VLIW: old architecture of the new generation

by M. Len and I. Vaitsman

The VLIW (Very Long Instruction Word) architecture is rooted in the parallel microcode used yet at the dawn of computer engineering and in Control Data CDC6600 and IBM 360/91 supercomputers. In 1970 most computation systems were equipped with additional vector signal processors using VLIW-like long instructions flashed in ROM. These processors were used for fast Fourier transformation and other computational algorithms. The first real VLIW computers were mini-supercomputers released at the beginning of 1980 by MultiFlow, Culler and Cydrome, but they didn’t meet with much success. The scheduler of calculations and program pipelining were offered by Fisher and Rau (Cydrome). Today this is a base of the VLIW compiling technology.

The first VLIW computer MultiFlow 7/300 used two ALUs for whole numbers and two ALUs for floating-point numbers, and a decision box - all the stuff were based on several chips. Its 256bit instruction word contained 8 32bit op codes. The integer processing units could implement two operations at a clock 130ns long (that is, 4 in all with two ALUs), which provided up to 30 MIPS in case of whole numbers. Also, it was possible to combine hardware solutions to get computational systems from either 256bit or 1024bit. The first VLIW computer Cydrome Cydra-5 used a 256bit instruction and a special mode supporting a sequence of 6 40bit ops that is why its compilers could generate a mixture of a parallel code and a serial one. While the VLIW-BM used several chips the Intel i860 could be considered the first VLIW processor based on a single chip. But the i860 is not a normal VLIW system - it has just a program-controlled instruction pairing in contrast to the later hardware-controlled one, which became a part of super-scalar processors. Besides, FPS computers (AP-120B, AP-190L and later models under the FPS trade mark), also based on the VLIW architecture, were quite popular some time ago. Moreover, there were such "canonical" machines as Kertsev's M10 and M13, and Elbrus-3, - and although the latter didn't succeed on the market, it was a decent stage in the development of the VLIW. I must say that efficiency of a VLIW processor mostly depends on a compiler (not so much on equipment) because optimization of sequence of operations has a greater effect than a frequency growth.

Recently we witnessed a "confrontation" of the CISC vs. RISC, and now a new battle comes along involving VLIW and RISC. Strictly speaking, the VLIW and the superscalar RISC are not real rivals. The latter is not actually the architectural property but a certain way of implementation. We might soon see superscalar VLIW processors supporting "squared parallelism", i.e. a combination of the explicit static parallelism and the implicit dynamic one. But at the present stage of the processor development there are no evident ways of combination of static and dynamic reordering. That is why the Itanium/Itanium2 should be regarded not in terms of VLIW vs. CISC (or VLIW vs. O3E), but as synchronous VLIW vs. Out-Of-Order RISC. Also remember that Intel-HP alliance came up with a separate name for its architecture - EPIC that stands for Explicitly Parallel Instruction Computing.

Although the VLIW appeared at the earliest stage of the computer industry (Turing designed a VLIW computer back to 1946), it hasn't been commercially successful. Now Intel has implemented some ideas of the VLIW in the Itanium line. But a marginal performance boost and speed-up of calculations compared to the existent classical "RISC-inside CISC-outside" architectures can be obtained by moving the intellectual functions from hardware to software (compiler). So, the success of the Itanium/Itanium2 depends primarily on the software means, - this is the problem. And lack of much interest to the Itanium, which is not a newcomer on the market shows that the problem does exist.

EPIC: Explicitly Parallel Instruction Computing

The conception of Explicitly Parallel Instruction Computing determines a new type of architecture, which is able to compete against the RISC. This approach is going to simplify hardware and extract as much as possible hidden parallel computing on the instruction level using WIW (Wide Issue-Width) and DPL (Deep Pipeline-Latency), than it's possible with the VLIW or superscalar strategies. EPIC simplifies two key aspects. First of all, it cancels checking for dependences between operations, which are declared by the compiler as independent. Secondly, the long architecture lets us avoid a complicated logic of out-of-order execution of operations in favour of the order of instruction issuing determined by the compiler. Moreover, the EPIC improves the ability of the compiler to generate plans of execution statically at the expense of various code transitions during compilation, which are not correct in the serial architecture. Earlier solutions approached this aim mainly at the expense of hardware complication, which, in the long run, turned, into an obstacle preventing further performance boost. EPIC was developed exactly to reach a higher degree of parallel instruction computing with an acceptable hardware complexity.

