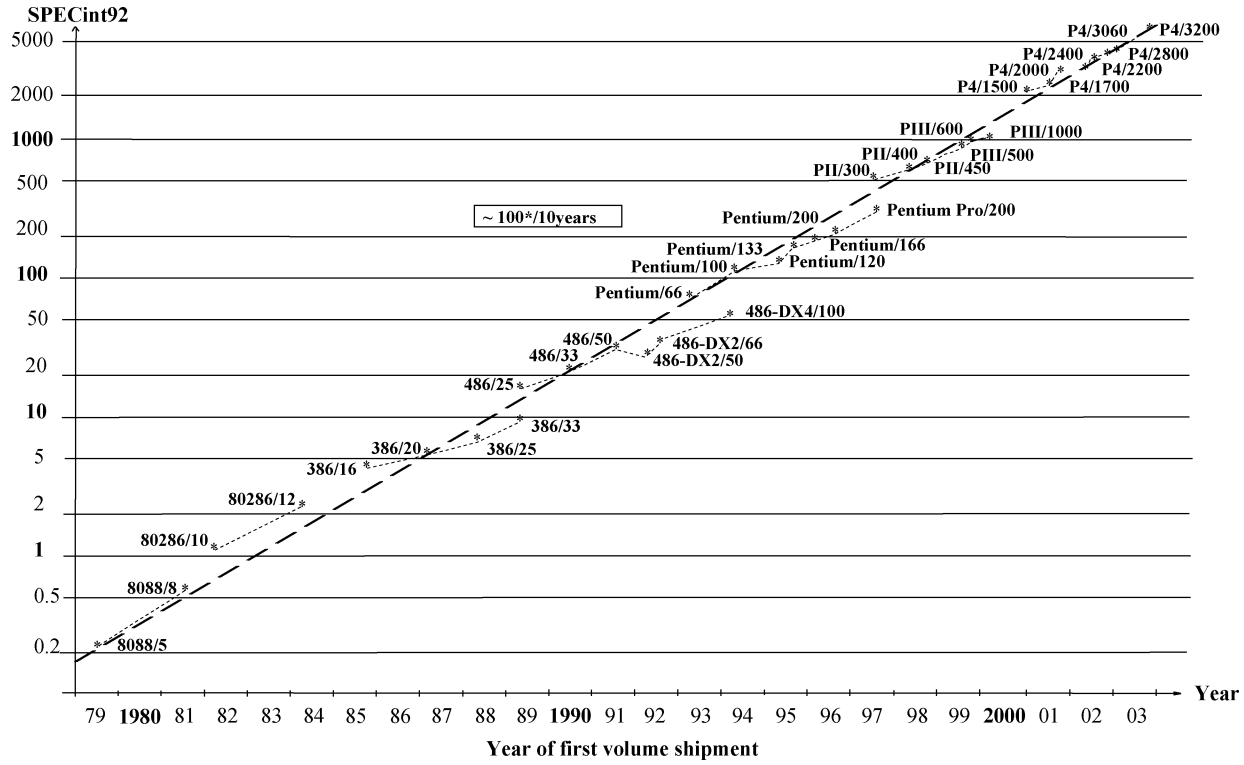


Fig. 1. Increase of the relative integer performance of Intel x86 processors over time.

main evolutionary steps of the microarchitecture of high-performance microprocessors, followed by Section VIII, which sums up the decisive aspects of this evolution.

II. Design Space of Increasing Processor Performance

The results supplied by today's industry standard benchmarks, including the SPEC benchmark suite [14]–[17], Ziff–Davis' CPUmark [18] as well as BAPCo's SYSmark [19], are all *relative performance measures*. This means that they give an indication of how fast a processor will run a set of applications under given conditions in comparison to a reference installation. These benchmarks are commonly used for processor performance comparisons and in papers discussing quantitative aspects of the evolution. We note that the widely used performance metrics of Henessy and Patterson that reflects the reciprocal value of the CPU time ($1/\text{CPU}_{\text{time}} = f_C * \text{IPC/Instruction count}$) [20] is a relative performance metric as well. (For the interpretation of IPC, see the subsequent paragraphs.)

Unlike relative performance measures, *absolute processor performance* (P_{pa}) is usually interpreted as the average number of instructions executed by the processor per second. This score is typically given in units like "million instructions per second" (MIPS), "giga instructions per second" (GIPS), "million floating point instructions per second" (MFLOPS) or "giga floating point instructions per second" (GFLOPS). Earlier synthetic benchmarks like Whetstone [21] or Dhrystone [22] also supplied scores given as absolute measures.

P_{pa} can be expressed as the product of clock frequency (f_C) and the average number of IPC, as follows:

$$P_{\text{pa}} = f_c * \text{IPC} \quad (\text{instructions/s}). \tag{1}$$

IPC is also designated as the *per cycle throughput* and may be interpreted as the *execution width* of the processor (P).

Absolute measures are appropriate for discussing the performance potential of processors, but they are not suitable for comparing processor lines with different instruction set architectures (ISA). The reason is that instructions from different ISAs do not necessarily perform the same amount of computation. In these cases, a valid performance comparison needs relative performance measures. However, as we focus on the evolution of microarchitectures, our paper is based on discussing *absolute* processor performance.

First, in order to identify the contribution of different sources of parallelism within the microarchitecture, let us express IPC with internal operational parameters of the microarchitecture. As shown in the Appendix, IPC can be expressed by two internal operational parameters, as follows:

$$\text{IPC} = \frac{\text{IPII}}{\text{CPII}}. \tag{2}$$

In (2), CPII and IPII are interpreted as follows:

Cycles Per Issue Interval (CPII) denotes the average length of issue intervals in clock cycles. As detailed in the Appendix, *issue intervals* are subsequent segments of the

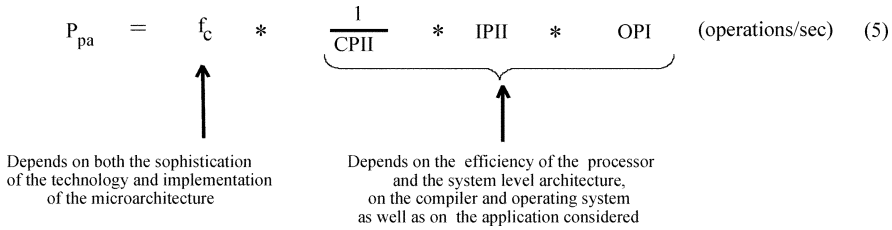


Fig. 2. Constituents of processor performance.

execution time of a program such that each issue interval begins at a clock cycle when the processor issues[3] at least one instruction and ends when the next issue interval begins. We note that for traditional sequential processors, CPII equals the average execution time of the instructions expressed in number of cycles, with CPII $\gg 1$. On the other end, for pipelined or superscalar processors, ideally, CPII $= 1$, since such processors are assumed to issue instructions at each clock cycle. Finally, for recent superscalars, usually CPII > 1, but can roughly be approximated by one. We point out that CPII reflects the *temporal parallelism* of instruction processing.

Instructions per Issue Interval (IPII) designates the average number of instructions issued per issue interval. Obviously, for scalar processors IPII $= 1$, whereas for superscalars typically $1 <$ IPII $< n_{\text{ir}}$, where n_{ir} is the issue rate of the processor. We point out that for CPII $= 1$ (clock cycle): IPII $=$ IPC. Clearly, IPII reflects the *issue parallelism*.

We note that authors usually refer to the notion of IPC (or CPI, which equals 1/IPC) when discussing processor performance. The benefit of using the notions IPII and CPII instead of IPC or CPI (while IPC $=$ IPII/CPII) is being able to identify two sources of parallelism (that is, temporal and issue parallelism) separately, whereas IPC (or CPI) amalgamates these constituents. Moreover, the resulting formal expression of processor performance highlights consecutive steps of processor evolution, as discussed later.

Furthermore, as the inclusion of multioperation instructions, such as single instruction, multiple data (SIMD) instructions, into the ISA has become a major trend, it is appropriate to reinterpret the notion of absolute processor performance, *while taking into account the average number of data operations* processed per cycle (designated as OPC) rather than the average number of *instructions* processed per cycle (IPC). If we denote the number of data operations executed by the instructions on average by *OPI (operations per instructions)*, we get for OPC

$$\text{OPC} = \text{IPC} * \text{OPI}. \tag{3}$$

While executing instructions of a traditional ISA, ideally we assume OPI $= 1$. Practically, however, OPI < 1 due to the existence of instructions that do not manipulate data, such as control transfer instructions (CTIs). For ISAs including multioperation instructions such as SIMD instructions, and for very large instruction word (VLIW) architectures, we expect OPI > 1, as detailed in Section VI. We point out that OPI reveals the *intrainstruction parallelism*.

With expressions (2) and (3), the average number of operations executed per cycle (OPC) is

$$\text{OPC} = \underbrace{\frac{1}{\text{CPII}}}_{\text{Temporal parallelism}} * \underbrace{\text{IPII}}_{\text{Issue parallelism}} * \underbrace{\text{OPI}}_{\text{Intrainstruction parallelism.}} \tag{4}$$

In fact, (4) is very telling for two reasons: 1) it reveals the dimensions of parallelism utilized in microarchitectures and 2) its given sequence reflects the sequence of their introduction along the main road of the evolution of microarchitectures.

Finally, absolute processor performance, interpreted as the average number of operations executed per second (P_{pa}) yields (5) (see Fig. 2).

As far as the clock frequency is concerned, we point out that the *maximum clock frequency* depends on both the *sophistication of the fabrication technology* and *implementation of the microarchitecture*, as detailed in the next section.

The remaining three components of processor performance, i.e., the *temporal, issue, and intrainstruction parallelism*, are determined first of all by the *efficiency of the microarchitecture*, but they depend on a number of further factors as well, such as the characteristics of the application and the programming language used, the efficiency of the optimizing compiler, the ISA and the system architecture, including the chipset, the main memory, etc., as well as the operating system. From all the factors mentioned, our paper focuses on the efficiency of the microarchitecture.

Equation (5) (Fig. 2) provides an appealing framework for a discussion of the major possibilities in increasing processor performance. According to (5), the *key possibilities for boosting processor performance* are: 1) increasing the clock frequency and introducing and increasing 2) temporal, 3) issue, and 4) intrainstruction parallelism in the microarchitecture, as summarized in Fig. 3.

In subsequent sections we address each of these possibilities individually.

[3]When modeling processor performance, we understand instruction issue as disseminating instructions from the fetch/decode subsystem for further processing. For an interpretation of the notion instruction issue (designated also as instruction dispatch) in superscalars, see Section V.

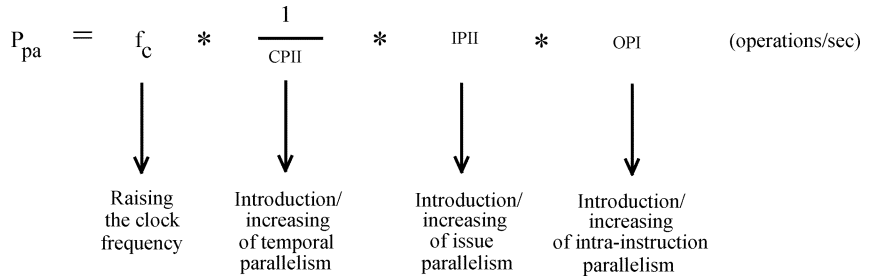


Fig. 3. Main possibilities to increase processor performance.

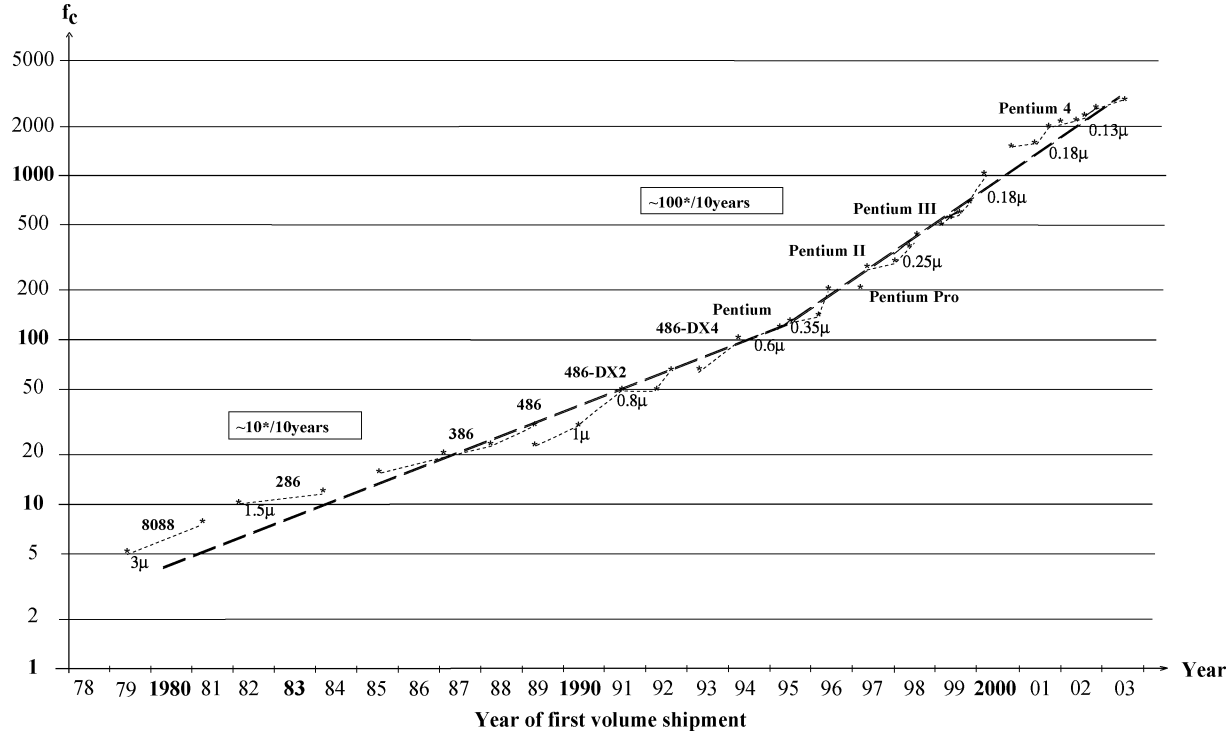


Fig. 4. Historical increase in the clock frequency of Intel x86 processors.

III. Increasing the Clock Frequency and Its Implications

A. Growth Rate of the Clock Frequency of Microprocessors

In synchronous pipelined designs, the *maximum clock frequency* is determined by the *worst case path length* in the pipeline stages. The path length is given as the propagation delay of the useful logic plus the overhead associated with latches, clock skew, and jitter [23]. Here the path length of useful logic follows from the product of the gate delay and the logic depth per pipeline stage, often expressed in terms of "fan-out-of-four" (FO4).[4] The gate delay, in turn, depends on the feature size of the process technology used, whereas the logic depth per pipeline stage results from the implementation of the microarchitecture. So *scaling down process technology* and decreasing logic depth per pipeline stage by means of *increasing pipeline lengths* are the key opportunities to raise clock frequency. In fact, in the course of the evolution of superscalars process technology was scaled down from about 0.5 to 0.13 μm (as Fig. 4 indicates), while designers decreased logic depth from about 50 to 80 FO4 to about 10 to 20 FO4 per pipeline stage [FO4] and increased pipeline length from 4 to 5 stages to about 10 to 20 stages.

However, *longer pipelines* incur *higher misprediction penalties* and *execution unit* (EU) *latencies*. Further on, in shorter pipeline stages of longer pipelines, the overhead portion of the path length becomes higher, which reduces the relative value of decreasing logic depth. Consequently, there is an *optimum design point* for the logic depth and the associated pipeline length as a tradeoff between the impacts mentioned. The optimum design point depends on a number of factors, such as the feature size of the process technology, the microarchitecture chosen, or the application profile and constitutes an interesting recent research area [23], [24]. Concerning this point, we mention that according to published results the optimum logic depth per pipeline stage in recent superscalars amounts to about 6–8 FO4 for integer benchmarks and 6 FO4 for floating point benchmarks [23].

As a consequence of the outlined evolution, over the past two decades clock frequencies of processors (f_c) were tremendously increased, as Fig. 4 illustrates for the Intel x86 line of processors [1].

[4]FO4 designates the delay of an inverter driving four copies of itself.

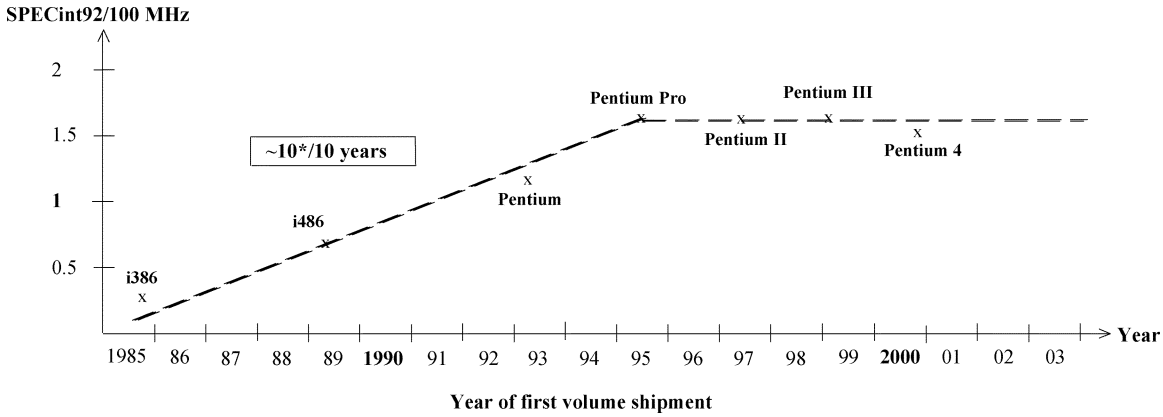


Fig. 5. Increase in the efficiency of the microarchitecture of Intel's x86 line of processors.

As Fig. 4 indicates, the clock frequency of Intel's processors was raised by approximately one order of magnitude per decade until the middle of the 1990s, and subsequently even more rapidly, by about two orders of magnitude per decade. This massive frequency boost was achieved mainly by a continuous downscaling of the process technology, by using longer pipelines, and by improving circuit layouts.

We emphasize that the processor's clock frequency only indicates its performance potential. Actual performance depends on a number of system components, as discussed in the previous section. "Weak" components, such as an inadequate branch handling subsystem of the microarchitecture or long latency caches, may strongly impede performance.

Since processor performance may be increased basically either by raising the clock frequency or by increasing the efficiency of the microarchitecture or both (see Fig. 2), Intel's example of how it increased the efficiency of the microarchitecture in its processors is worth a discussion.

As Fig. 5 shows, the overall efficiency (performance at the same clock frequency) of Intel processors [1] was raised between 1985 and 1995 by about an order of magnitude. During this period, both the clock frequency and the efficiency of the microarchitecture were increased approximately ten times per decade, resulting in a performance boost of approximately two orders of magnitude per decade. However, after the introduction of the Pentium Pro (and until the arrival of the Pentium4), Intel continued to use basically the same processor core in all of its Pentium II and Pentium III processors. The enhancements introduced—including multimedia (MM) and three-dimensional (3-D) support [Streaming SIMD Extension (SSE)], doubling the size of both L1 instruction and data caches, etc.—made only a marginal contribution to the efficiency of the microarchitecture in general purpose applications, as reflected in SPEC benchmark figures. Moreover, the successor P4 core also has roughly the same cycle-by-cycle performance, as Fig. 5 demonstrates. So, since the middle of the 1990s, the per-cycle efficiency of Intel's microarchitectures remained basically unchanged. Obviously, this fact indicates a push by Intel to intensify their efforts to raise clock frequency, resulting in a frequency boost of about two orders of magnitude per decade, as Fig. 4 shows.

B. Implications of Increasing the Clock Frequency

When increasing processor performance, either by raising the clock frequency or by increasing the throughput of the microarchitecture or by both, designers are forced *to enhance the system-level architecture* as well in order to avoid introducing new bottlenecks. System-level enhancements predominantly address the bus, memory, and I/O subsystems. Since the evolution of the system-level architecture is a topic of its own, with a complexity comparable to that of the evolution of the microarchitectures, we will not go into details here, but will indicate only a few dimensions of this evolution and refer to the literature given.

1) Enhancing the Bus Subsystem: Higher clock frequencies and more effective microarchitectures call for more bandwidth in the buses connecting the processor to the memory and the I/O subsystem for obvious reasons. This requirement has driven the evolution of front-side processor buses (system buses), general purpose peripheral buses (such as the ISA and the PCI buses), dedicated peripheral buses and ports intended to connect storage devices (IDE/ATA, SCSI standards), video (AGP), audio (AC'97), or low-speed peripherals (USB bus, LPC port etc.). In order to exemplify the progress achieved, Fig. 6 depicts how the data width and the maximum clock frequency of major *general purpose peripheral bus standards* have evolved.

As depicted in the figure, the standardized 8/16-b wide AT-bus, known as the ISA bus (International Standard Architecture) [25], was first extended to provide 32-b data width (this extension is called the EISA bus [26]). The ISA bus was subsequently replaced by the PCI bus and its wider and faster versions, such as PCI versions 2–2.3 [27] and PCI-X versions 1.0 and 2.0 [28]. Fig. 6 demonstrates that the maximum bus frequency was raised at roughly the same rate as the clock frequency of the processors.

2) Enhancing the Memory Subsystem: Higher clock frequencies call for higher memory bandwidth and reduced load-use latencies (the time needed to use requested memory data). In addition, more efficient microarchitectures (in terms of higher IPC values) aggravate the memory bandwidth requirements. Designers of memory subsystems responded by impressive innovations along many dimensions, including: 1) the use of enhanced main memory components, such as

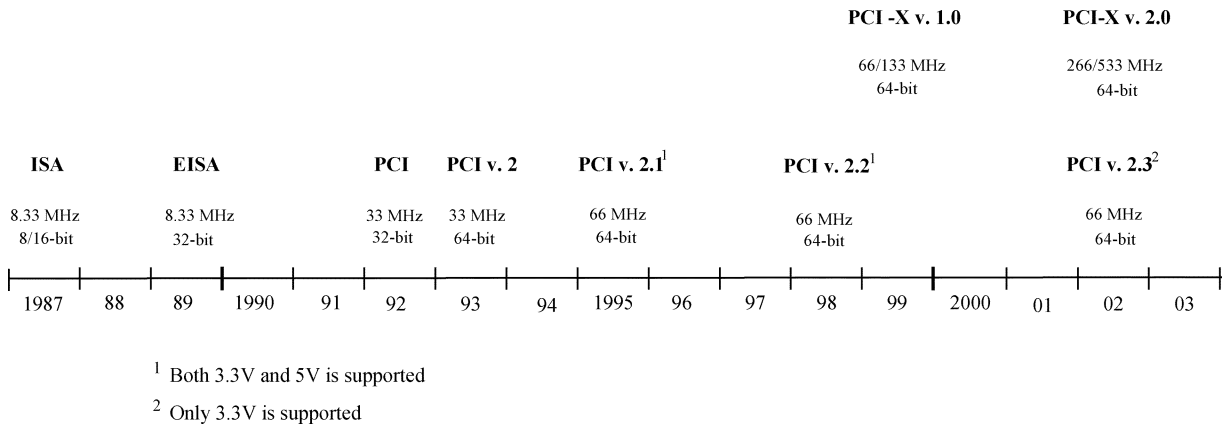


Fig. 6. Evolution of major general purpose peripheral buses.

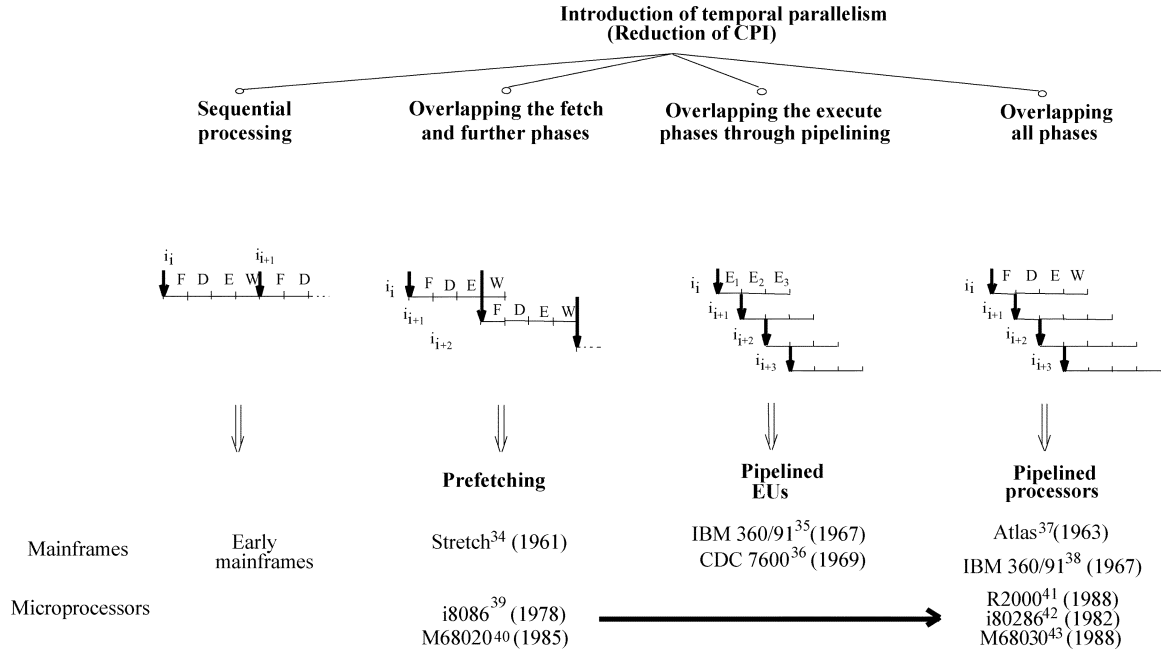


Fig. 7. Main approaches to achieve temporal parallelism. (F: fetch phase, D: decode phase, E: execute phase, W: write phase.) The superscripts following machine or processor designations indicate references. Dates in this and all subsequent figures refer to the year of first shipment (in the case of mainframes) or that of first volume shipment (in the case of microprocessors).

fast page mode DRAMs (FPM DRAMs), extended data out DRAMs (EDO DRAMs), synchronous DRAMs (SDRAMs), Rambus DRAMs (RDRAMs), and direct Rambus DRAMs (DRDRAMs) [29]; 2) introducing and enhancing caches through improved cache organization, increasing the number of cache levels, implementing higher cache capacities, using directly connected or on-die level 2 caches, etc. [30], [31]; and 3) introducing latency reduction or hiding techniques, such as software or hardware controlled data prefetch [32], lockup-free (nonblocking) caches, out-of-order loads, store forwarding, etc., as outlined later in Section V-D4e.

3) Enhancing the I/O Subsystem: Concerning this point, we again do not delve into details, but refer to the spectacular evolution of storage devices (hard disks, CD-ROM drives, etc.) in terms of storage capacity and access time as well as the evolution of display devices in terms of their resolution, etc., in order to better support more data-intensive recent applications such as multimedia, 3-D graphics, etc. For details, see, e.g., [33].

IV. Introduction and Evolution of Utilizing Temporal Parallelism

A. Overview of Possible Approaches to Introduce Temporal Parallelism

A traditional von Neumann processor executes instructions in a strictly sequential manner, as indicated in Fig. 7. For *sequential processing*, the average length of the issue intervals given in terms of clock cycles (CPII) equals the average execution time of instructions (CPI), i.e., CPII = CPI. Usually CPII $\gg$ 1 (for example, CPII is equal to four in the figure).

Assuming a given ISA, CPII can be reduced by introducing some form of overlapped processing like pipelining—that

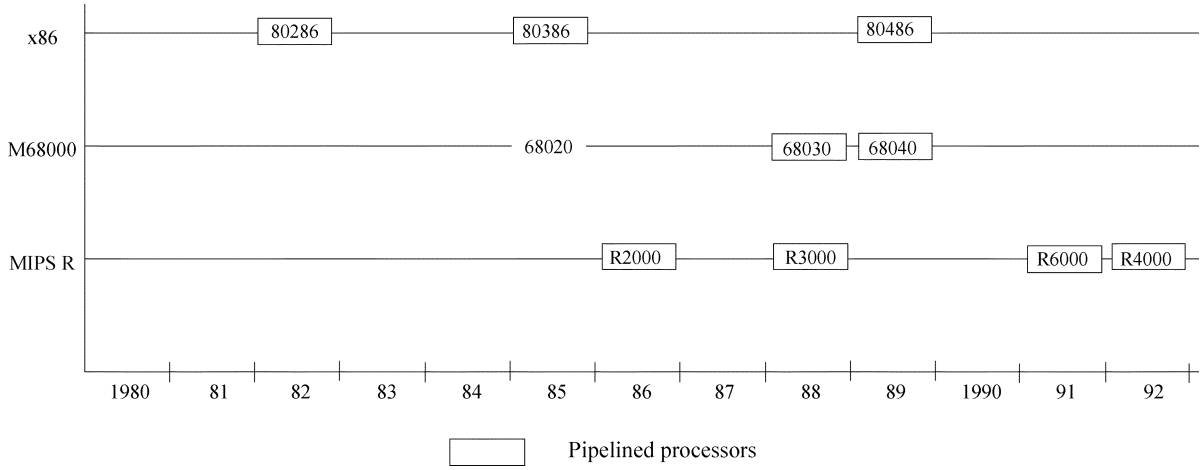


Fig. 8. Introduction of pipelined microprocessors.

is, by utilizing temporal parallelism. In this sense, CPII reflects *the extent of temporal parallelism* achieved in instruction processing, as already emphasized in Section II.

Basically, the processing of subsequent instructions may be overlapped in three ways: 1) by overlapping the fetch phases and the last processing phase(s) of the preceding instruction; 2) by overlapping the execute phases of subsequent instructions processed in the same EU using pipelined EUs; or 3) overlapping all phases of instruction processing using pipelined processors, as shown in Fig. 7. In the figure, we represent issued instructions by arrows. Furthermore, we assume that instructions are processed in four subsequent phases, called the Fetch (F), Decode (D), Execute (E), and Write (W) phases.

1) *Overlapping the fetch phases and the last phase(s) of the preceding instruction* is called *prefetching*, a term coined in the early days of computing [34]. If the processor overlaps the fetch phases with the write-back phases, as indicated in Fig. 7, the average execution time is reduced by one cycle compared to fully sequential processing. However, the execution of CTIs lessens the achievable performance gain of instruction prefetching to less than one cycle per instruction, since CTIs divert instruction execution from the sequential path and, thus, render the prefetched instructions obsolete.
2) The next possibility is *to overlap the execution phases of subsequent instructions* processed in the same pipelined EU. *Pipelined EUs*, introduced in mainframes in the late 1960s [35], [36], execute a new instruction ideally in every new clock cycle, provided that subsequent instructions are independent. Clearly, pipelined EUs are very effective in processing vectors.
3) Finally, the ultimate solution to exploit temporal parallelism is *to extend pipelining to all phases of instruction processing*, as indicated in Fig. 7. *Pipelined processors* arrived in the 1960s [37], [38]. Ideally they accept a new instruction for execution in every clock cycle, i.e., the issue intervals become one cycle long $(\text{CPII} = 1)$. However, CTIs and dependencies between subsequent instructions impede instruction processing, as discussed in Section IV-B3.

We note that even in pipelined instruction processing, the execution phase of certain complex instructions, such as division or square root calculation, is not pipelined in order to reduce the complexity of the implementation. Due to this fact and the occurrence of CTIs as well as dependencies between subsequent instructions, in real pipelined processors the average issue intervals are longer than one cycle $(\text{CPII} > 1)$.

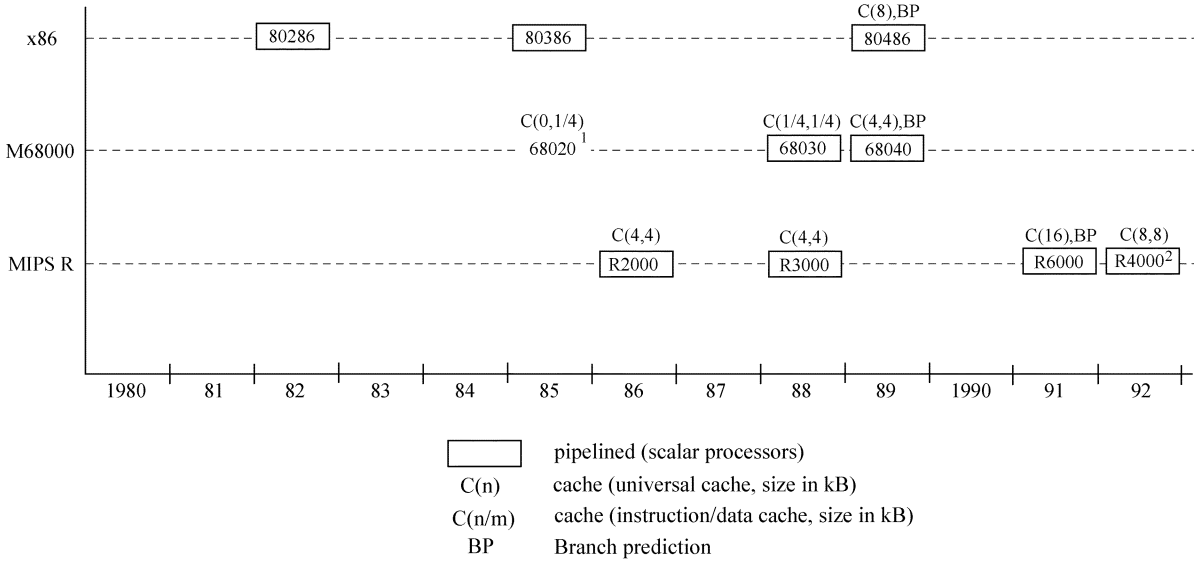
Although both prefetching and overlapping of the execution phases of subsequent instructions already represent a partial solution to parallel execution, processors providing these techniques alone are usually not deemed to belong to the ILP processors. In contrast, pipelined processors are considered to be a class of the *ILP processors* category.

In microprocessors, prefetching arrived two decades later than in mainframes, with the advent of 16-b micros [39], [40]. Subsequently, pipelined microprocessors emerged and became the main road of the evolution because of their highest performance potential among the alternatives discussed [41]–[43]. They came into widespread use in the second half of the 1980s, as shown in Fig. 8. We point out that pipelined microprocessors represent the second major step in the path of microprocessor evolution. In fact, the very first step of their evolution was increasing the word length gradually from 4 to 16 b and introduce a new ISA for each new processor, as exemplified by Intel's 4004, [44], 8008, 8080, and 8086 [45] processors. For this reason, we discuss the evolution of the microarchitecture of microprocessors beginning with 16-b designs.

B. Implications of the Introduction of Pipelined Instruction Processing

1) Overview: Pipelined instruction processing calls for higher memory bandwidth and smart processing of CTIs, as detailed below. The basic techniques needed to avoid processing bottlenecks due to the requirements mentioned above are caches and speculative branch processing.

2) The Demand for Higher Memory Bandwidth and the Introduction of Caches: Ideally, a pipelined processor



[1] The 68020 prefetches instructions and accesses the instruction cache and main memory simultaneously with instruction execution
[2] Branch prediction was planned but due to implementation constraints not realized

Fig. 9. Introduction of caches and branch prediction.

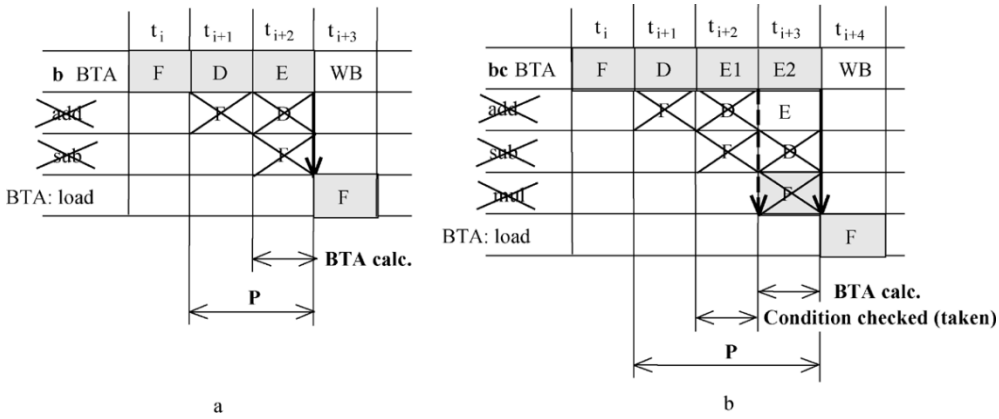


Fig. 10. Pipelined processing of unconditional and conditional branches assuming straightforward execution. (a) Unconditional branches. (b) Conditional branches.

executes a new instruction in every new clock cycle. Accordingly, pipelined instruction processing requires higher memory bandwidth for both instructions and data than sequential processing. As the memory is typically slower than the processor, the increasing memory bandwidth requirement of pipelined instruction processing accelerated and inevitably brought about the introduction of *caches*, an innovation pioneered in the IBM 360/85 [46] in 1968. With caches, frequently used program segments (cycles) can be held in fast memory, which allows instruction and data requests to be served at a higher rate. In microprocessors, caches came into widespread use in the second half of the 1980s, essentially along with the introduction of pipelined instruction processing (see Fig. 9). As the performance of microprocessors is increasing more rapidly than that of main memories (by a rate of about two orders of magnitude per decade, as indicated in Section I), there is a continuous demand to raise the performance of the memory subsystem. As a consequence, the enhancement of caches and their connection to the processor has remained one of the focal points of microprocessor evolution for more than one decade now.

3) Performance Degradation Due to CTIs and the Introduction of Branch Prediction: Through diverting the sequential flow of instructions to the branch target path either unconditionally or conditionally, CTIs disrupt the smooth supply of instructions. This impedes the performance of the instruction fetch subsystem and, consequently, that of the entire pipeline, as shown below.

Let us first consider the required execution steps of an *unconditional CTI* assuming a simple four-stage pipeline, consisting of the F (fetch), D (decode), E (execute), and WB (write-back) stages, as shown in Fig. 10. Without any additional hardware support, branches will be executed in three pipeline cycles, needed to fetch and decode the branch instruction (b BTA) as well as to calculate the branch target ad-

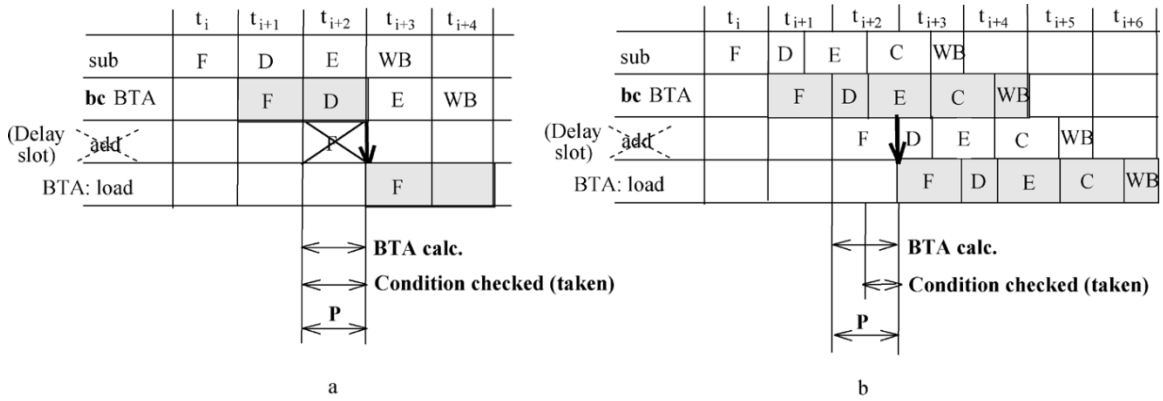


Fig. 11. Pipelined processing of taken conditional branches. In the figure, we assume that the specified condition is already known during checking. (a) In the case of a conventional four-stage pipeline *(examples: Sparc, Am29000)*. (b) In the case of MIPS's sophisticated five-stage pipeline *(R2000, R3000, R42000)*.

dress (BTA). Using this address, the branch target instruction (load) can be fetched in the next pipeline cycle. Meanwhile, however, the pipeline has already fetched two sequential instructions (add, sub) that need to be discarded. As a consequence, two penalty cycles (P) (called pipeline bubbles) arise, as shown in the figure.

The straightforward execution of conditional CTIs needs even one more pipeline cycle (E1) to check the specified branch condition, as demonstrated in Fig. 10(b). Thus, conditional branches will cause a branch penalty of three cycles altogether. Since both straightforward execution schemes discussed above would strongly decrease performance, designers typically *speed up accessing the taken path* by augmenting scalar pipelines by dedicated hardware.

One possibility to speed up accessing the taken path is to augment the pipeline by both a dedicated address adder and a logic comparator and thus perform both condition checking and BTA calculation in parallel with decoding (see Fig. 11). In this way the BTA becomes available as early as at the end of the D cycle, allowing access to the branch target path with a penalty of only one cycle. However, a drawback of this scheme (used, e.g., in the SPARC or Am29000 processors) is that the specified condition needs to be known already at the beginning of the D cycle. So conditions which are set by instructions immediately preceding the CTI cannot be evaluated without a pipeline blockage of at least one clock cycle.

In order to avoid the deficiency mentioned above, MIPS conceived an unconventional five-stage pipeline consisting of three full and two half-cycles, as shown in Fig. 11(b). In this pipeline arrangement (used in the processors R2000, R3000, and R4200), condition checking is deferred to the first half of the execute cycle (E cycle), so results generated by the preceding instruction can become available for checking in due time as well. The other side of this arrangement, however, is that it results in a time critical path for evaluating the branch condition in a half pipeline cycle (E cycle). Due to this constraint, the above-discussed pipeline structure is not appropriate for higher clocked pipelines, such as MIPS' R4000, where already a full pipeline cycle is needed for condition checking.