A higher performance level can be obtained by increasing a signal speed and a denser arrangement of chip's functional components. After that execution of programs can be accelerated due to implementation of a specific type of parallel computing. Thus, the ILP (Instruction-Level Parallelism) is based on processors and compilation techniques, which speed up operation at the expense of parallel fulfilment of separate RISC operations. The ILP
based systems use programs written in traditional higher-level languages, for serial processors, and the hidden parallelism can be revealed automatically due to the respective compiling technology and hardware.

The fact that these techniques do not call for more efforts from application programmers is very important because the traditional microprocessor parallelism assumes that programmers have to rewrite their applications. Parallel instruction-level processing is the only reliable solution, which allows for performance growth without fundamental application remaking.

Superscalar processors are a realization of the ILP processor for serial architectures a program for which must not and, actually, cannot deliver precise information on parallelism. Taking into account that the program doesn’t have precise information on ILP, the problem of revealing of parallelism must be solved on a hardware level, and the hardware, in its turn, must develop an action plan for revealing the hidden parallelism. VLIW processors are an example of the architecture for which the program gives precise data on parallelism - the compiler reveals parallelism in the program and notifies the hardware on which operations do not depend on each other. This information is crucial for the hardware as it lets it know which operations can be started in the same cycle without further check-up. The EPIC architecture is evolved from the VLIW architecture and absorbed many conceptions of the superscalar architecture, though they are optimized for the EPIC. This is actually a philosophy, which determines creation of ILP processors, as well as a set of characteristics of the architecture, which support this base. From this point of view the EPIC is similar to the RISC: a determining class of architectures following the general basic principles. There are multiple architectures of instruction suites (ISA) for the RISC, and the same way we can have more than one ISA for the EPIC. Depending on the EPIC characteristics used by the EPIC ISA architecture, the latter can be optimized for various applications, for example, for general-purpose systems or built-in devices. The first example of the commercial EPIC ISA was the IA-64 architecture.

The code for superscalar processors contains a sequence of instructions, which gives a correct result if it's fulfilled in the established order. The code determines a sequential algorithm and, except a definite instruction set, it doesn’t determine exactly the nature of hardware it’s going to be used on and a precise time order when the instructions are going to be fulfilled. In contrast to programs for superscalar processors, the VLIW offers a strictly defined plan (POE - Plan Of Execution, created statically during compilation). The code determines when each operation is to be fulfilled, which functional units are to work and which registers are to contain operands. Having a full idea on the processor, the VLIW compiler creates such a POE to get a required ROE (Record Of Execution, a sequence of events which really take place when the program is working). The compiler delivers the POE (via the architecture of the instruction set which describes parallelism precisely) to the hardware, which implements it. This plan allows the VLIW to use relatively simple hardware, which can reach a high ILP level. In contrast to the VLIW, the superscalar architecture builds up the POE dynamically using a serial code. Although such approach complicates the physical implementation, the superscalar processor creates a plan using advantages of those factors, which can be defined only during the execution.

One of the objectives of development of the EPIC was to retain the principle of the POE static development realized in the VLIW, and at the same time to enrich it with capabilities typical of the superscalar processor, which allow the new architecture account for dynamic factors that traditionally constrain the parallelism peculiar to the VLIW. The philosophy of the EPIC follows the certain principles, which account for these objectives.