A supplementary possibility to increase the efficiency of pipelined processing of branches is to utilize the else wasted cycle(s) behind a CTI by means of *delayed branching*. The *branch delay slot* concept originates from IBM's 801 reduced instruction set computer (RISC) machine and is widely used in RISC processors. The delay slot is the instruction slot following the CTI in the program, as indicated in Fig. 11. An instruction placed into the delay slot is (optionally) executed before the branch is performed, whether the branch is taken or not. The compiler or the programmer can fill the delay slot with a useful instruction, reducing the impediments of branch processing. Examples of scalar pipelined processors using the delayed branching concept are the R2000, R3000, R4200, SPARC, and Am29000.

Drawbacks of both early implementation alternatives of speeding up the access of the taken path can be avoided by *branch prediction*, a sophisticated technique introduced in advanced pipelined processors like the i486, the M68040, and the M88100, around 1990. It is worth noting that branch prediction emerged first in complex instruction set computer (CISC) processors, since they do not use delay slots, so increasing their efficiency was of primary interest.

With branch prediction [47]–[50], the processor makes a guess for the outcome of conditional branches (or possibly each branch in order to simplify implementation) usually at the end of the decode cycle, as denoted in Fig. 12(a), then resumes instruction processing along the estimated path. Later, when the specified condition becomes known (typically in the E cycle), it checks the foregoing guess. For a correct prediction, the processor resumes execution; otherwise, it cancels all incorrectly executed processing steps and goes on with execution along the correct path.

We emphasize that pipelined processors introduced a rather straightforward *fixed prediction* that guesses the outcome of all branches usually as always taken and allows the pipeline to resume operation accordingly, i.e., to fetch the branch target instruction in the next cycle [see Fig. 12(a)].

Branch prediction progressed immensely in ILP processors, since both longer pipelines and higher issue rates in-

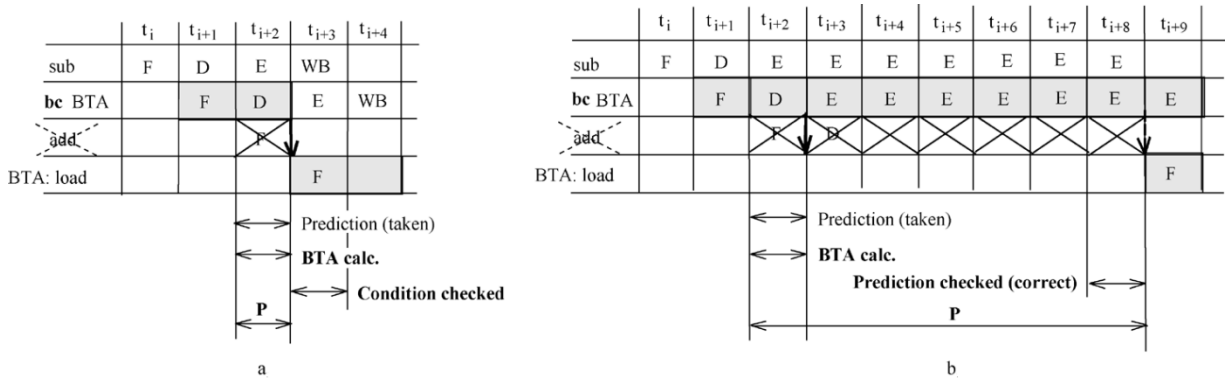


Fig. 12. Pipelined processing of a resolvable and an unresolved taken conditional branch assuming branch prediction. (a) The branch condition is known at the end of the D cycle. (b) The branch condition is not known at the end of the D cycle.

crease the number of instructions to be discarded in the case of a misprediction [49].

Beyond misprediction, *unresolved conditional branches* also cause additional wasted cycles. Unresolved conditional branches are conditional branches whose referenced condition is not yet available at the time of checking, since they refer to, e.g., the result of a long latency operation such as a multiplication or division. If unconditional branches cannot be resolved in the execution stage, the pipeline becomes blocked until the condition becomes known. As a consequence, a number of wasted cycles occur, as shown in Fig. 12(b).

Unresolved conditional branches impede the issue subsystem of both pipelined scalar processors and first-generation superscalars (to be discussed in Section V-C). Only second-generation superscalars, discussed in Section V-D, started to cope with unresolved conditional branches by means of a sophisticated technique, designated as speculative branch processing (see Section V-D4d).

4) Limits of Utilizing Temporal Parallelism: With the introduction of pipelining and its enhancement by both caches and branch prediction (see Fig. 9), the average length of issue intervals (CPII) could be significantly reduced in comparison to sequential processors. However, once designers arranged the debut of both caches and branch prediction, the intrinsic potential of pipelined instruction processing has been exhausted. As CPII $= 1$ marks the absolute limit achievable through temporal parallelism, any further substantial performance increase calls for the introduction of parallel operation along another dimension. There are two possibilities for this: to introduce either issue parallelism or intrainstruction parallelism. Following the evolutionary path of microprocessors, we now first discuss the former alternative.

V. Introduction and Evolution of Utilizing Issue Parallelism

A. Introduction of Issue Parallelism

Issue parallelism, also known as the *superscalar instruction issue* [5], [51], [52], refers to a processor's ability to disseminate multiple instructions per clock cycle from the decode unit for further processing. The peak rate of instructions issued per clock cycle is called the *issue rate* (n_{ir}).

Superscalar instruction issue has been implemented in *superscalar processors* that were introduced after the full potential of pipelined instruction processing became exhausted around 1990. Superscalars rapidly began to dominate all major processor lines, as Fig. 13 shows. Superscalar instruction issue has however significant implications on both the required memory bandwidth and the efficiency of branch processing, as discussed in the next section.

B. Overall Implications of Introducing Superscalar Issue

1) Aggravating the Memory Bottleneck: Unlike pipelined processors that issue at most 1 instruction per cycle for execution, superscalars can issue up to n_{ir} instruction per cycle, as illustrated in Fig. 14. As a consequence, superscalars must be able to fetch roughly n_{ir} times as many instructions and memory data and must store roughly n_{ir} times as much memory data per cycle than pipelined processors. In other words, superscalars require roughly n_{ir} times *higher memory bandwidth* than pipelined processors at the same clock frequency. Given that n_{ir} refers to the width of the superscalar processor (as we will discuss later on in Section V-C-3.), the wider the processor, the more serious the resulting memory bottleneck. Furthermore, since clock frequencies of superscalars are rapidly increasing over time as well (see Fig. 4), superscalars aggravate the memory bottleneck in proportion to their width and their clock rates, in accordance with Section III-B-2.

2) Aggravating the Branch Handling Bottleneck: Superscalar issue also impacts branch processing for two reasons. First, superscalar instruction issue with an issue rate of n_{ir} raises the frequency of branches per cycle roughly n_{ir} times compared to scalar pipelined processing. Second, each wasted cycle caused by branches will restrict up to n_{ir} instructions from being issued. Consequently, superscalar processing gives rise to *more sophisticated branch processing* than pipelined processing—the higher n_{ir} (that is, the wider the superscalar processor), the greater the necessary sophistication. Accordingly, branch processing underwent an impressive evolution, while focusing on the following three tasks: (i) increasing the accuracy of branch

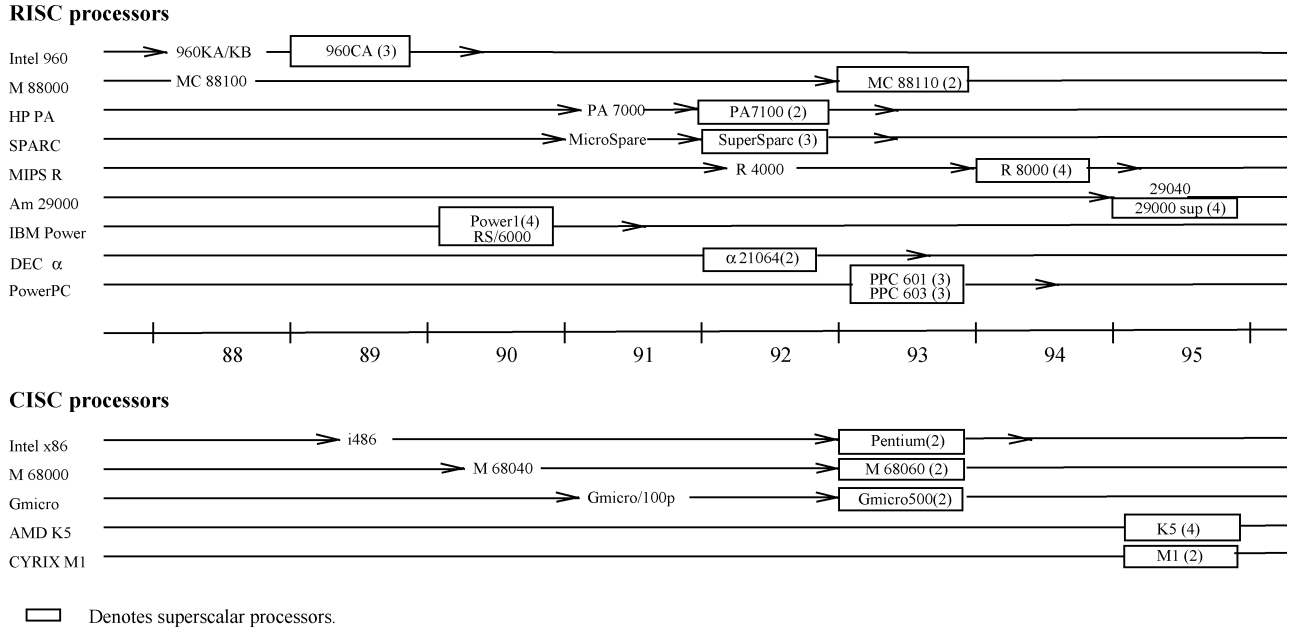


Fig. 13. The appearance of superscalar processors.

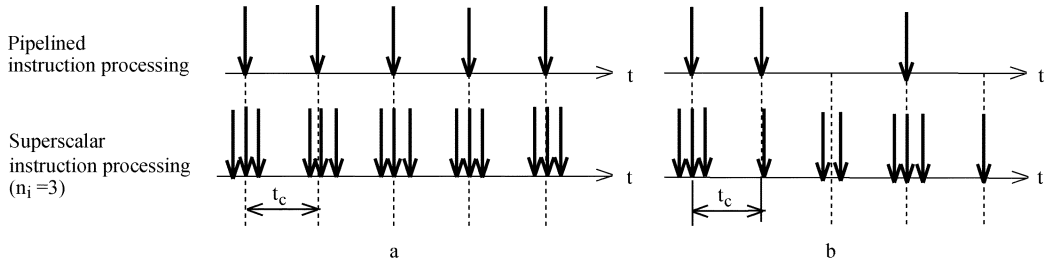


Fig. 14. Contrasting pipelined instruction processing with superscalar processing (arrows indicate issued instructions). (a) Ideal case; (b) in practice.

prediction, (ii) speeding up access to the branch target path, and (iii) handling of unresolved conditional branches. Usually, all these tasks are managed by the fetch subsystem. In Sections V-C and V-D, we highlight key points of the progress achieved so far, but for details we refer to [49] and [55].

C. First-Generation Superscalars

1) Overview: Most early superscalars issue instructions in a straightforward manner by forwarding decoded, not dependent, instructions directly (i.e., with no buffering) to available EUs like the Alpha 21064, PA 7100, PowerPC 601, SuperSPARC, or the Pentium. This issue mechanism is called *direct issue* and will be discussed in the next section. Processors using the direct issue scheme share a more or less common set of features, as summarized in Fig. 15. So it is convenient to treat early superscalars employing the direct issue scheme as a distinguished group of processors, called the *first-generation superscalars*. Later, in the middle of the 1990s, they were superseded by the more advanced second-generation superscalars, as discussed in Section V-D.

First-generation superscalars usually have "thin" cores, typically capable of issuing two to three RISC instructions/cycle or two CISC instructions/cycle. Usually, their fetch subsystems perform static branch prediction to speed up accessing the branch target path. Static prediction is based on some features of the code (such as the operation code or sign of the displacement). It provides a higher accuracy than fixed prediction, typically used in advanced scalar pipelined processors. Their cores make use of the direct issue scheme, as discussed below, and yet do not address unresolved conditional branches by speculative branch processing (to be discussed in Section V-D4b), but tolerate issue blockages until the referenced condition becomes known. Further on, they include a two-level cache subsystem in most cases that consists of single-ported split L1 data and instruction caches supported by an off-chip unified L2 cache attached via the system bus. Their data caches usually handle cache misses in a straightforward way, either by rejecting any further requests after a cache miss (blocking data caches) or by continuing to service access requests until another subsequent cache miss occurs (nonblocking data caches allowing a single pending cache miss). From among the features mentioned above, we will only cover the direct issue

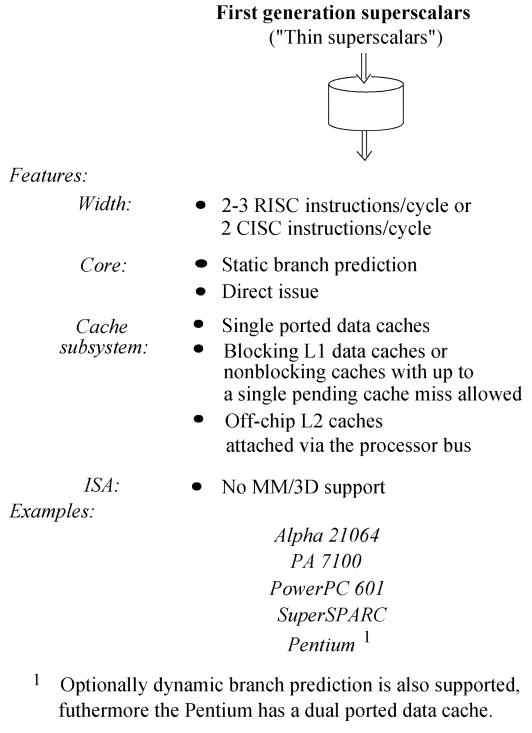


Fig. 15. Main features of first-generation superscalars.

scheme in detail in the following section due to its relevance to the evolution of superscalars.

2) Principle of the Direct Issue Scheme: When using *direct issue*, the processor issues decoded instructions immediately, i.e., without buffering, to the EUs, as shown in Fig. 16.

The issue process itself can best be viewed by introducing the concept of the *instruction window* (issue window). Conceptually, the instruction window is a buffer that contains the instructions to be issued at any given clock cycle. Assuming direct issue it is n_{ir} instructions wide. The instruction window is filled from the instruction buffer and is depleted while issuing instructions. The instructions held in the window are decoded and checked for dependencies. Executable instructions are issued from it directly to free EUs, whereas dependent instructions remain in the window until existing data-, control-, or resource dependencies are resolved. Variants of this scheme differ on two aspects: 1) whether instructions are issued from the window in order or out of order and 2) whether the window is immediately refilled after issuing one or more instructions or only when it becomes empty [49], [52].

In Fig. 16 we demonstrate the direct issue scheme for an issue rate of three ($n_{\mathrm{ir}} = 3$) with the following two assumptions: 1) the processor issues instructions in order, meaning that dependent instructions block the issue of all subsequent independent instructions from the window and 2) the processor needs to issue all instructions from the window before shifting it along the instruction stream. Examples of processors that issue instructions this way are the Power1, the PA7100, or the SuperSPARC. In the figure, we assume that in cycle c_i the instruction window holds instructions $i_1 - i_3$. If in cycle c_i instructions i_1 and i_3 are free of dependencies, but i_2 depends on instructions that are still in execution, in cycle c_i only instruction i_1 can be issued, and both i_2 and i_3 will be withheld in the window, since the dependent instruction i_2 blocks the issue of any subsequent instructions. Let us assume that in the next cycle (c_{i+1}) i_2 becomes executable. Then in cycle c_{i+1} instructions i_2 and i_3 will be issued for execution as well and the window becomes empty. Subsequently, in the next cycle (c_{i+2}) the window is shifted along the instruction stream by three instructions while filling it with the next three instructions $(i_4 - i_6)$ and the issue process resumes in a similar fashion.

3) Execution Model of First-Generation Superscalars: As far as the throughput (IPC) of the microarchitecture is concerned, the microarchitecture may best be viewed as a number of subsystems linked together via buffers. Instructions are processed in the microarchitecture as they flow from one subsystem to the other in a pipelined fashion. These subsystems are typically responsible for fetching, decoding and/or issuing, executing, and finally retiring (i.e., completing in program order) instructions. The kind and number of subsystems depends on the microarchitecture in question.

Fig. 17 shows the simplified *execution model of first-generation superscalar RISC processors*. Basically, the microarchitecture consists of a front end and a back end section that are connected by the instruction window. The *front end* includes the fetch subsystem, its task is to "fill" the instruction window.

The instruction window is "depleted" by the *back end* section of the microarchitecture that also takes care of executing the issued instructions. The back end contains the decode, issue, execute, and retire subsystems. The issue subsystem forwards executable instructions from the instruction window to the execute subsystem. The execute subsystem performs the operations required, where referenced register operands are supplied to the EUs from the architectural register file. Finally, executed instructions are completed by the retire subsystem in program order and the generated results are committed either to the architectural register file or to the memory.

Let us now discuss the notions of throughput and the "width" of the microarchitecture in relation to the execution model presented.

Each subsystem has a *maximum throughput* in terms of the maximum number of instructions that may be processed per cycle, designated as the *fetch*, *decode*, *issue*, *execution*, and *retire rate*, respectively, as indicated in Fig. 17. Now, these rates can be interpreted as the *width* of the respective subsystems. Then, obviously, the *width of the entire microarchitecture* is determined by its narrowest subsystem. We mention that this notion is analogous to the notion of the "word length of an ISA or a processor" (see Section V-D5.)

In fact, the width of a subsystem only indicates its *performance potential*. When running an application, subsystems actually have less than maximum throughput, since they usually operate under worse than ideal conditions. For instance, branches decrease the actual throughput of the fetch subsystem, or the actual throughput of the issue subsystem depends on the number of parallel executable instructions that

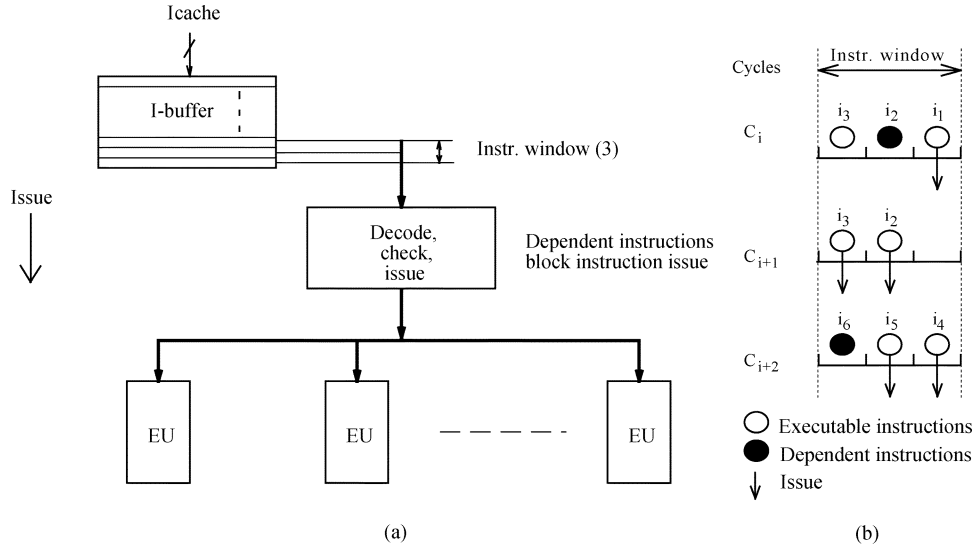


Fig. 16. Principle of the direct issue scheme. (a) Simplified structure of a superscalar microarchitecture that employs the direct scheme and has an issue of three. (b) The issue process.