The first principle is development of the plan of execution during compilation. The EPIC entrusts the computer with the POE development. Although the architecture and physical implementation can hinder the compiler it gong this, the EPIC processors provide functions, which help the compiler to make up the POE. The EPIC processor’s behavior must be predictable and controlled for the compiler. Dynamic out-of-order execution can tangle up the situation and the compiler will cease to "understand" how its solutions influence a real record of execution created by the processor, that is why it must be able to predict actions of the processor, and this complicates the task even more. In this case we should have a processor, which does what it is told to by the program. Creation of the plan during compilation is aimed at reordering of the source serial code to make possible to use all advantages of parallelism of the application and spend the hardware resources the most beneficial way with the execution time being as low as possible. Without the respective architecture’s support such reordering can affect correctness of the program. So, taking into account that the EPIC hands in creation of the POE to the compiler, it must provide the required architectural capabilities that support intensive code reordering during compilation.

The next principle is probabilistic assessment used by the compiler. The EPIC’s compiler runs into a tough problem when creating a POE: information of a certain type, which has a great effect on the record of execution, is uncovered only during execution of the program. For example, the compiler doesn't know for which branch is going to be executed after the transition operator, when the planned code is to pass the base blocks and which of the graph's paths is going to be chosen. Besides, usually it’s impossible to create a static plan, which simultaneously optimizes all paths in the program. The situation is also undetermined when the compiler doesn't know whether the links are going to lead to the same place in the memory. If yes, they must be addressed in turn; if no, it can be done in the random order. With such indeterminacy a certain end result becomes the most probable. One of the most important principles of the EPIC in this situation is that it allows the compiler to operate with probabilistic assessment - it creates and optimizes the POE for most probable cases. However, the EPIC provides the architectural support such as Control and Data Speculation to guarantee
that the program is correct even if the primitive conjectures are wrong. When a conjecture turns out to be wrong, execution of the program evidently gets slower. Such a performance effect can be seen on the program's plan, for example, when in spite of a highly optimized program area, the code is executed in the less optimized one. The performance may fall down in case of Stalls which cannot be seen on the plan - certain operations which are most probable, and thus, match the optimized case, are executed at the highest speed, but they hold up the processor to guarantee the correct execution when a less probable case occurs.

When the plan is created, the compiler delivers it to the hardware. The ISA must have quite rich capabilities to be able to inform on the compiler's solutions such as when every operation is to be initiated and what resources are to be used (in particular, there must be a method to indicate what operations are to be initiated simultaneously). An alternative approach can be a serial program created by the compiler and then dynamically reordered by the processor to get a required record. However, in this case hardware is not relieved from dynamic planning. When delivering the POE to the hardware it's very important to provide necessary information in time. One of the examples can be a jump operation which, when used, requires that the instructions are prefetched and preexecuted before the jump. Instead of letting the hardware to decide when this must be done and which jump address must be used, such information is delivered to the hardware accurately and in proper time with the code, according to the basic principles of the EPIC. The microarchitecture makes other decisions as well which are not related with code execution but which affect the execution time. For example, cache-memory hierarchy management and decisions on which data are necessary to maintain the hierarchy and which must be replaced. The cache controller functioning algorithm usually provides such rules. The EPIC expands the principle, which determines that the compiler makes up a POE so that it can be possible to control these mechanisms of the microarchitecture. The architectural features supporting software management of mechanisms, which are usually controlled by the microarchitecture, are provided.

**VLIW Soft hardware complex**

The VLIW architecture is implementation of the conception of the internal parallelism in microprocessors. Their efficiency can be improved by increasing either the clock speed or the number of operations fulfilled per clock. In this case it's necessary to use "fast" technologies (for example, using gallium arsenide instead of silicon) and such architectural solutions as depth pipelining (pipelining within one clock when the entire die is used, not its separate parts). To increase the number of operations executed at a clock one chip must house a lot of functional processing units, and reliable parallel execution of instructions must be supported, which allows enabling all units simultaneously. Here, reliability means that all the results are correct. Let's take as an example two expressions which are interconnected the following way: $A = B + C$ and $B = D + E$. The variable $A$ will have different values depending on the order the expressions are handled in (first $A$ then $B$ or vice versa), but the program implies just one, definite, value. And if we execute these expressions simultaneously the correct results can be guaranteed only to a certain degree of probability.