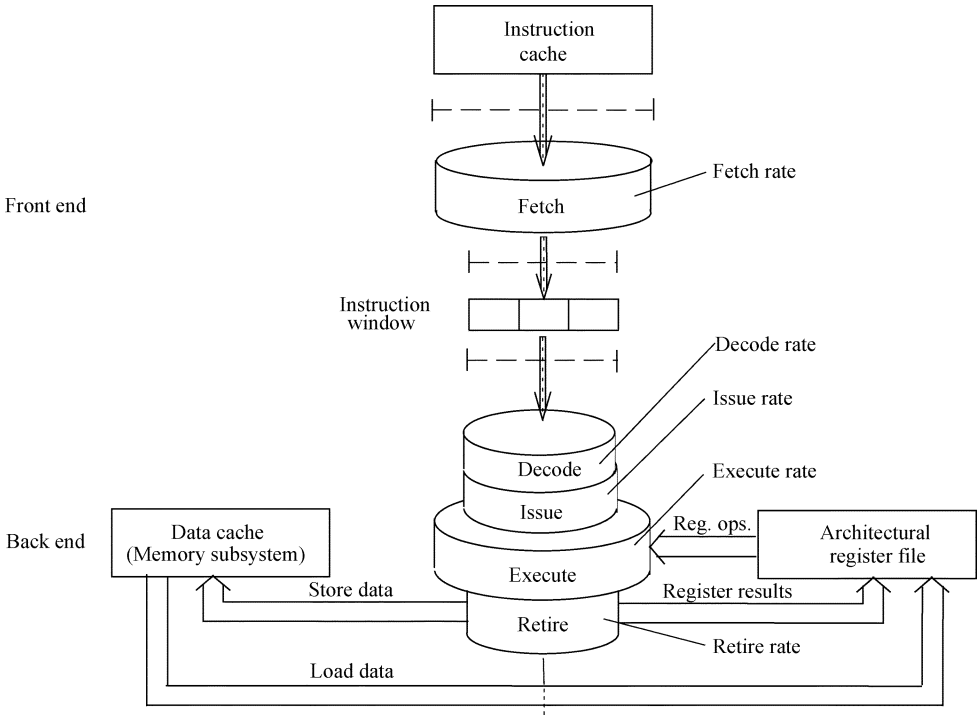


Fig. 17. Simplified execution model of a superscalar RISC processor that employs direct issue.

are available in the window from one cycle to the next. In any application, the subsystem with the smallest throughput will be the *bottleneck* that determines the resulting throughput (IPC) of the entire microarchitecture.

4) Issue Bottleneck of First-Generation Superscalars: Beyond the obvious memory and branch processing bottlenecks of superscalars, discussed in Section V-B, first-generation superscalars suffer also from the *issue bottleneck* caused by the direct issue scheme employed. The reason of this bottleneck is as follows. In each cycle, some instructions in the instruction window are available for parallel execution, while others are not, as they are locked by dependencies until existing dependencies are resolved by finishing the execution of "older" instructions. Since in the direct issue scheme, all data or resource dependencies block instruction issue from the instruction window,[5] the average number of instructions available for parallel execution is actually limited to about two in general purpose applications [56], [57]. Obviously, when the issue subsystem is confined to issuing on average less than about 2 instructions per cycle, the throughput of the entire microarchitecture is also limited to about 2 instructions per cycle, no matter how wide other

[5]Here we assume that control dependencies are handled by branch prediction, but unresolved control dependencies block instruction issue.

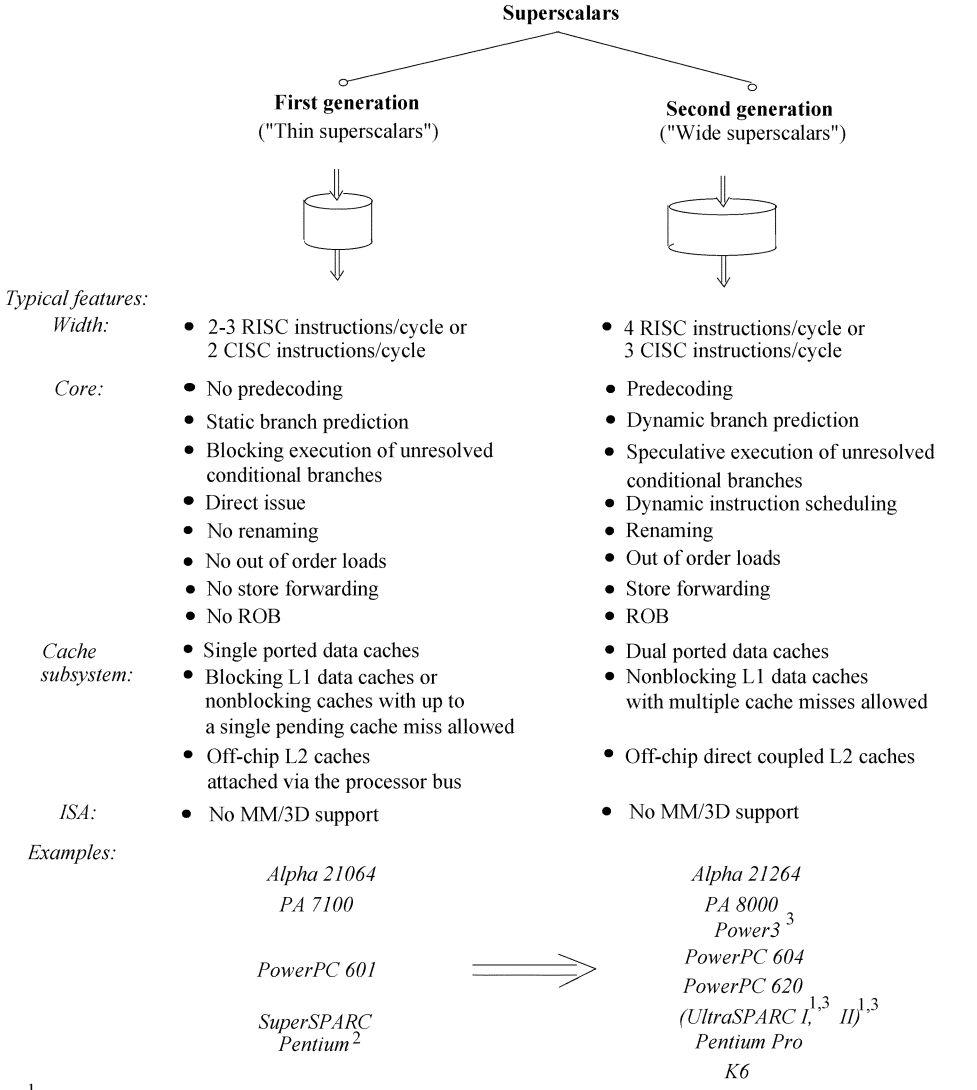


Fig. 18. Contrasting key features of first- and second-generation superscalars.

subsystems of the microarchitecture are. Here we note that due to the constraint mentioned direct issue results usually in actual IPC values of less than one [56], [57]. Consequently, the direct issue scheme leads to an *issue bottleneck* that severely limits the throughput of the microarchitecture.

Due to this restriction, first-generation superscalars are "thin" superscalars, usually having an issue rate of two to three RISC instructions/cycle or two CISC instructions/cycle, as mentioned before and indicated in Fig. 15. Consequently, their execution subsystems typically consist of either two general purpose pipelines, like in Intel's Pentium or Cyrix's M1 processors or two to four dedicated pipelined EUs, such as, e.g., in DEC's (later Compaq's, now HP's) Alpha 21 064.

In order to substantially increase the throughput of the microarchitecture, designers had to remove the issue bottleneck and—in order to capitalize on more available parallel executable instructions—also to appropriately raise the throughput of all relevant subsystems of the microarchitecture. This development led to the emergence of second-generation ("wide") superscalars, to be discussed along with the innovative techniques needed to implement them in the next section.

D. Second-Generation Superscalars

1) Overview: Conceptually, we consider superscalars as *second-generation models* if they *remove the issue bottleneck* caused by the direct issue scheme of their predecessors by introducing *dynamic instruction scheduling* and *register renaming*, as described in the next section. In addition, second-generation superscalars introduced a number of innovative techniques, such as predecoding, speculative branch processing, out-of-order execution of loads, store forwarding, or using reorder buffers and a significantly enhanced cache subsystem as well, all in order to appropriately increase the throughput of all subsystems of the microarchitecture, as shown in Fig. 18 and discussed in Section V-D4.

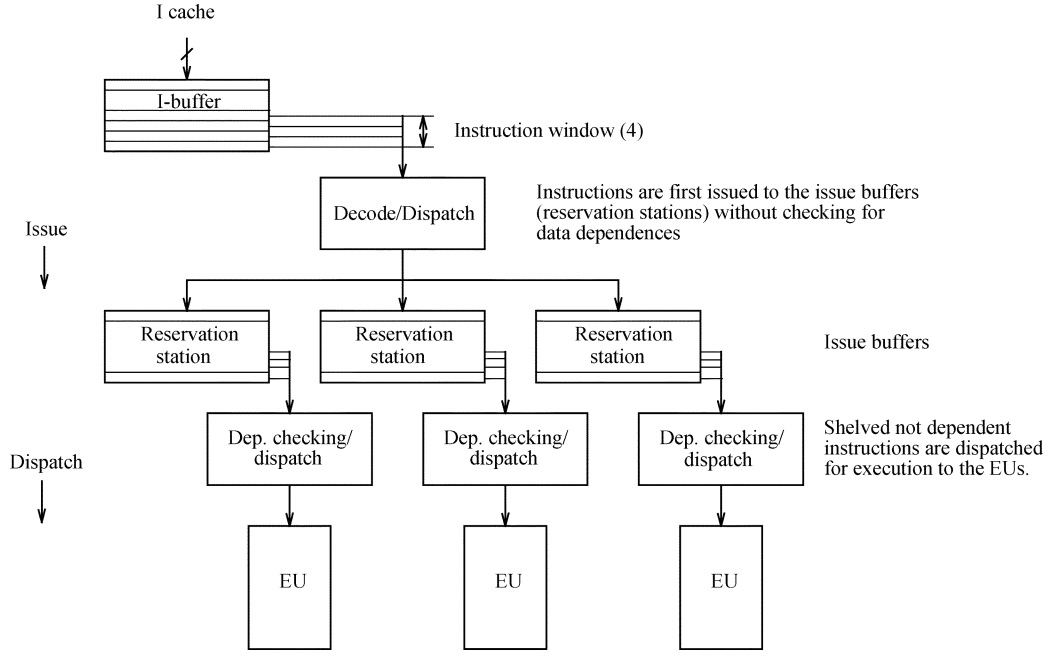


Fig. 19. Principle of dynamic instruction scheduling, assuming that the processor has individual issue buffers (called reservation stations) in front of the EUs.

All these enhancements result then in "wide" second-generation superscalar cores capable of processing typically four RISC or three CISC IPC. Examples of second-generation superscalars are the Alpha 21264, PA 8000, PowerPC 604, Power3, and Pentium Pro or K6.

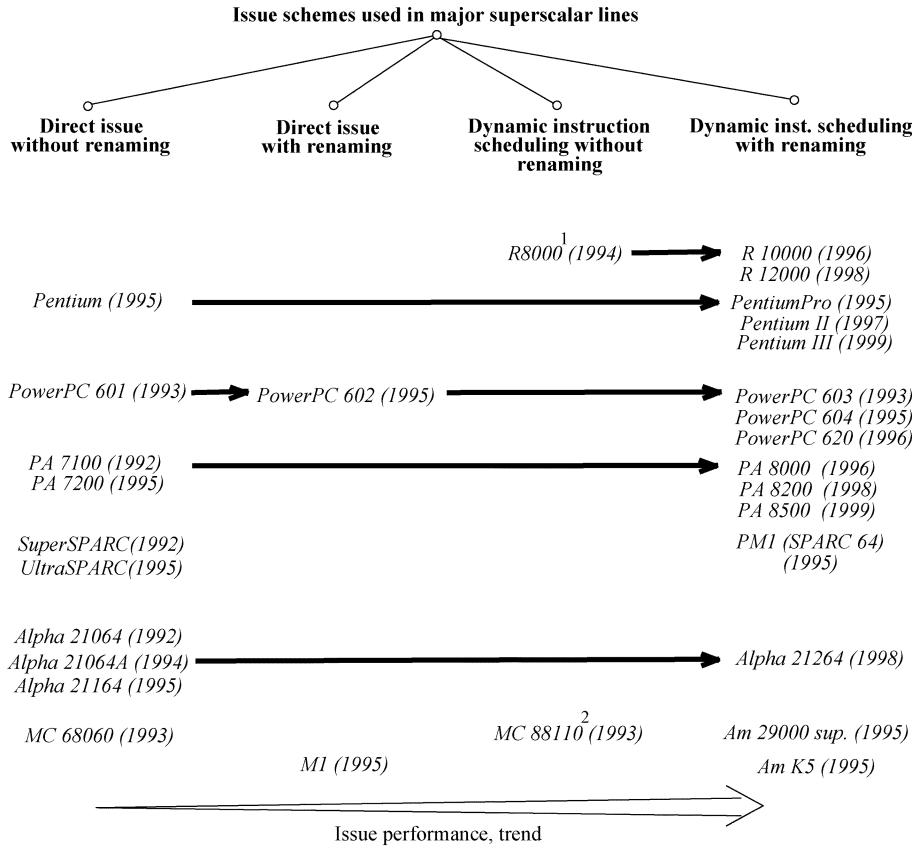
As far as the categorization of superscalars is concerned, we note that a few processors implement dynamic instruction scheduling or register renaming partially; that is, they restrict these techniques only to particular data types. For instance, the Power1 and Power2 implement both dynamic instruction scheduling and renaming partially. Such processors may be designated as *restricted second-generation processors*.

2) Techniques Introduced to Remove the Issue Bottleneck: a) Dynamic instruction scheduling: Known also as shelving or buffered instruction issue, dynamic instruction scheduling is the key technique used to remove the issue bottleneck of first-generation superscalars [4], [5], [58]. Simply speaking, it means buffered instruction issue, enhanced usually by out-of-order instruction scheduling. Its implementation presumes the availability of *issue buffers* (also designated as "reservation stations" in specific implementations) in front of the EUs, as shown in Fig. 19.[6] With dynamic instruction scheduling, the processor first issues the instructions into the issue buffers without checking for dependencies, such as data or control dependencies or busy EUs. Issued instructions remain in the issue buffers until they become free of dependencies and can be *dispatched* for execution. Now, as data or control dependencies or busy EUs no longer restrict the flow of instructions, the processor is able to issue in each cycle as many instructions into the issue buffers as its issue rate (usually four), provided that no trivial hardware restrictions, such as lack of available issue buffers or data path width limitations, exist. Since, in a well-designed microarchitecture, the hardware restrictions mentioned will not severely impede the throughput of the issue subsystem, dynamic instruction scheduling effectively eliminates the issue bottleneck of first-generation superscalars.

Dynamic instruction scheduling improves the throughput of the front end of the microarchitecture not only by removing the issue bottleneck of the direct issue scheme but also by significantly *widening the instruction window* and allowing instructions to be issued usually out-of-order for execution. (As a reminder, in the direct issue scheme, the instruction window is usually only two to three instructions wide.) In contrast, while using dynamic instruction scheduling, the instruction window consists of all issue buffer entries that are scanned for executable instructions. As second-generation superscalars usually check dozens of issue buffer entries for independent instructions, dynamic instruction scheduling greatly widens the instruction window compared to the direct issue scheme. Since a wider out-of-order window enables the processor to find considerably more parallel executable instructions per clock cycle on average than a smaller in-order one, dynamic instruction scheduling greatly increases the throughput potential of the microarchitecture.

b) Register renaming: This technique is used to augment the benefits of dynamic instruction scheduling by elimi-

[6]Here we note that beyond individual reservation stations, each one serving a single EU as shown in Fig. 19, there are further possibilities to implement dynamic instruction scheduling [49], [58], like using group reservation stations serving a group of EUs, as used in a number of processors, such as the Alpha 21 264 or the Pentium4, or having a centralized reservation station serving all available EUs, as implemented in Intel's Pentium Pro, Pentium II, and Pentium III processors.



[1] The R8000 shelves only FP instructions.
[2] The MC88110, shelves only load/store instructions.

Fig. 20. Introduction of dynamic instruction scheduling and register renaming in superscalars.

nating false data dependencies, i.e., write after read (WAR)[7] and write after write (WAW)[8] dependencies between register operands of subsequent instructions before issuing them into the issue buffers. In this way, register renaming raises the number of parallel executable instructions in the instruction window and, thus, the performance of the issue subsystem.

In order to *implement register renaming*, the processor allocates a rename buffer to each destination register of an instruction before issuing instructions. This buffer will be used to temporarily hold the result of the instruction instead of the architectural register. The rename logic also tracks current register allocations, renames source registers in order to ensure that source operands are fetched from the appropriate locations (either from the rename register or the specified architectural register), writes the results from the rename buffers into the addressed architectural registers, and, finally, reclaims rename buffers that are no longer in use. Renaming must also support a recovery mechanism for erroneously speculated branches and occurring interrupts [4], [5], [49].

Fig. 20 follows the introduction of dynamic instruction scheduling and renaming in major superscalar lines. As indicated, first-generation superscalars typically made use of the direct issue scheme. A few subsequent processors introduced either renaming alone (like the PowerPC 602 or the M1) or dynamic instruction scheduling alone (such as the MC88110 or the R8000). In general, however, dynamic instruction scheduling and renaming emerged together, marking the debut of the second-generation superscalars in the mid-1990s [58]. Soon, both techniques became standard constituents of mainline superscalars except for the UltraSPARC line. Designers of this line ruled out both techniques at the beginning of the design process in order to reduce time to market [59].

3) Execution Model of Second-Generation Superscalars: First, we will address RISC processors and subsequently we will expand our discussion to CISC processors.

In Section V-D1, we summarized the characteristic features of second-generation superscalars. As emphasized, they typically employ dynamic instruction scheduling and renaming in order to remove the issue bottleneck of their predecessors. Accordingly, in the simplified *execution model*

[7] A WAR dependency exists between two instructions i_k and i_l if i_l specifies as a destination one of the source registers of i_k, as in the following example:

$$i_k : \text{mul}\, r1, r2, r3; \quad //r1 \leftarrow (r2) \times (r3)$$
$$i_l : \text{add}\, r2, r5, r6; \quad //r2 \leftarrow (r5) + (r6).$$

[8] Two instructions i_k and i_l are said to be WAW dependent if they both write the same destination, as in the following example:

$$i_k : \text{mul}\, r1, r2, r3; \quad //r1 \leftarrow (r2) \times (r3)$$
$$i_l : \text{add}\, r1, r5, r6; \quad //r1 \leftarrow (r5) + (r6).$$

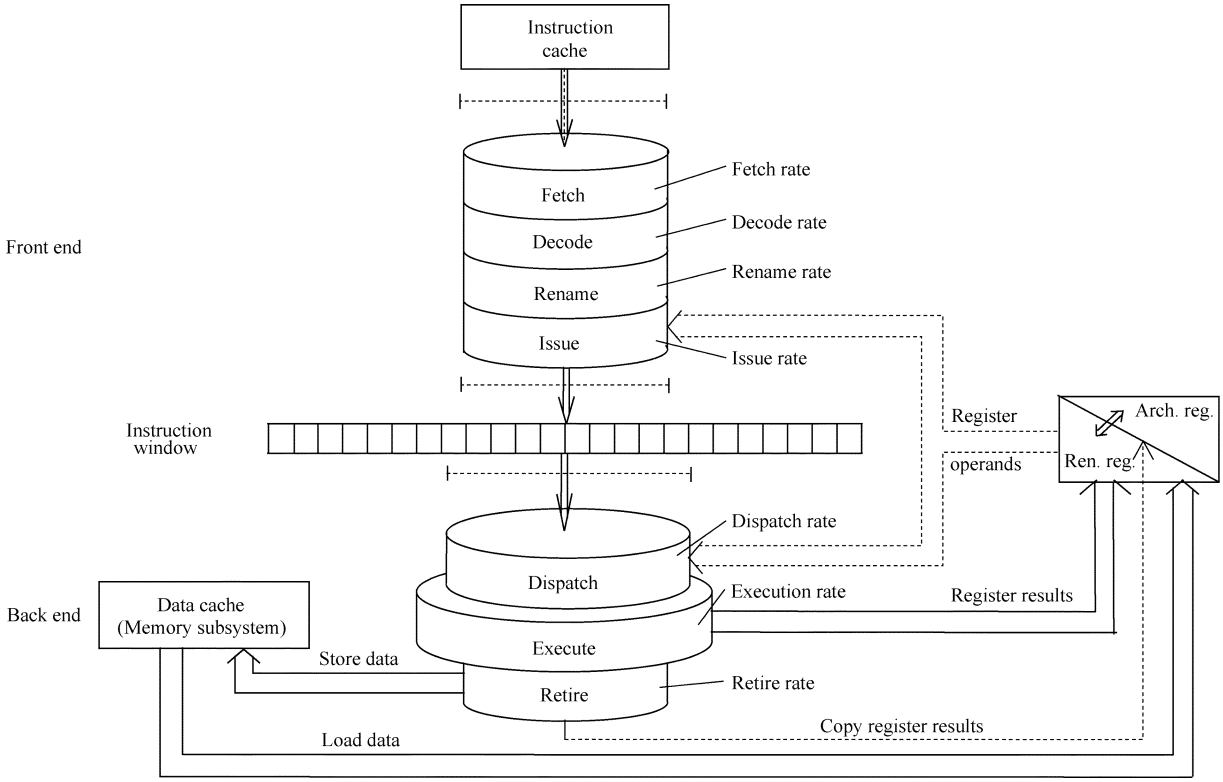


Fig. 21. Simplified execution model of a second-generation superscalar RISC processor.

of second-generation superscalar RISCs, the front end of the microarchitecture consists of the fetch, decode, rename, and issue subsystems, as indicated in Fig. 21. Since second-generation superscalar RISCs are usually four instructions wide by design, their fetch, decode, rename, issue, and retire rates typically all equal four instructions/cycle.

The front end feeds instructions into the issue buffers, whose active parts (buffer entries checked for executable instructions) constitute the instruction window. The instruction window may be considered as the interface between the front end and the back end sections of the microarchitecture.

Executable instructions are dispatched from the instruction window to available EUs by the dispatch subsystem. Referenced register operands are supplied either during instruction issue or during instruction dispatch. The execute subsystem performs the operations required. Results are forwarded to the rename registers rather than to the architectural register for temporal holding. Finally, executed instructions are retired in program order directed by the retire subsystem. Then register results are copied at this stage in program order from rename registers to the corresponding architectural registers or are forwarded to the data cache (in case of memory data).

We note that dispatch rates are typically higher than issue rates, as indicated in Fig. 21. Whereas second-generation superscalar RISCs usually issue 4 instructions per cycle, they dispatch in most cases 5–8 instructions per cycle (see Table 1). The reason for this is as follows. EUs often cannot accept new instructions for execution in consecutive cycles, as their repetition rates for complex instructions are usually higher than one. For particular instructions, like division, the repetition rate is even much higher than one. Thus, in order to sustain the desired execution flow—equaling the issue rate—in spite of temporarily busy EUs, the processor obviously needs to be able to supply more instructions/cycle for execution than the issue rate. Execution rates are usually even higher than dispatch rates in order to support the desired execution flow for a wide variety of possible mixes of the dispatched instructions. As different sets of EUs can be called up cycle by cycle to execute the actual mix of dispatched instructions, sustaining a high enough execution flow requires ample parallel operating execution resources (EUs).

Table 1
Comparing the Issue and Dispatch Rates of Second-Generation Superscalar RISC Processors

Processors/year of volume shipment	Issue rate (instr./cycle)	Dispath rate [1] (instr./cycle)
PowerPC 603 (1993)	3	3
PowerPC 604 (1995)	4	6
Power2 (1993)	4/6 [2]	10
PM1 (SPARC 64) (1995)	4	8
PA8000 (1996)	4	4
R10000 (1996)	4	5
Alpha 21264 (1998)	4	6

[1] Since address calculations are performed separately, the given numbers are usually to be interpreted as operations/cycle. For instance, the Power2 performs maximum 10 operations/cycle, which corresponds to 8 instr./cycle.
[2] The issue rate is 4 for sequential mode and 6 for target mode.

We note that the *microarchitectures of second-generation CISC processors* differ somewhat from those shown for RISC processors, as they usually first convert CISC instructions into simple *RISC-like internal operations* and then execute these by a RISC core, similar to that shown in Fig. 17. We note that the internal operations are denoted differently in different CISC lines, e.g., "μops" in Intel's Pentium Pro and subsequent models, "RISC86 operations" in AMD's K5-K7, or "ROP's" in Cyrix's M3. In CISC processors, the retire subsystem also performs a "reconversion" by completing those internal operations that belong to the same CISC instruction all at once. Thus, the execution model of RISC processors is basically valid for CISC processors as well.

As far as *second-generation CISC processors* are concerned, they typically decode up to three CISC instructions per clock cycle and usually perform an internal conversion to RISC-like operations, as discussed above. Since x86 CISC instructions generate on average approximately 1.2–1.5 RISC-like instructions [60], the front end of advanced CISC processors has roughly the same width as that of advanced RISC processors in terms of RISC-like operations.

Finally, we point out an important feature concerning the internal operation of second-generation superscalars. Assuming the use of dynamic instruction scheduling, register renaming, branch prediction, and speculative branch processing, basically only producer–consumer-type register data dependencies (RAW dependencies) between register operands as well as memory data dependencies restrict the processor from executing instructions in parallel from the instruction window (here we neglect any obvious hardware limitations, such as the lack of buses or EUs needed, or possible dispatch constraints, such as in-order dispatch schemes). With the assumptions mentioned, second-generation superscalars execute instructions with register operands internally according to the *dataflow principle of operation*. Consequently, in the assumed mode of operation, the producer–consumer-type register data dependencies (RAW dependencies) set the *dataflow limit of execution*.

4) Techniques Introduced to Increase the Throughput of Particular Subsystems of Second-Generation Superscalars: a) Overview: Raising the throughput of the microarchitecture is a real challenge, as it requires a properly orchestrated enhancement of all subsystems involved. In addition to enhancing the issue subsystem by dynamic instruction scheduling and register renaming, second-generation superscalars introduced a number of techniques in order to increase the throughput of all subsystems of the microarchitecture. Below we give an overview of these techniques.

b) Increasing the throughput of the instruction fetch subsystem: Ideally, the instruction fetch subsystem supplies instructions for processing cycle by cycle at the issue rate. However, branches, both unconditional and conditional as well as instruction cache misses may disrupt the continuous stream of supplied instructions. This impedes performance the more the wider the core and the longer the pipeline. In order to "smooth" the instruction flow despite branches,

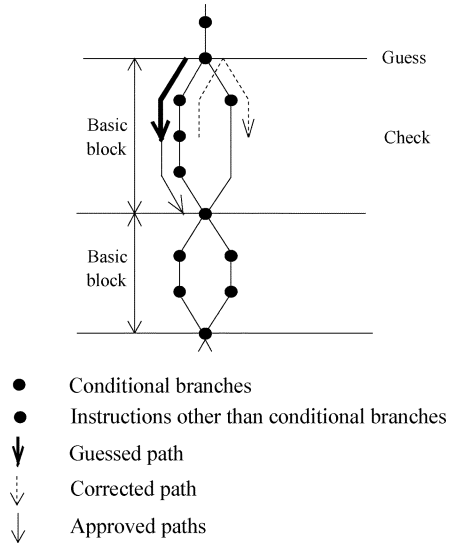


Fig. 22. Principle of speculative execution of conditional branches assuming speculation along a single conditional branch.

designers introduced three sorts of advanced techniques: 1) more intricate and accurate dynamic branch prediction schemes that are based on branch history, like 2-b dynamic prediction or two-level prediction [49], [53]–[55]; 2) diverse techniques to access branch target paths as quickly as possible using branch history tables, branch target buffers, subroutine return stacks, etc., techniques that already appeared in a few first-generation superscalars [49] but became widespread only in second-generation models; and 3) speculative branch processing to avoid blocking of the fetch subsystem due to unresolved conditional branches.

We remind the reader that in first-generation superscalars, unresolved conditional branches typically block instruction issue until the referenced condition becomes known. This can cause a number of idle fetch cycles, first of all for conditions set by results of long latency operations, such as divisions. *Speculative branch processing* aims to remedy this situation. Its basic idea is to speculatively fetch, issue, and execute instructions after an unresolved conditional branch along the predicted path; later, when the condition becomes known, check the correctness of the prediction made; for correct predictions, acknowledge the speculatively executed instructions and go on with the execution; otherwise, discard all speculatively executed instructions and resume execution along the right path, as indicated in Fig. 22.

A key point of speculative branch processing is *the depth* of the implemented branch speculation. On the one hand, wide instruction windows, keeping tens of instructions, call for deep speculation, i.e., for speculation along multiple consecutive conditional branches, in order to fill the instruction window as much as possible even in this case. On the other hand, the deeper branch speculation (i.e., the more consecutive branches a guessed path may involve), the higher the penalty for wrong guesses in terms of wasted cycles needed to discard wrongly executed instructions. As a tradeoff, second-generation superscalars allow branch speculation usually along a different number of unresolved conditional branches, e.g., along two conditional branches

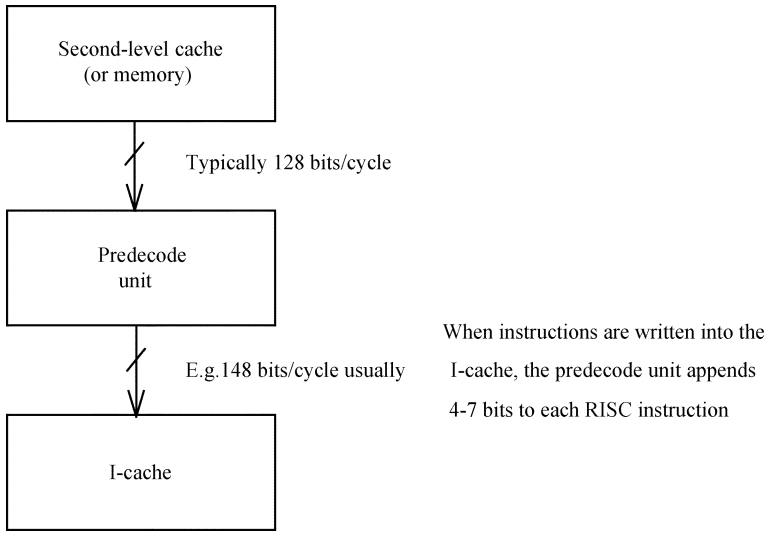


Fig. 23. Basic idea behind predecoding.

like in the Power2, or four conditional branches like in the PowerPC 620 or the R10 000.

As a rule of thumb, *wider instruction windows call for deeper speculation and more accurate branch prediction* [55].

We note that speculative branch processing became feasible through the introduction of two novel techniques (that is, renaming and the reorder buffer), both of which emerged in second-generation superscalars. Renaming (see Section V-D2b) provides temporary storage space for speculative results, while the reorder buffer (to be described in Section V-D4f) tracks the speculative execution and, thus, supports a replay if needed.

Finally, we add that instruction cache misses are addressed by various *instruction prefetch schemes* to reduce incurred latencies [32].

c) Increasing the throughput of the decode subsystem: In second-generation superscalars, decoding becomes yet more complex than in their predecessors because of two reasons.

1) The introduction of dynamic instruction scheduling and renaming gives rise to a time critical path consisting of decoding, renaming and issuing instructions to the issue buffers, since along this path a variety of checks are needed, such as for the availability of empty rename and issue buffers, of the required bus widths to forward multiple instructions into the same buffer, etc.
2) Increased issue rates (of typically four in RISCs and three in CISCs) of second-generation superscalars call for even more complex checks and lengthen this time critical path even further.

In fact, pipelined instruction processing segments this path into decode, rename and issue subtasks, where one or more clock cycles (pipeline stages) can be scheduled for each subtask. If, however, the time to perform one of the subtasks becomes longer than foreseen, either the clock frequency must be lowered or additional pipeline stages need to be inserted, which unfortunately increases the penalty for mispredicted branches. An appropriate technique to avoid the unduly lengthening of the decode cycle is known as predecoding [49], a technique that emerged along with second-generation superscalars and rapidly became their standard feature.

The fundamental idea behind *predecoding* is to reduce the complexity of the decode stage by partially decoding instructions while fetching them into the instruction cache, as indicated in Fig. 23. The results of predecoding may include identifying instruction types, recognizing branches, determining instruction length (in the case of a CISC processor), etc.

We note that *trace caches* holding already decoded instructions, used in some third-generation superscalars such as the Pentium4, mentioned in Section VI-C, represent another solution to alleviate the difficulties caused by the time-critical decode–rename–issue path [61], [62].

d) Increasing the throughput of the dispatch subsystem: In order to increase the throughput of the dispatch subsystem, the dispatch rate needs to be raised and the instruction window widened as well, as detailed below.

1) *Raising the dispatch rate* means increasing the maximum number of instructions that can be dispatched per cycle to the available EUs. This is the "brute force" approach to increasing the throughput of the dispatch subsystem. It assumes more available execution resources, such as EUs, data paths, and more complex logic to select more executable instructions from the window, as already mentioned. Dispatch rates of various second-generation superscalars are given in Table 1.
2) *Widening the instruction window* is a more subtle approach to increasing the throughput of the dispatch subsystem. In fact, a higher dispatch rate can only be implemented if the dispatch subsystem can find enough executable instructions in the instruction window cycle by cycle. Since more parallel executable instructions are expected to be found per cycle in a wider window than in a smaller one, higher dispatch rates necessitate wider instruction windows. This is

Table 2
Width of the Instruction Window in Second-Generation Superscalar Processors

	Processor	Width of the instr. window
RISC processor	PowerPC 603 (1993)	3
	PM1 (Sparc64) (1995)	36
	PowerPC 604 (1995)	12
	PA8000(1996)	56
	PowerPC 620 (1996)	15
	R10000 (1996)	48
	Alpha 21264 (1998)	35
	Power3 (1998)	20
	R12000 (1998)	48
	PA8500 (1999)	56
CISC processor	Nx586 (1994)	42
	K5 (1995)	12
	PentiumPro (1995)	20
	K6 (1996)	24
	Pentium II (1997)	20
	K7 (1998)	54
	M3 (2000)	56

the reason why second-generation (as well as more recent) superscalars have considerably wider instruction windows than earlier ones, achieved by means of much more issue buffers and an enhanced, usually out-of-order dispatch logic, as shown in Table 2. However, a wider window requires deeper and more accurate branch speculation, as emphasized earlier.

Finally, we note that *parallel optimizing compilers* also contribute to increasing the average number of parallel executable instructions available in the window per cycle. Nevertheless, as our paper focuses on the microarchitecture itself, readers interested in this topic are referred to the literature [63], [64].

e) Increasing the throughput of the execution subsystem: There are three main options to increase the throughput of the execution subsystem: 1) increasing the execution rate of the processor by providing more parallel operating EUs; 2) reducing the repetition rates of EUs (i.e., the number of clock cycles needed until an EU can accept a new instruction for execution); and 3) shortening the execution latencies of EUs (i.e., the number of clock cycles needed until the result of an instruction becomes available to a subsequent instruction). Below we will discuss only the last issue mentioned.

Clearly, the earlier that existing RAW dependencies are resolved in the instruction window, the more instructions will be available for parallel execution on average per cycle. This calls for *shortening the execution latencies of instructions*. As the memory subsystem operates much more slowly than the instruction pipelines of the core (relatively the slower the higher the clock frequency), and since load instructions represent a large fraction—that is, approximately 25%–35%—of all instructions [65], a crucial point for increasing the throughput of the execution subsystem is *reducing the execution latencies of loads*. Both the cache subsystem, to be discussed in Section V-D4g, and the execution subsystem of the core (the load/store units or the load/store pipelines) contribute to this. Here we focus on the execution subsystem and subsequently outline two related techniques that emerged mainly in second-generation superscalars, designated as: 1) out-of-order execution of loads and 2) store forwarding.

1) *Out-of-order execution of loads* is a technique that allows younger, already executable loads to bypass older loads and stores that are not yet ready to execute. This technique effectively contributes to reducing the impediments of cache misses. However, in order to preserve program logic, loads to be reordered must not bypass older stores writing to the same address. To avoid this, the referenced address of the load to be reordered needs to be compared with all store addresses, typically held in a pending store queue. Here we will not go into details about possible alternatives of implementing the address checks required, but refer to the related literature [66], [67]. Superscalars making use of out-of-order execution of loads are the PowerPC 604, PowerPC 620, R12000, SPARC64, or the recent Pentium4.
2) *Store forwarding* is another method for reducing load latencies. It means forwarding store data to dependent loads before the stores are written into the data cache. Superscalars employing this technique include the Alpha 21264, Cyrix's M3, the K6-3, UltraSPARC3, or recent third-generation superscalars, such as the Pentium4 or the Opteron.

f) Increasing the throughput of the retire subsystem: A further characteristic technique introduced in second-generation superscalars is the *reorder buffer (ROB)*. It tracks the state of instructions in flight (issued and not yet retired), and guarantees their in-order retirement. First described in 1988 [68], the ROB is basically a circular buffer with head and tail pointers, as Fig. 24 denotes. The head pointer indicates the next free entry awaiting the subsequently issued instruction, whereas the tail pointer marks the instruction which will retire next and leave the ROB. As instructions are issued for processing, a new entry is allocated to each one in the ROB directed by the head pointer. While instructions are processed, their states, such as issued, in execution, or finished, are tracked in the corresponding ROB entry. The ROB ensures the in-order retirement of the instructions by allowing instructions to retire if they are finished and all previous instructions are already retired. Retired instructions leave the ROB and write their results into the architectural registers or the memory. As an instruction leaves the ROB, the tail pointer is stepped to mark the instruction that will retire next.

We note that the ROB can effectively support both speculative execution and interrupt processing as well.

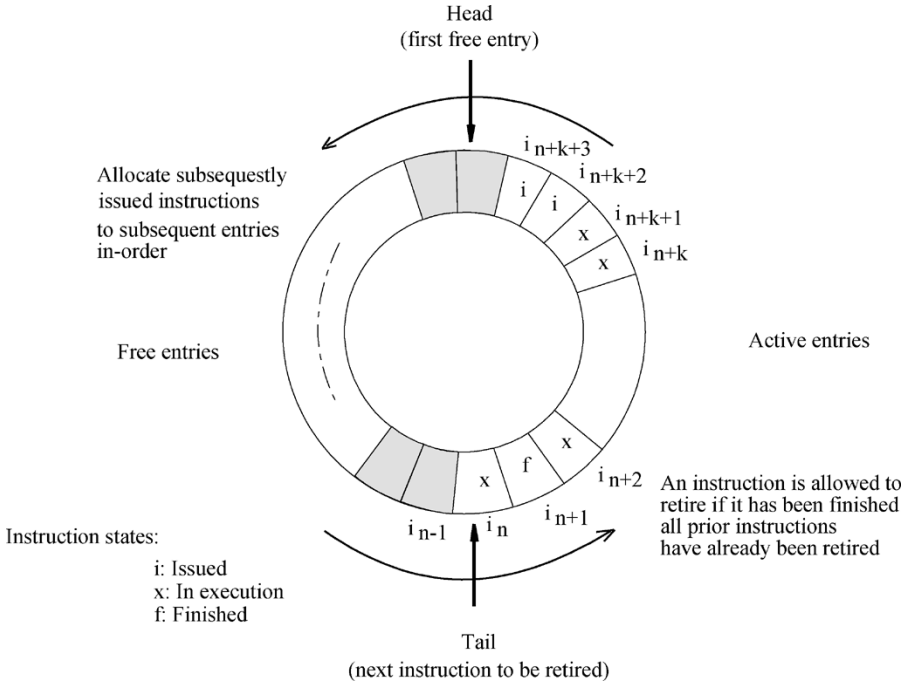


Fig. 24. Principle of the ROB.

Due to their effective method of operation, ROBs proliferated rapidly in the mid-1990s and became the standard solution of ensuring the sequential consistency of instruction execution along with the second-generation superscalars [49].

g) Increasing the throughput of the cache subsystem: As pointed out in Section V-B1, wider or higher clocked cores need more advanced cache subsystems. Since second-generation cores are both wider and clocked higher than their predecessors, they typically need more advanced cache subsystems than their first-generation peers.

Cache subsystems evolved in two aspects. First, second-generation superscalars typically include *nonblocking data caches* allowing more than one pending cache miss, unlike first-generation processors that incorporate either blocking data caches or straightforward nonblocking data caches allowing only a single pending cache miss. Second, *their L2 caches* are usually *direct coupled* to the processor chip via a dedicated bus (called the back-side bus) rather than via the relatively slow processor bus (front-side bus) as typical in first-generation superscalars (see Fig. 25).

Finally, we emphasize that the overall design of a microarchitecture calls for discovering and removing possible bottlenecks in all subsystems of the microarchitecture. This task, however, usually requires tedious, iterative cycle-by-cycle simulation on a number of benchmark applications.

5) Increasing the Word Length of Superscalar Processors: The word length of a processor refers both to the related ISA and the core of the processor, by defining the length of basic data types (fixed point and logical data), addresses, and the respective registers and data paths. As Fig. 26 indicates, first-generation superscalars had word lengths (see Section V-C3.) of both 32 and 64 b. While most first-generation RISCs were 32-b designs, including the models PA7100, PA7200, POWER, PowerPC 601, and SuperSPARC, two RISC lines already offered 64-b ISAs and cores, such as DEC's (now HP's) Alpha line and MIPS with its MIPS IV ISA and R8000 core. In contrast, first-generation CISCs, such as the Pentium, made use of a 32-b word length. As far as second-generation RISCs are concerned, they were typically 64-b designs, whereas second-generation CISCs, such as Intel's PentiumPro and AMD's K6 processors, stuck to a word length of 32 b. Here we note that Intel still stayed with the 32-b word length in its third-generation superscalars, such as the Pentium III, and Pentium4 processors as well, whereas AMD took the initiative and defined a 64-b x86 ISA (designated as the x86-64 ISA) and implemented it in its high-end (Opteron) and desktop (Athlon 64) lines. Concerning this point, we note that Intel's conservative position is presumably determined by the intention to avoid direct competition with its own 64-b VLIW-based Itanium line.

6) Limits of Utilizing Issue Parallelism Available in General Purpose Applications: Obviously, it is rather impractical to widen the microarchitecture beyond the extent of available ILP. As general purpose programs have no more than about 4–6 parallel executable RISC instructions per cycle on average [69], and the cores of second-generation microarchitectures are already at least four-wide designs, not much room seems to remain for performance increase through widening the microarchitecture even further, at least not for general purpose applications. Nevertheless, the performance of superscalar processors can be increased further for dedicated applications by introducing intrainstruction parallelism through SIMD instructions, as discussed in the next section.