Sequence scheduling is a quite difficult problem, which is to be tackled when designing a modern processor. To find dependences between machine instructions, the superscalar architectures use a special hardware solution (for example, the Intel's P6 and post-P6 architectures utilizes the ReOrder Buffer, ROB). However, the size of such hardware scheduler increases in the geometric progression when the number of functional units rises, which, at the long run, can take the entire processor's die. That is why superscalar projects limit the number of instructions processed in a cycle to 5-6. The current realization of the VLIW can't boast of the complete packet fill - the real load of 6-7 instructions per clock is similar to that of the leaders among RISC processors. Another approach makes possible to hand over scheduling to the software, as it's realized in the projects with VLIW. A "smart" compiler must find all instructions in the program which are entirely independent, combine very long lines (long instructions) and then send them for simultaneous execution to the functional units the number of which is not smaller than the number of operations in such a long instruction. Very long instructions (VLIW) take usually 256-1024 bits, but there can be smaller samples. The size of fields, which code operations for each functional unit, is much smaller in such metainstruction.

**Logical layer of the VLIW processor**

The VLIW processor shown below can execute up to 8 operations per clock and work at a lower clock speed much more effective than existent superscalar chips. Additional functional units can lift up the performance (at the expense of reducing conflicts for resources) without sophisticting the chip considerably. However, such extension is limited by the physical abilities, i.e. by the number of read/write ports, which provide simultaneous access of functional units to the register files, and interdependences, which grow up in the geometrical progression as the number of functional units increases. Besides, the compiler must parallelize the program to a definite level to load up each unit - this is the key factor, which limits utilization of this architecture.
This hypothetical instruction has 8 operative fields, each executing a traditional trioperand RISC-like instruction <operation> <receiver register> <source register> (MOV AX BX) and can control a specific functional unit with the minimal decoding.

Let's take up the IA-64 as one of the examples of incarnation of the VLIW. With time this architecture can crowd out the x86 (IA-32) as a class. Nevertheless, the necessity to develop complicated compilers for the IA-64 and difficulties in creation of optimized machine codes can cause lack of experts working in IA-64 Assembler, especially at the primary stages.

The most important feature of the IA-64 compared to the RISC is Explicitly Parallel Instruction Computing (EPIC), which brings in into the IA-64 some elements resembling the architecture of the very long instruction word, which are called bundles. So, in both architectures the explicit parallelism is implemented on the instruction level, which controls simultaneous operation of functional execution units.

In this case the bundle is 128bit long and includes 3 41bit fields for instructions and a 5bit template slot. It’s assumed that instructions of the bundle can be executed in parallel by different FUs. Possible interdependences, which prevent parallel execution of instructions from the same bundle, are reflected in the template field. But it’s not claimed that instructions from different bundles cannot be executed in parallel. However, according to the specified parallel execution level, which assumes up to 6 instructions per clock, we can see that at least two bundles can be processed at the same time.

The template indicates instructions of which type are located in the bundle slots. In general, instructions of the same type can be executed in more than one type of functional units. The template also assigns slots whose instructions, when started, must be waited for completion by the instructions of the following fields. The order of slots in a bundle (more important are on the right) corresponds to the byte order (Little Endian), but data in the memory can be arranged in the Big Endian order (more important on the left), which is set with a special bit in the register of the user mask.

Rotation of registers is a particular case of register renaming used in most superscalar processors with an out-of-order speculative instruction execution. In contrast, rotation of registers in the IA-64 can be controlled on the software level. Usage of this mechanism in the IA-64 allows avoiding overheads connected with storage/recovery of a great number of registers when invoking and leaving subroutines, but it’s sometimes necessary to save and restore static registers explicitly coding respective instructions.

By the way, the system of instructions of the IA-64 is rather unique. One of its peculiarities is speculative instruction execution and usage of predicates - exactly this subset defines exclusiveness of the IA-64. All such instructions can be divided into instructions of operation with a register stack, integer instructions, instructions of comparison and operation with predicates, memory access instructions, jump instructions, multimedia
instructions, interregister move instructions, "miscellaneous" instructions (operations with lines and count of bits in a word) and floating-point instructions.

The hardware implementation of the VLIW processor is very simple: there are several small functional units (addition, multiplication, branching etc.) connected to the processor bus and several registers and cache memory units. There are two reasons why the VLIW architecture can be interesting for the semiconductor industry. The first one is that the die provides more space for processing units (instead of, for example, branch prediction units). And the second one is that the VLIW processor can be high-speed as the maximum processing speed depends only on internal features of the functional units. Besides, in certain conditions the VLIW can fulfil old CISC instructions more effectively than the RISC, because programming of the VLIW processor resembles microcode development (only low-level language which can thoroughly program a hardware layer synchronizing operation of logical gates and data exchange buses and controlling data transfer between functional units).

When the PC memory was still too expensive the programmers saved on it using complicated instructions of the x86 processor like STOS and LODS (indirect write/read from/to memory). The CISC deals with such instructions as microprograms flashed into the ROM and executed by the processor. The RISC architecture excludes usage of a microcode dealing with instructions on the hardware level - the instructions of the RISC processor are an integral part of the instruction set used in the CISC. VLIW takes another approach - it takes a microcode generation procedure out of the processor (as well as from the execution stage) and transfers it to the compiler, to the stage of development of the executable code. As a result, emulation of the instructions of the x86 processor such as STOS turns out to be very effective as the processor receives ready macros for execution. But it comes along with many problems, as creation of an effective microcode is very difficult. Also, a "smart" compiler can bring the VLIW architecture to life. This is what limits usage of computers based on the VLIW architecture: they are currently used only in vector processors (for scientific calculations) and signal ones.

**Principle of operation of the VLIW compiler**

The rising interest to the VLIW as an architecture, which can be used for general-purpose calculations, has given a considerable impulse to development of technique of the VLIW compilation. Such compiler packs groups of independent operations into very long instruction words to provide fast start-up and more effective execution by functional units. The compiler first detects all interdependences between data and then defines how to unbind them. It's mostly done by reordering the whole program - its blocks are moved from one place to another. This approach differs from that used in the superscalar processor which uses a special hardware solution during execution of the program to detect interdependences (optimizing compilers certainly improve operation of the superscalar processor but do not make it "tied" to them). Most superscalar processors can find interdependences and plan parallel execution only inside base program blocks (group of serial program operators which do not have a stall or logical branching allowable only at the end). Some reordering systems initiate extension of the scanning area without limiting it with base blocks. For a higher parallelism degree the VLIW computers must monitor operations from different base blocks to place these operations into the same long instruction (their "viewing angle" must be wider than that of the superscalar processors), - this is provided with tracing. Tracing is the most optimal (for a certain set of source data) route in the program (the data mustn't intersect for a correct result), i.e. the route which covers the parts suitable for parallel computing (these parts are formed also by delivering the code from other places of the program), after that they must be combined into long instructions and sent for execution. The computation scheduler implements optimization on the level of the whole program, not its separate base blocks. For the VLIW, as well as for the RISC, branching in the program prevents its effective execution. While the RISC uses a hardware solution for branch prediction the VLIW leaves this for the compiler. The compiler uses information collected by profiling the program, though future VLIW processors will have a small hardware extension providing collection of statistics for the compiler during execution of a program, which is very important in cycle operation with a variable set. The compiler predicts the most suitable route and plans the pathway as one big basic block, then repeats the procedure for all other appeared program branches, and so forth until the end of the program. It takes also some other intellectual steps when analyzing the code, such as program cycle unrolling and IF transformation during which all logic transitions are temporarily removed from the section undergoing the tracing. While the RISC can only check the code for branching, the VLIW compiler moves it from one place to another until it detects branching (according to the tracing), but it also supports rollback to the previous program state. It also can be done in the RISC processor but the price/effectiveness ratio is too high. Certain hardware added to the VLIW processor can help the compiler. For example, operations with several branches can be put into the same long instruction and, therefore, executed in the same clock cycle. That is why execution of conditional operations, which depend on the previous results, can be implemented on the software level. The cost of performance growth of the VLIW processor is much lower than the cost of compilation - that is why primary expenses fall on shoulders of the compilers.