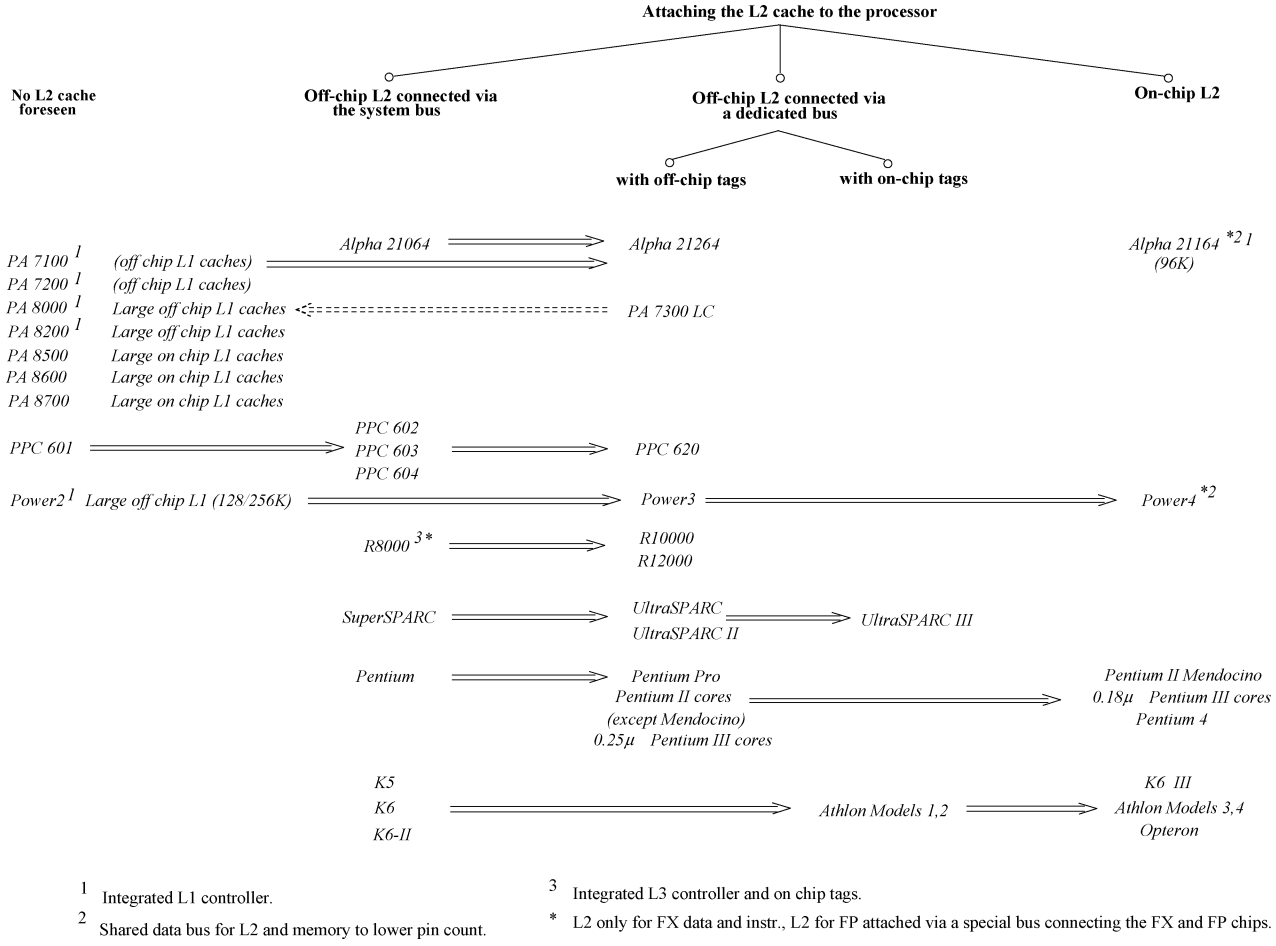


Fig. 25. Attachment of L2 caches in superscalars.

VI. Introduction and Evolution of Utilizing Intrainstruction Parallelism

A. Main Approaches to Introduce Intrainstruction Parallelism

The last major option for increasing processor performance at the instruction level is to include *multiple data operations* into the instructions—that is, introducing *intrainstruction* parallelism. There are three different approaches to achieve this: 1) by means of dual-operation instructions; 2) by SIMD instructions; and 3) by VLIW instructions, as indicated in Fig. 27. However, unlike the introduction of temporal and issue parallelism, intrainstruction parallelism requires the introduction of either an appropriate *extension of the ISA*, as in approaches 1) and 2), or a *completely new ISA*, as in alternative 3). In the next sections, we outline these options.

B. Dual-Operation Instructions

Dual-operation instructions, as the name suggests, include two different data operations within the same instruction. The most widely used one is the *multiply–add instruction* ("multiply-and-accumulate" or "fused multiply–add" instruction) that calculates the dot product $(x = a * b + c)$ for floating-point data.

Multiply–add instructions were introduced in the early 1990s in the POWER [70], PowerPC [71], PA-RISC [72], and MIPS-IV [73] ISAs and in the respective processor implementations. Other examples of dual-operation instructions are fused load/op, shift & add, etc., instructions. Although useful mainly for numeric computations, dual-operation instructions found their way into most RISC ISAs and the related processor models. In general purpose applications, however, they increase only marginally the average number of operations executed per instruction (OPI).

C. SIMD Instructions

SIMD instructions allow the same operation to be performed on more than one set of operands. For example, in Intel's MMX multimedia extension [74] the PADDW MM1, MM2 SIMD instruction performs four fixed point additions on the four 16-b operand pairs held in the 64-b registers MM1 and MM2.

As Fig. 27 indicates, SIMD instructions may refer to either FX (fixed point) or FP (floating point) data. *FX-SIMD instructions* enhance multimedia applications by allowing mul-

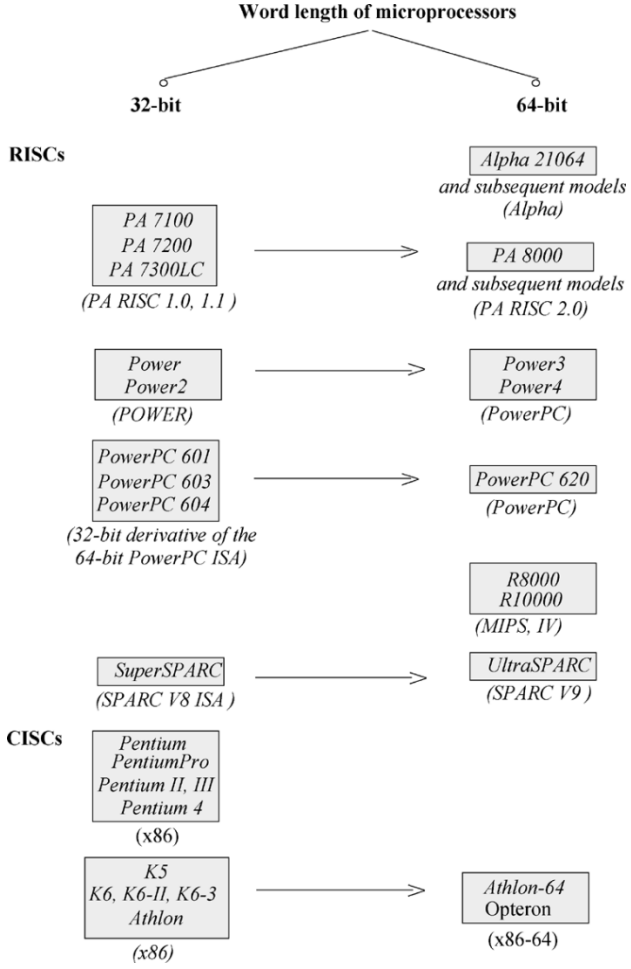


Fig. 26. Word length of superscalar processors.

tiple (2/4/8/16/32) operations on pixels, whereas *FP-SIMD instructions* accelerate 3-D graphics by executing (usually) two to four FP operations simultaneously.

FX-SIMD instructions were pioneered in 1993–1994 in the MC88110 and PA-7100LC processors, as shown in Fig. 28. Driven by the proliferation of multimedia (MM) applications, FX-SIMD extensions (such as AltiVec from Motorola [75], MVI from Compaq [76], MDMX from MIPS [77], MAX-2 from Hewlett-Packard [78], VIS from Sun [79], and MMX from Intel [74]) soon became a standard feature of most established processor families.

Aimed at supporting 3-D graphics, FP-SIMD extensions emerged in the very end of the 1990s with the 3DNow! x86 enhancement from AMD, Cyrix, and IDT [80], followed by the SSE (Streaming SIMD Extension) from Intel [81]. These extensions were implemented in AMD's K6-2, K6-3, and Intel's Pentium III processors, respectively. Other manufacturers shortly followed suit with their FP-SIMD enhancements, such as Motorola in its G4 or IBM in the Power4, as indicated in Fig. 28.

Concerning both FX- and FP-SIMD extensions, Intel made substantial progress with its SSE2 extension [82], which introduced 144 new instructions and doubled the word length of SIMD data and associated registers from 64 to 128 b, implemented in the company's Pentium4 processor. Subsequently, forced by compatibility requirements, AMD also implemented both the SSE and SSE2 extensions in their Athlon and Opteron lines. We note that today SSE and SSE2 became a *de facto* standard for SIMD extensions of x86 processors.

In fact, the introduction of SIMD instructions marks the advent of *third-generation superscalars* (see Fig. 29). We emphasize that unlike previous generations, third-generation superscalars require an extension of their ISAs, as already mentioned. Beyond the enhancements needed to execute SIMD instructions, their cores are typically augmented with on-die L2 caches, to cope with increased memory bandwidth requirements of multimedia and 3-D applications. Further on, these applications also gave rise to enhancements of the system architecture, such as the introduction of the AGP bus.

We point out that MM and 3-D support boosts processor performance mostly in dedicated applications. For instance, Intel reported impressive per-cycle performance gains of its multimedia and 3-D-enhanced processors over their predecessors based on selected MM and 3-D benchmarks, as indicated in Table 3.

In contrast, MM and 3-D support only results in a rather modest, if any, per-cycle performance gain for general purpose applications, as Table 4 indicates for subsequent Intel processors.

D. VLIW Approach

The third major possibility to introduce intrainstruction parallelism is the *VLIW approach*. VLIWs provide a number of simultaneously operating EUs, much like superscalars, but these EUs are controlled by different fields of the same instruction word rather than by subsequent instructions, like in superscalars. As a consequence, VLIW processors with a large number of EUs need VLIWs—hence, the name. For instance, Multiflow's TRACE VLIW machine, an early commercial VLIW implementation, used 256–1024-b long instruction words to specify 7–28 simultaneous operations within the same instruction word [136].

Unlike superscalars, VLIWs are *scheduled statically*. This means that the compiler assumes full responsibility for resolving every single type of dependency. To be able to do so, the compiler needs intimate knowledge of the microarchitecture concerning the number, types, repetition rates and latencies of the EUs, load-use latencies of the caches, etc. On the one hand, this results in rather complex and *technology-dependent compilers*, while on the other hand it leads to *reduced hardware complexity* versus comparable superscalar designs. In addition, the compiler is expected to perform aggressive parallel optimizations in order to find enough executable operations per cycle for high throughput.

VLIW proposals emerged as paper designs in the first half of the 1980s (Polycyclic architecture [137], ELI-512 [138]), followed by two commercial machines in the second half of the 1980s (Multiflow's TRACE [136] and Cydrome's Cydra-5 [139]). We will term these traditional designs as *early VLIWs*.

Early VLIWs disappeared from the market fairly quickly, partly due to their deficiencies—technological

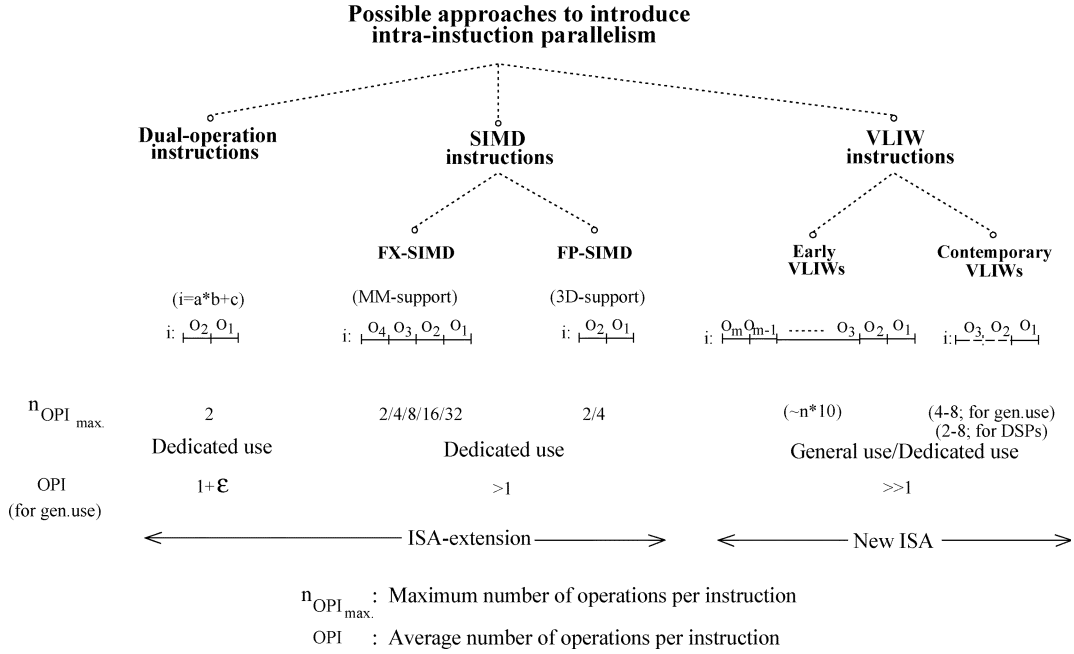


Fig. 27. Possibilities to introduce intrainstruction parallelism.

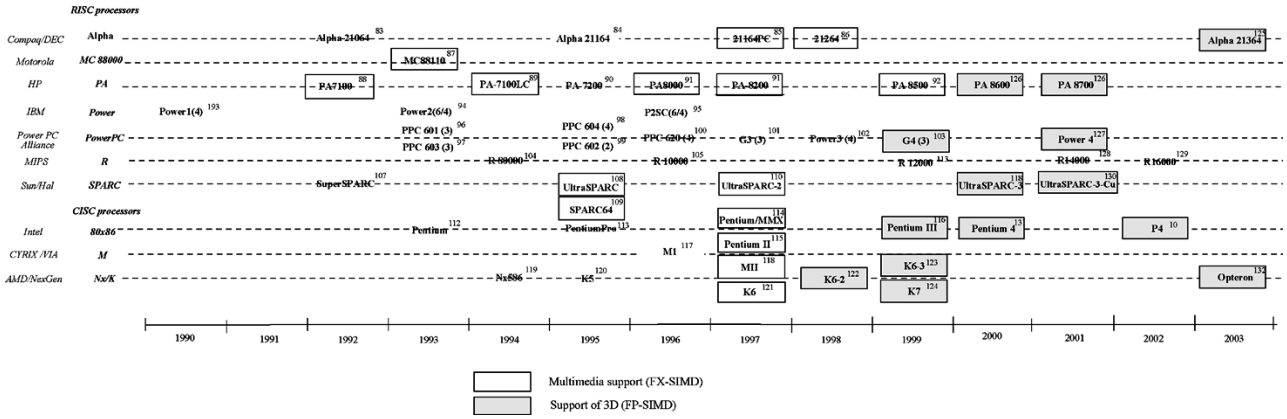


Fig. 28. Emergence of FX-SIMD and FP-SIMD instructions in microprocessors. The references to superscalar processors are given as superscripts behind processor designations.

sensitivity of their compilers, wasted memory fetch bandwidth owing to sparsely populated instruction words, etc. [4]—as well as to the onus of their manufacturers being startup companies.

The reduced hardware complexity of VLIW designs versus superscalar designs and the progress achieved in compiler technology have led to a revival of VLIWs in the late 1990s, both for digital signal processors (DSPs) and general purpose applications. *VLIW-based DSPs*, such as Philips' TM1000 TriMedia processors [140], TI's TMS320C6000 cores [141], the SC140 core from Motorola and Lucent [142], ADI's TigerSharc [143], or Sun's MAJC core [144] are intended in the first line for embedded multimedia applications. These designs exploit the inherent high parallelism of such applications that can be easily exposed by a compiler and effectively executed on two- to eight-wide WLIW cores.

General purpose VLIWs with four to eight operations per instruction have recently emerged on the horizon, including Intel's six-wide 64-b Itanium (aka Merced) line [145], [146] that implements the Explicitly Parallel Instruction Computing (EPIC) VLIW style [147] and Transmeta's four-wide Crusoe [148] and eight-wide Efficeon lines [149] (see Fig. 30). While Intel's six-wide Itanium2 is intended to compete with typically four-wide superscalars on the server market, Transmeta's VLIW processors are aimed at the portable computing sector, taking advantage of the simpler and, consequently, less power-consuming hardware of their statically scheduled VLIW designs compared to dynamically scheduled superscalars.

In summary, out of the above approaches, DSP-oriented VLIWs found their way into dedicated embedded applications, whereas general purpose VLIWs became vital rivals of recent superscalars first of all in the server and portable markets.

VII. Main Road of the Microarchitecture Evolution

As pointed out before, the main road of the microarchitecture evolution is marked by an increasing utilization of

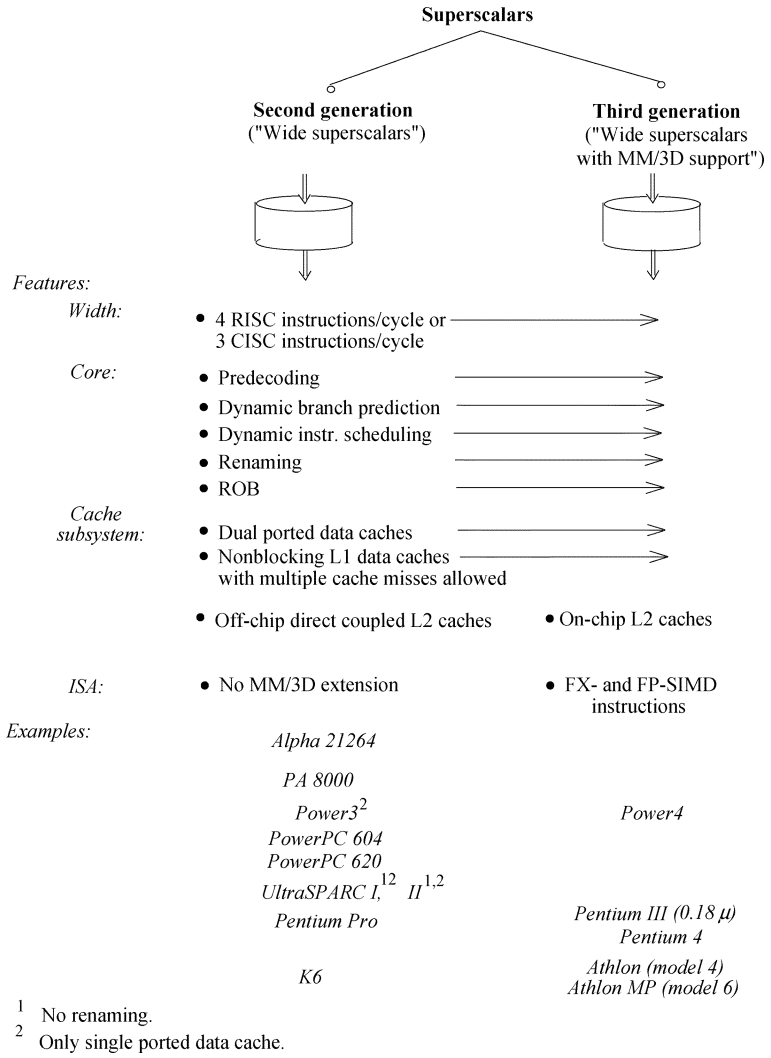


Fig. 29. Contrasting second- and third-generation superscalars.

Table 3
Per-Cycle Performance Gains of Intel's Subsequent Processor Models Measured on Selected MM and 3-D Benchmarks

Related processors	Benchmark	Per cycle performance gain	Reference
Pentium II over Pentium Pro	Media Benchmark	37 %	[133]
Pentium III over Pentium II	3D Lighting and Transformation Test of the 3D WinBench 99	61 %	[134]
Pentium 4 over Pentium III	3D Rendering Adobe After Effects 6 3D Mark03 Pro	40 % 19,4 % 9 %	[135]

available ILP. This progress took place while designers introduced temporal, issue, and intrainstruction parallelism one after another in subsequent models of their superscalar lines (see Fig. 31).

In particular, temporal parallelism was the first to make its debut with *pipelined processors*. The emergence of pipelined instruction processing stimulated the introduction of caches and of branch prediction, as Fig. 32 shows. Thus, the entire potential of temporal parallelism could be exhausted. For further performance increase, issue parallelism became utilized next via the introduction of superscalar instruction issue. Superscalars evolved in three generations. *First-generation superscalars* made use of the direct (unbuffered) instruction issue scheme, accompanied

Table 4
Per-Cycle Performance Gains of Intel's Subsequent Processor Models Measured on the CPU Intensive PCMark Benchmarks

Related processors	Benchmark	Per cycle performance gain	Reference
Pentium II over Pentium Pro	PCMark	3-5 %	[1]
Pentium III over Pentium II	PCMark	3-5 %	[1]
Pentium 4 over Pentium III	PCMark	~ 2 %	[135]

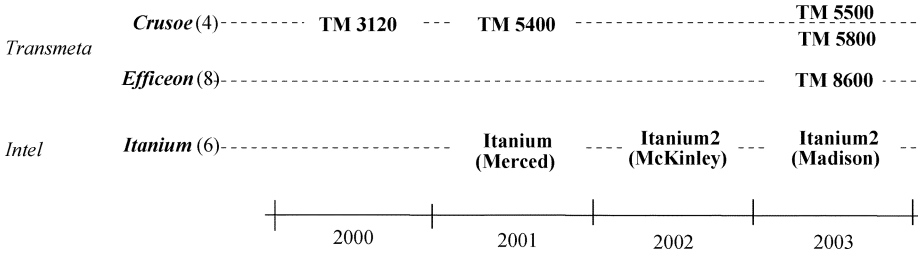


Fig. 30. General purpose VLIWs (we indicate the width of the core in brackets after line designations).

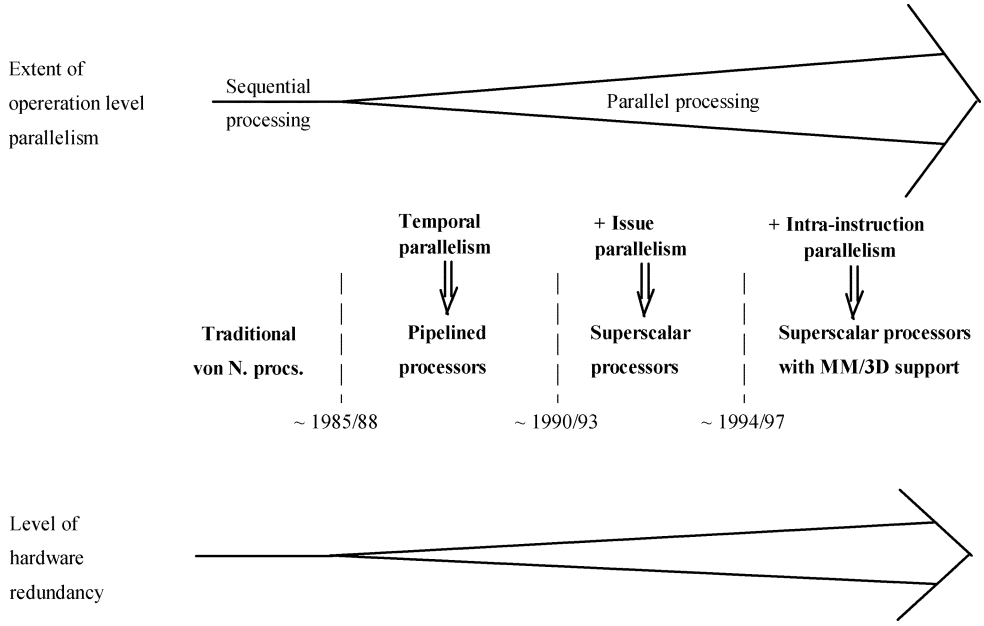


Fig. 31. Main road of the evolution of microarchitectures.

by advanced branch processing and a more powerful memory subsystem. The issue bottleneck inherent to the direct issue scheme basically limited the microarchitecture to a two- to three-wide RISC or a two-wide CISC design. However, the demand for still higher throughput called for widening the microarchitecture. This gave rise to the *second generation of superscalars* featuring dynamic instruction scheduling (buffered instruction issue), register renaming and several additional techniques to widen particular subsystems, as outlined in Section V-D4. Finally, having exhausted the extent of ILP available in general purpose programs, intrainstruction parallelism has been introduced with SIMD instructions, giving rise to the *third generation of superscalars*. However, this enhancement, effective primarily in emerging multimedia and 3-D applications, already required an extension of the ISA.

The sequence of introducing possible dimensions of parallel operation has been determined basically by two aspects: first, the drive *to keep hardware complexity as low as possible* and second, the goal of *maintaining backward compatibility* with preceding models. Presumably, it is sound to state that the price to be paid for increased performance is *increased hardware redundancy* (in terms of hardware complexity related to the number of operations performed per second).

In this respect we point out that scalar pipelined processors solely utilizing temporal parallelism make the best use of available hardware resources, since most of their pipeline stages are involved in processing instructions most of the

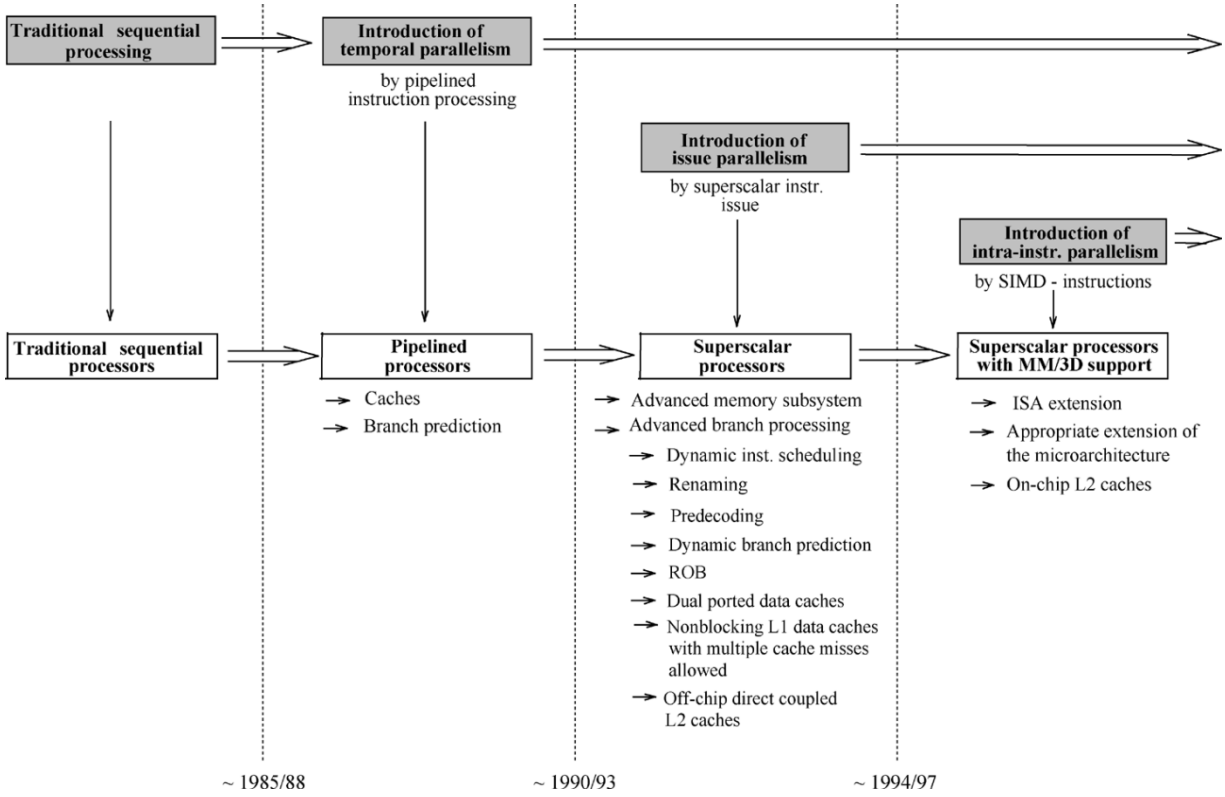


Fig. 32. Major steps in the evolution of microprocessors.

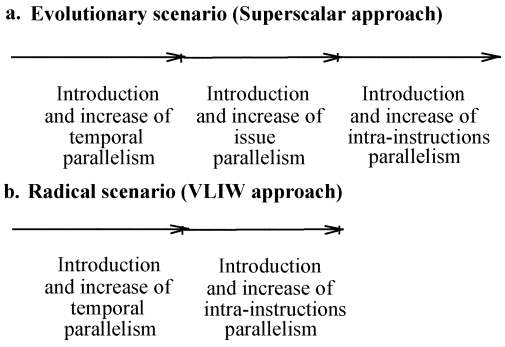


Fig. 33. Possible scenarios for the development of processors.

time. Superscalar processors that also utilize issue parallelism use their hardware resources less efficiently due to the availability of multiple (parallel) execution paths. SIMD hardware extensions—which also enable architectures to exploit intrainstruction parallelism—are the least utilized, as they are used only for multimedia and 3-D applications. In summary, we can state that higher per-cycle throughput necessarily leads to higher hardware redundancy, as indicated in Fig. 33.

We note that beyond the above-discussed evolutionary scenario, a second scenario was also open for the development of microarchitectures.

In the latter scenario, the introduction of temporal parallelism is immediately followed by the debut of intrainstruction parallelism in the form of *VLIW instructions*, as indicated in Fig. 33. Introducing multiple data operations per instruction instead of issuing and executing multiple instructions per clock cycle is obviously a viable alternative to boost throughput. However, in contrast to the *evolutionary path* that preserves backward compatibility, this scenario represents a quite "*radical*" *path* in a sense, since the introduction of multioperation VLIW instructions demand a completely new ISA. This is the key reason why early *VLIW processors* turned out to be a dead end. However, through refurbishing the VLIW concept by means of the EPIC technology [147] and harnessing the immense advances in compiler technology, Intel and HP now offer competitive 64-b six-wide VLIW cores, such as the Itanium2 [146], as a migration path for 32-b x86 superscalar CISCs as well as 64-b four- to five-wide superscalar RISCs. In the x86 scene, AMD also heated up competition by introducing its *64-b x86 ISA* extension [150] (designated as x86/64) and its related processor implementations (named Opteron for server applications and Athlon 64 for desktop processors). Nevertheless, irrespective of these recent developments, the potentials of ILP processors are becoming more and more exhausted, and any further substantial performance increase requires the utilization of available parallelism at a higher than instruction level—that is, at the *thread level* as well. However, this step of the evolution is already beyond the scope of our paper.

VIII. Conclusion

At the instruction level, microarchitectures evolved basically in *three consecutive cycles*, not unlike a spiral with three arms. Each cycle can be attributed to a new dimension of parallelism introduced into the microarchitecture. In the

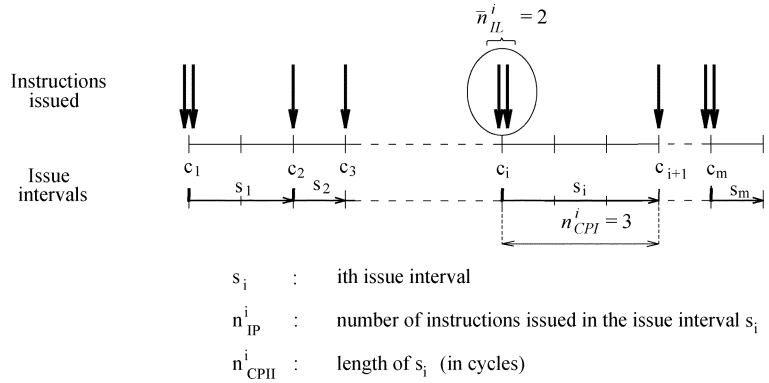


Fig. 34. Assumed model of processor operation. Arrows indicate decoded instructions issued for processing.

course of evolution *temporal*, *issue*, and *intrainstruction parallelism* were introduced one after another by means of the *basic techniques* of pipelining, superscalar issue, and SIMD extensions, respectively. However, each basic technique introduced gave rise to bottlenecks in particular subsystems of the microarchitecture. Consequently, the introduction of each basic technique calls for several *auxiliary techniques* to resolve the implied bottlenecks. Nevertheless, once the required auxiliary techniques have been introduced, the basic technique considered attains its full potential. This terminates the associated cycle of the evolution and at the same time gives rise to a new cycle in order to satisfy the incessant market demand for increased processor performance.

All the decisive aspects mentioned above constitute a framework that explains the main road of the evolution of microarchitectures at the instruction level, including the sequence of major innovations encountered. With the main dimensions of parallelism more or less exhausted at the instruction level, processor evolution will necessarily continue by utilizing parallelism at a higher than instruction level—i.e., at the thread level—as well. Therefore, the next cycle of evolution will be devoted to the thread level and to the auxiliary techniques needed to utilize the full potential of multithreading.

APPENDIX

The throughput of the processor (IPC). In order to express the throughput of the processor in terms of average number of issued IPC by operational parameters of the microarchitecture, we assume the following *model of processor operation* with reference to Fig. 34.

1) The processor operates in cycles, issuing in each cycle $0, 1 \ldots n_{\text{ir}}$ instructions, where n_{ir} is the issue rate of the processor.
2) Of the cycles needed to execute a given program, we focus on those in which the processor issues at least one instruction. We call these cycles *issue cycles*, and denote them by c_i, $i = 1 \ldots m$. The issue cycles c_i subdivide the execution time of the program into *issue intervals* s_i, $i = 1 \ldots m$, such that each issue cycle c_i gives rise to the beginning of a new issue interval s_i while the next issue cycle c_{i+1} ends it up, as indicated in the figure. Then s_1 is the first issue interval, and s_m is the last one belonging to the given program.
3) We express the operation of the processor by describing each of the issue intervals s_i, $i = 1 \ldots m$ by the following two parameters (see Fig. 34):

n^i_{IPII} the number of instructions issued in the issue interval s_i, $i = 1 \ldots m$.

n^i_{CPII} the length of the issue interval s_i $i = 1 \ldots m$, in cycles. Here n^m_{CPII} is the length of the last issue interval, which is interpreted as the number of cycles to be elapsed until the processor is ready to issue instructions again.

Then in issue interval s_i the processor issues n^i_{IPII} instructions of n^i_{CPII} cycles length. Accordingly, we obtain for the number of instructions the processor issues per cycle in the issue interval s_i (n^i_{IPC})

$$n^i_{\text{IPC}} = \frac{\overline{n}^i_{\text{IPII}}}{\overline{n}^i_{\text{CPII}}}. \tag{6}$$

Now let us consider n^i_{IPC} to be a stochastic variable, which is derived from the stochastic variables n^i_{IPII} and n^i_{CPII}, as indicated in (6). Assuming that the stochastic variables involved are independent, the average number of instructions issued per cycle $(\overline{n}_{\text{IPC}})$ can be calculated from the averages of both stochastic variables included, as follows:

$$\overline{n}_{\text{IPC}} = \frac{\overline{n}_{\text{IPII}}}{\overline{n}_{\text{CPII}}}. \tag{7}$$

Finally, we rename the terms introduced above for better readability as follows:

$$\overline{n}_{\text{IPC}} = \text{IPC} \tag{8}$$

$$\overline{n}_{\text{IPII}} = \text{IPII} \tag{9}$$

$$\overline{n}_{\text{CPII}} = \text{CPII}. \tag{10}$$

As a result, we obtain for the average number of instructions the processor issued per cycle

$$\text{IPC} = \frac{\text{IPII}}{\text{CPII}}. \tag{11}$$

ACKNOWLEDGMENT

The author would like to thank the anonymous reviewers for their valuable comments and suggestions on earlier drafts of this paper.

REFERENCES

[1] B. R. Rau and J. A. Fisher. *Intel Microprocessor Quick Reference Guide* [Online]. Available: http://developer.intel.com/pressroom/kits/processors/quickref.html

[2] ——, "Instruction level parallel processing: History, overview and perspective," *J. Supercomput.*, vol. 7, no. 1, pp. 9–50, 1993.

[3] J. E. Smith and G. S. Sohi, "The microarchitecture of superscalar processors," *Proc. IEEE*, vol. 83, pp. 1609–1624, Dec. 1995.

[4] A. Yu, "The future of microprocessors," *IEEE Micro*, vol. 16, pp. 46–53, Dec. 1996.

[5] R. Ronen, A. Mendelson, K. Lai, S.-L. Lu, F. Pollack, and J. P. Shen, "Coming challenges in microarchitecture and architecture," *Proc. IEEE*, vol. 89, pp. 325–340, Mar. 2001.

[6] A. Moshovos and G. S. Sohi, "Microarchitectural innovations: Boosting microprocessor performance beyond semiconductor technology scaling," *Proc. IEEE*, vol. 89, pp. 1560–1575, Nov. 2001.

[7] Y. Patt, "Requirements, bottlenecks, and good fortune: Agents for microprocessor evolution," *Proc. IEEE*, vol. 89, pp. 1553–1559, Nov. 2001.

[8] D. M. Tullsen, S. J. Eggers, and H. M. Levy, "Simultaneous multithreading: Maximizing on-chip parallelism," in *Proc. 22th Annu. Int. Symp. Computer Architecture*, 1995, pp. 392–403.

[9] S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, R. L. Stamm, and D. M. Tullsen, "Simultaneous multithreading: A platform for next generation processors," *IEEE Micro*, vol. 17, pp. 12–19, Sept./Oct. 1997.

[10] D. T. Marr, F. Binns, D. L. Hill, G. Hinton, D. A. Koufaty, J. A. Miller, and M. Upton, "Hyper-threading technology architecture and microarchitecture," *Intel Technol. J.*, vol. 6, no. 1, pp. 4–16, 2002.

[11] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang, "The case for a single chip multiprocessor," in *Proc. 7th Int. Conf. Architectural Support for Programming Languages and Operating Systems*, 1996, pp. 2–11.

[12] L. Hammond, B. A. Nayfeh, and K. Olukotun, "A single-chip multiprocessor," *IEEE Computer*, vol. 30, pp. 79–85, Sept. 1997.

[13] J. M. Tendler, J. S. Dodson, J. S. Fields, Jr., H. Le, and B. Sinharoy, "Power4 system microarchitecture," *IBM J. Res. Develop.*, vol. 46, no. 1, pp. 5–27, 2002.

[14] ——, *SPEC Benchmark Suite, Release 1.0*. Santa Clara, CA: SPEC, 1989.

[15] ——, SPEC CPU92 benchmarks. [Online]. Available: http://www.spec.org/osg/cpu92/

[16] ——, SPEC CPU95 benchmarks. [Online]. Available: http://www.spec.org/osg/cpu95

[17] ——, SPEC CPU2000 V1.2 documentation. [Online]. Available: http://www.spec.org/cpu2000

[18] ——, WinBench 99. [Online]. Available: http://www.zdnet.com/zdbop/winbench/winbench.html

[19] ——, SYSmark Bench Suite. [Online]. Available: http://www.babco.com

[20] J. L. Hennessy and D. A. Patterson, *Computer Architecture, A Quantitive Approach*. San Mateo, CA: Morgan Kaufmann, 2003.

[21] H. J. Curnow and B. A. Wichmann, "A synthetic benchmark," *Comput. J.*, vol. 19, no. 1, pp. 43–49, 1976.

[22] R. P. Weicker, "Drystone: A synthetic systems programming benchmark," *Commun. ACM*, vol. 27, no. 10, pp. 1013–1030, 1984.

[23] M. S. Hrishikesh, N. P. Jouppi, K. I. Farkas, D. Burger, S. W. Keckler, and P. Shivakumar, "The optimal logic depth per pipeline stage is 6 to 8 FO4 inverter delays," in *Proc. 29th Annu. Int. Symp. Computer Architecture*, 2002, pp. 1–11.

[24] A. Hartstein and T. R. Puzak, "The optimum pipeline depth for a microprocessor," in *Proc. 29th Annu. Int. Symp. Computer Architecture*, 2002, pp. 1–7.

[25] D. Anderson and T. Shanley, *ISA System Architecture*, 3rd ed. Reading, MA: Addison-Wesley, 1995.

[26] ——, *EISA System Architecture*, 2nd ed. Reading, MA: Addison-Wesley, 1995.

[27] ——, *PCI System Architecture*, 4th ed. Reading, MA: Addison-Wesley, 1999.

[28] ——, (2003) PCI technology overview. [Online]. Available: http://www.digi.com/pdf/prd_msc_pcitech.pdf

[29] V. Cuppu, B. Jacob, B. Davis, and T. Mudge, "A performance comparison of contemporary DRAM architectures," in *Proc. 26th Annu. Int. Symp. Computer Architecture*, 1999, pp. 222–233.

[30] G. S. Sohi and M. Franklin, "High bandwidth data memory systems for superscalar processors," in *Proc. 4th Int. Conf. Architectural Support for Programming Languages and Operating Systems*, 1991, pp. 53–62.

[31] T. Juan, J. J. Navarro, and O. Teman, "Data caches for superscalar processors," in *Proc. Int. Conf. Supercomputing*, 1997, pp. 60–67.

[32] W. C. Hsu and J. E. Smith, "A performance study of instruction cache prefetching methods," *IEEE Trans. Comput.*, vol. 47, pp. 497–508, May 1998.

[33] W. L. Rosch, *Hardware Bible*, 5th ed. New York: Macmillan, 1999.

[34] E. Bloch, "The engineering design of the STRETCH computer," in *Proc. East. Joint Comp. Conf.*, 1959, pp. 48–58.

[35] R. M. Tomasulo, "An efficient algorithm for exploiting multiple arithmetic units," *IBM J. Res. Develop.*, vol. 11, no. 1, pp. 25–33, 1967.

[36] R. W. Hockney and C. R. Jesshope, *Parallel Computer*. Bristol, U.K.: Hilger, 1981.

[37] T. Kilburn, D. B. G. Edwards, M. J. Lanigan, and F. H. Sumner, "One-level storage system," *IRE Trans. EC-11*, vol. 2, pp. 223–235, 1962.

[38] D. W. Anderson, F. J. Sparacio, and F. M. Tomasulo, "The IBM System/360 Model 91: Machine philosophy and instruction-handling," *IBM J.*, vol. 11, no. 1, pp. 8–24, 1967.

[39] B. J. Katz, S. P. Morse, W. B. Pohlmand, and B. W. Ravenel, "8086 microcomputer bridges the gap between 8- and 16-bit designs," *Electronics*, pp. 99–104, Feb. 16, 1978.

[40] ——, *MC68000 Family Reference Manual*. Austin, TX: Motorola Inc., 1988.

[41] G. Kane and J. Heinrich, *MIPS RISC Architecture*. Englewood Cliffs, NJ: Prentice Hall, 1992.

[42] ——, "80 286 High performance microprocessor with memory management and protection," in *Microprocessors*. Mt. Prospect, IL: Intel Corp., 1991, vol. 1, pp. 3.60–3.115.

[43] ——, "The 68 030 Microprocessor: A window on 1988 computing," *Comput. Des.*, vol. 27, no. 1, pp. 20–23, 1988.

[44] F. Faggin, M. Shima, M. E. Hoff, Jr., H. Feeney, and S. Mazor, "The MCS-4: An LSI micro computer system," in *Proc. IEEE Region 6 Conf.*, 1972, pp. 8–11.

[45] S. P. Morse, B. W. Ravenel, S. Mazor, and W. B. Pohlman, "Intel Microprocessors: 8008 to 8086," in *Computer Structures: Principles and Examples*, D. P. Siewiorek, C. G. Bell, and A. Newell, Eds. New York: McGraw-Hill, 1982.

[46] C. J. Conti, D. H. Gibson, and S. H. Pitkowsky, "Structural aspects of the System/360 Model85, part 1: General organization," *IBM Syst. J.*, vol. 7, no. 1, pp. 2–14, 1968.

[47] J. E. Smith, "A study of branch prediction strategies," in *Proc. 8th Annu. Int. Symp. Computer Architecture*, 1981, pp. 135–148.

[48] K. F. Lee and A. J. Smith, "Branch prediction strategies and branch target buffer design," *IEEE Computer*, vol. 17, pp. 6–22, Jan. 1984.

[49] D. Sima, T. Fountain, and P. Kacsuk, *Advanced Computer Architectures*. Harlow, U.K.: Addison-Wesley, 1997.

[50] K. Skadron, P. S. Ahuja, M. Martonosi, and D. W. Clark, "Branch prediction, instruction-window size, and cache size: Performance tradeoffs and simulation techniques," *IEEE Trans. Comput.*, vol. 48, pp. 1260–1281, Nov. 1999.

[51] Y. Patt, W.-M. Hwu, and M. Shebanow, "HPS, a new microarchitecture: Rationale and introduction," in *Proc. 18th Annu. Workshop Microprogramming*, 1985, pp. 103–108.

[52] D. Sima, "Superscalar instruction issue," *IEEE Micro*, vol. 17, pp. 28–39, Sept. 1997.

[53] T.-Y. Yeh and Y. N. Patt, "Alternative implementations of two-level adaptive branch prediction," in *Proc. 19th Annu. Int. Symp. Computer Architecture*, 1992, pp. 124–134.

[54] S. McFarling. (1993) Combining branch predictors. Western Res. Lab. [Online]. Available: ftp://gatekeeper.research.compaq.com/pub/DEC/WRL/research-reports/WRL-TN-36.pdf

[55] S. Duta and M. Franklin, "Control flow prediction schemes for wide-issue superscalar processors," *IEEE Trans. Parallel Distrib. Syst.*, vol. 10, pp. 346–359, Apr. 1999.

[56] N. P. Jouppi and D. W. Wall, "Available instruction-level parallelism for superscalar and superpipelined machines," in *Proc. 3rd Int. Conf. Architectural Support for Programming Languages and Operating Systems*, 1989, pp. 272–282.

[57] M. S. Lam and R. P. Wilson, "Limits of control flow on parallelism," in *Proc. 19th Annu. Int. Symp. Computer Architecture*, 1992, pp. 46–57.

[58] D. Sima, "The design space of shelving," *J. Syst. Architect.*, vol. 45, no. 11, pp. 863–885, 1999.
[59] R. Yung, "Evaluation of a commercial microprocessor," Ph. D. dissertation, Univ. California, Berkeley, 1998.
[60] L. Gwennap, "Nx686 goes toe-to-toe with Pentium Pro," *Microprocessor Rep.*, vol. 9, no. 14, pp. 1, 6–10, 1998.
[61] E. Rothenberg, S. Bennett, and J. E. Smith, "Trace cache: A low latency approach to high bandwidth instruction fetching," *Proc. 29th Int. Symp. Microarchitectures (MICRO29)*, pp. 24–35, 1996.
[62] S. J. Patel, D. H. Friendly, and Y. N. Patt, "Critical issues regarding the trace cache fetch mechanism," Univ. Michigan, Ann Arbor, Tech. Rep. CSE TR-335-97, May 1997.
[63] S. V. Adve, "Changing interaction of compiler and architecture," *IEEE Computer*, vol. 30, pp. 51–58, Dec. 1997.
[64] J. Shipnes and M. Phillips, "A modular approach to Motorola PowerPC compilers," *Commun. ACM*, vol. 37, no. 6, pp. 56–63, 1994.
[65] M. Butler, T.-Y. Yeh, Y. Patt, M. Alsup, H. Scales, and M. Shebnow, "Single instruction stream parallelism is greater than two," in *Proc. 18th Annu. Int. Symp. Computer Architecture*, 1991, pp. 276–286.
[66] M. Franklin and G. S. Sohi, "ARB: A hardware mechanism for dynamic reordering of memory references," *IEEE Trans. Comput.*, vol. 45, pp. 552–571, May 1996.
[67] G. Z. Chrysos and J. S. Emer, "Memory dependence prediction using store sets," in *Proc. 25th Annu. Int. Symp. Computer Architecture*, 1998, pp. 142–153.
[68] J. E. Smith and A. R. Pleszkun, "Implementing precise interrupts in pipelined processors," *IEEE Trans. Comput.*, vol. 37, pp. 562–573, May 1988.
[69] D. W. Wall, "Limits of instruction level parallelism," in *Proc. 4th Int. Conf. Architectural Support for Programming Languages and Operating Systems*, 1991, pp. 176–188.
[70] R. R. Oehler and M. W. Blasgen, "IBM RISC System/6000: Architecture and performance," *IEEE Micro*, vol. 11, pp. 14–17–56–62, May/June 1991.
[71] K. Diefendorff and E. Shilha, "The PowerPC user instruction set architecture," *IEEE Micro*, vol. 14, pp. 30–41, Oct. 1994.
[72] D. Hunt, "Advanced performance features of the 64-bit PA-8000," in *Proc. 40th IEEE Computer Soc. Int. Conf.*, 1995, pp. 123–128.
[73] ——, "MIPS IV instruction set architecture (white paper)," MIPS Technol. Inc., Mountain View, CA, 1994.
[74] A. Peleg and U. Weiser, "MMX technology extension to the Intel architecture," *IEEE Micro*, vol. 16, pp. 42–50, Aug. 1996.
[75] S. Fuller, "Motorola's AltiVec technology (white paper)," Motorola Inc., Austin, TX, 1998.
[76] ——, Advanced technology for visual computing: Alpha architecture with MVI (white paper). [Online]. Available: http://www.digital.com/semiconductor/mvi-backgrounder.htm
[77] D. Sweetman, *See MIPS Run*. San Francisco, CA: Morgan Kaufmann, 1999.
[78] R. B. Lee, "Subword parallelism with MAX-2," *IEEE Micro*, vol. 16, pp. 51–59, Aug. 1996.
[79] L. Kohn, G. Maturana, M. Tremblay, A. Prabhu, and G. Zyner, "The Visual Instruction Set (VIS) in UltraSPARC," in *Proc. IEEE Computer Soc. Int. Conf.*, 1995, pp. 462–469.
[80] S. Oberman, G. Favor, and F. Weber, "AMD 3Dnow! technology: Architecture and implementations," *IEEE Micro*, vol. 19, pp. 37–48, Mar./Apr. 1999.
[81] ——, Intel architecture software developers manual (SSE). [Online]. Available: http://developer.intel.com/design/Pentium4/manuals/
[82] ——, Intel architecture software developers manual (SSE2). [Online]. Available: http://developer.intel.com/design/Pentium4/manuals/index.htm
[83] ——, *DECchip 21 064 and DECchip 21 064A Alpha AXP Microprocessors Hardware Reference Manual*. Maynard, MA: DEC, 1994.
[84] ——, *Alpha 21 164 Microprocessor Hardware Reference Manual*. Maynard, MA: DEC, 1994.
[85] ——, *Microprocessor Hardware Reference Manual*. Maynard, MA: DEC, 1997.
[86] D. Leibholz and R. Razdan, "The Alpha 21 264: A 500 MIPS out-of-order execution microprocessor," in *Proc. IEEE Computer Soc. Int. Conf.*, 1997, pp. 28–36.
[87] K. Diefendorff and M. Allen, "Organization of the Motorola 88 110 superscalar RISC microprocessor," *IEEE Micro*, vol. 12, pp. 40–62, Mar./Apr. 1992.
[88] T. Asprey, G. S. Averill, E. Delano, B. Weiner, and J. Yetter, "Performance features of the PA7100 microprocessor," *IEEE Micro*, vol. 13, pp. 22–35, June 1993.
[89] R. L. Lee, "Accelerating multimedia with enhanced microprocessors," *IEEE Micro*, vol. 15, pp. 22–32, Apr. 1995.
[90] G. Kurpanek, K. Chan, J. Zheng, E. CeLano, and W. Bryg, "PA-7200: A PA-RISC processor with integrated high performance MP bus interface," in *Proc. IEEE Computer Soc. Int. Conf.*, 1994, pp. 375–382.
[91] A. P. Scott *et al.*, "Four-way superscalar PA-RISC processors," *Hewlett-Packard J.*, vol. 48, pp. 1–9, Aug. 1997.
[92] G. Lesartre and D. Hunt, PA-8500: The continuing evolution of the PA-8000 family, Hewlett-Packard Co., Palo Alto, CA, pp. 1–11, 1998.
[93] G. F. Grohoski, "Machine organization of the IBM RISC System/6000 processor," *IBM J. Res. Develop.*, vol. 34, no. 1, pp. 37–58, Jan. 1990.
[94] S. White and J. Reysa, *PowerPC and POWER2: Technical Aspects of the New IBM RISC System/6000*. Austin, TX: IBM Corp., 1994.
[95] L. Gwennap, "IBM crams Power2 onto single chip," *Microprocessor Rep.*, vol. 10, no. 11, pp. 14–16, 1996.
[96] M. Becker, "The PowerPC 601 microprocessor," *IEEE Micro*, vol. 13, pp. 54–68, Sept. 1993.
[97] B. Burgess *et al.*, "The PowerPC 603 microprocessor," *Commun. ACM*, vol. 37, no. 6, pp. 34–42, 1994.
[98] S. P. Song *et al.*, "The PowerPC 604 RISC microprocessor," *IEEE Micro*, vol. 14, pp. 8–17, Oct. 1994.
[99] D. Ogden *et al.*, "A new PowerPC microprocessor for low power computing systems," in *Proc. IEEE Computer Soc. Int. Conf.*, 1995, pp. 281–284.
[100] D. Levitan *et al.*, "The PowerPC 620 microprocessor: A high performance superscalar RISC microprocessors," in *Proc. IEEE Computer Soc. Int. Conf.*, 1995, pp. 285–291.
[101] ——, *MPC750 RISC Microprocessor User's Manual*. Austin, TX: Motorola Inc., 1997.
[102] M. Papermaster, R. Dinkjian, M. Jayfiield, P. Lenk, B. Ciarfella, F. O'Conell, and R. Dupont. POWER3: Next generation 64-bit PowerPC processor design. [Online]. Available: http://www.rs6000.ibm.com/resource/technology/index.html
[103] A. Patrizio and M. Hachman. Motorola Announces G4 Chip. [Online]. Available: http://www.techweb.com/wire/story/twb19 981 016S0013
[104] P. Y.-T. Hsu, "Designing the FPT microprocessor," *IEEE Micro*, vol. 14, pp. 23–33, Apr. 1994.
[105] ——, "R10000 microprocessor product overview," MIPS Technol. Inc., Mountain View, CA, Oct. 1994.
[106] I. Williams, "An illustration of the benefits of the MIPS R12000 microprocessor and OCTANE system architecture (white paper)," Silicon Graphics, Mountain View, CA, 1999.
[107] ——, "The SuperSPARC microprocessor technical white paper," Sun Microsyst., Mountain View, CA, 1992.
[108] D. Greenley *et al.*, "UltraSPARC: The next generation superscalar 64-bit SPARC," in *Proc. IEEE Computer Soc. Int. Conf.*, 1995, pp. 442–461.
[109] N. Patkar, A. Katsuno, S. Li, T. Maruyama, S. Savkar, M. Simone, G. Shen, R. Swami, and D. Tovey, "Microarchitecture of Hal's CPU," in *Proc. IEEE Computer Soc. Int. Conf.*, 1995, pp. 259–266.
[110] G. Goldman and P. Tirumalai, "UltraSPARC-II: The advancement of ultracomputing," in *Proc. IEEE Computer Soc. Int. Conf.*, 1996, pp. 417–423.
[111] T. Hore and G. Lauterbach, "UltraSPARC-III," *IEEE Micro*, vol. 19, pp. 73–85, Mar./Apr. 1999.
[112] D. Alpert and D. Avnon, "Architecture of the Pentium microprocessor," *IEEE Micro*, vol. 13, pp. 11–21, June 1993.
[113] ——, *The P6 Architecture: Background Information for Developers*. Mt. Prospect, IL: Intel Corp., 1995.
[114] M. Eden and M. Kagan, "The Pentium processor with MMX technology," in *Proc. IEEE Computer Soc. Int. Conf.*, 1997, pp. 260–262.
[115] ——, "P6 family of processors," in *Hardware Developers Manual*. Mt. Prospect, IL: Intel Corp., Sept. 1998.
[116] J. Keshava and V. Pentkovski, "Pentium III processor implementation tradeoffs," *Intel Technol. J.*, pp. 1–11, 2nd Quarter 1999.
[117] ——, *The Cyrix M1 Architecture*. Richardson, TX: Cyrix Corp., 1995.
[118] ——, *Cyrix 686 MX Processor*. Richardson, TX: Cyrix Corp., 1997.
[119] ——, Nx586 processor product brief. [Online]. Available: http://www.amd.com/products/cpg/nx586/nx586brf.html
[120] ——, *AMD-K5 Processor Technical Reference Manual*. Sunnyvale, CA: Advanced Micro Devices Inc., 1996.

[121] B. Shriver and B. Smith, *The Anatomy of a High-Performance Microprocessor*. Los Alamitos, CA: IEEE Comput. Soc. Press, 1998.
[122] ——, *AMD-K6-2 Processor Technical Reference Manual*. Sunnyvale, CA: Advanced Micro Devices Inc., 1999.
[123] ——, *AMD-K6-3 Processor Technical Reference Manual*. Sunnyvale, CA: Advanced Micro Devices Inc., 1999.
[124] ——, *AMD Athlon Processor Technical Brief*. Sunnyvale, CA: Advanced Micro Devices Inc., 1999.
[125] L. Gwennap, "Alpha 21 364 to ease memory bottleneck," *Microprocessor Rep.*, vol. 12, pp. 12–15, 1998.
[126] ——, (2000, Apr.) PA-RISC 8x00 family of microprocessors with focus on PA-8700 (white paper). Hewlitt-Packard. [Online]. Available: http://ftp.parisc-linux.org/docs/whitepapers/PA-8700wp.pdf
[127] J. M. Tendler, J. S. Dodson, J. S. Fields, Jr., H. Le, and B. Sinharoy, "Power4 system microarchitecture," *IBM J. Res. Develop.*, vol. 46, no. 1, pp. 5–26, 2002.
[128] ——, MIPS R14000 microprocessor chip set. [Online]. Available: http://www.sgi.com/processors/r14k/
[129] ——, Technical info R 16 000. [Online]. Available: http://www.sgi.com/workstations/fuel/tech_info.html
[130] ——, "UltraSPARC III Cu," in *User's Manual*. Mountain View, CA: Sun Microsyst. Inc., May 2002.
[131] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel, "The microarchitecture of the Pentium4 processor," *Intel Technol. J.*, pp. 1–13, 1st Quarter 2001.
[132] ——, "Microarchitecture for AMD Athlon64 and AMD Opteron processors," in *Software Optimization Guide*. Sunnyvale, CA: Advanced Micro Devices Inc., Apr. 2003.
[133] M. Mittal, A. Peleg, and U. Weiser, "MMX technology overview," *Intel Technol. J.*, pp. 1–10, 3rd Quarter 1997.
[134] ——, 3D Winbench 99-3D lightning and transformation test. [Online]. Available: http://developer.intel.Com/procs/perf/PentiumIII/ed/3dwinbench.html
[135] ——, (2004, Feb.) Intel Pentium4 processor and Intel 875P and Intel 850E chipset performance brief. [Online]. Available: http://www.intel.com/performance
[136] R. P. Colwell, R. P. Nix, J. O. Donell, D. B. Papworth, and P. K. Rodman, "A VLIW architecture for a trace scheduling compiler," *IEEE Trans. Comput.*, vol. 37, pp. 967–979, Aug. 1988.
[137] B. R. Rau, C. D. Glaser, and R. L. Picard, "Efficient code generation for horizontal architectures: Compiler techniques and architectural support," in *Proc. 9th Annu. Int. Symp. Computer Architecture*, 1982, pp. 131–139.
[138] J. A. Fisher, "Very long instruction word architectures and the ELI-512," in *Proc. 10th Annu. Int. Symp. Computer Architecture*, 1983, pp. 140–150.
[139] B. R. Rau, D. W. L. Yen, W. Yen, and R. A. Towle, "The Cydra 5 departmental supercomputer," *IEEE Computer*, vol. 22, pp. 12–35, Jan. 1989.
[140] ——, *TM1000 Preliminary Data Book*. New York: Philips Electron. Corp., 1997.
[141] ——, *TMS320C6000 Technical Brief*. Dallas, TX: Texas Instrum., Feb. 1999.
[142] ——, *SC140 DSP Core Reference Manual*. Murray Hill, NJ: Lucent Technol., Inc., Dec. 1999.
[143] S. Hacker, "Static superscalar design: A new architecture for the TigerSHARC DSP processor (white paper)," Analog Devices Inc., Norwood, MA, 2004.
[144] ——, "MAJC architecture tutorial (white paper)," Sun Microsyst. Inc., Mountain View, CA, Sept. 1999.
[145] H. Sharangpani and K. Arora, "Itanium processor microarchitecture," *IEEE Micro*, vol. 20, pp. 24–43, Sept./Oct. 2000.
[146] ——, (2002, July) Inside the Intel Itanium 2 processor (technical white paper). [Online]. Available: http://h21007.www2.hp.com/dspp/ddl/ddl_Download_File_TRX/ 1,1249,952,00.pdf
[147] M. S. Schlansker and B. R. Rau, "EPIC: Explicitly Parallel Instruction Computing," *IEEE Computer*, vol. 33, pp. 37–45, Feb. 2000.
[148] ——, "Crusoe processor," Transmeta Corp., Santa Clara, CA, 2000.
[149] ——, "Efficeon," Transmeta Corp., Santa Clara, CA, 2004.
[150] ——, *AMD 64-bit Technology*. Sunnyvale, CA: Advanced Micro Devices Inc., 2001.

Dezsö Sima (Member, IEEE) received the M.Sc. degree in electrical engineering and the Ph.D. degree in telecommunications from the Technical University of Dresden, Dresden, Germany, in 1966 and 1971, respectively.

He has taught computer architecture at the Technical University of Dresden; at Kandó Polytechnic, Budapest, Hungary; and at South Bank University, London, U.K. and been a guest lecturer on computer architectures at several European universities. He was the first professor to hold the Barkhausen Chair at the Technical University of Dresden. He is currently the Dean of the John von Neumann Faculty of Informatics at Budapest Tech, Budapest. He has authored more than 50 papers and is the principal author of *Advanced Computer Architectures: A Design Space Approach* (Harlow, U.K.: Addison-Wesley, 1997), which is used at universities in more than 30 countries in advanced architecture courses. His research interests include computer architectures and computer-assisted teaching and learning.

Dr. Sima is an Institution of Electrical Engineers (IEE) Fellow. Between 1994 and 2000, he was President of the John von Neumann Computer Society in Hungary.