**VLIW: the reverse of the medal**

Realization of the VLIW stumbles over other problems as well. The VLIW compiler must know internal features of the processor's architecture in detail coming down to the structure of the functional units. Therefore, at
launch of a new version of the VLIW processor with a greater number of processing modules (or even with the same number but with another performance level) the old software may need recompilation. The manufacturers of VLIW processors cannot narrow packets to let the old programs work reliably on the new devices. For example, there are 6-instruction packets today, and the next version can't do with just 6 functional units even for the sake of 2-3 times frequency increase. Apart from the fact that a program compiled for the eight-channel VLIW can't be executed on the six-channel architecture without special quite expensive means, it's necessary to rewrite the compiler as well. From this standpoint it's rational to use a three-instruction word in the IA64 system - such inconvenient limitation will let vary the number of execution units in the IA64 processors in future. And if it wasn't necessary to recompile software with a transition from the 386 to the 486, now it has to be done. One of compromise solutions is to divide the compilation process into two stages. All the software must be prepared in the hardware-independent format using an intermediate code, which will then be translated into the computer-dependent code only in course of installation on the end-user hardware. The OS with its Architecture-Neutral Distribution Format (ANDF) realizes such an approach. But the cross-platform software hasn't justified the hopes yet. First of all, it requires ports, which "explain" the compiler what should be done in each case when compiling a given program on the given platform (and exactly in the given OS). Secondly, the cross platform software is not a Speed Demon at all, and it even works slower than that written for a certain platform of the identical program class.

Another problem is the static nature of optimization provided by the VLIW compiler. It's hard to say what the program is going to do when it runs into unforeseen dynamic situations, such as input/output expectation, during compilation. The VLIW architecture answered the call of scientific and technical organizations which need a highly efficient processors for calculations, but it isn't so suitable for object-oriented and event-driven programs which take the greatest share in the IT industry. It's still hard to check that the compiler fulfills such difficult transforms correctly and reliably. On the contrary, the Out-Of-Order RISC processors can adapt to a given situation in the most advantageous manner.

It's obvious today that the developers of modern efficient VLIW processors step aside from the original idea of a pure VLIW architecture. At least, Intel left in the Itanium family a possibility to execute the classical x86 code, though at the expense of noticeable performance losses in such a processor operating mode. I can say that the processor giant surrendered the bastion of the pure VLIW under the pressure of software giants. But solving a difficult problem of interaction of hardware and software in the VLIW architecture requires thorough preliminary investigations. At present, the theoretical studies in the sphere of algorithms of automatic program paralleling haven't achieved much: for example, if it's necessary to parallel a program (for example, to insert SMP optimized parts), it will be done manually and by highly professional programmers. Besides, there are a lot of tasks, which require enormous efforts or can't be parallelized at all.

Let's take the Crusoe from Transmeta. Its principle of operation is based on dynamic recompilation of the compiled x86 code for the VLIW architecture. But if in development of an effective VLIW compiler for higher-level languages the developers face such big problems, what can be said about a compiler of such chaotic and unpredictable computer code as the x86, especially considering that it's optimized during compilation for other architectural features.

The Crusoe is actually used only in the x86 emulation mode though there are no limitations for emulation of another code. And all programs and the OS work above the low-level software called Code Morphing, which controls translation of the x86 code into bundles of 128bit. The Crusoe has its own terminology that defines a fine architecture level where bundles are called Molecules and 32bit subinstructions in bundles are called Atoms. Each molecule contains 2 or 4 atoms. For compatibility with the instruction format, when not all atomic slots of a molecule are used in the binary translation, empty fields are filled with a NOP instruction (No OPeration). So, due to two molecule types, each of them contains not more than one NOP. Up to 4 VLIW subinstructions can be executed at a clock. One of the most important architectural features of the VLIW core of the Crusoe is the relatively short pipelines: an integer one for 7 stages and a floating-point one for 10 stages. Such processor can be used for emulation of different architectures, but some peculiarities of its microarchitecture are designed exactly for effective emulation of the x86 code. An important distinguishing feature is lack of performance losses with the approximately equal processor frequencies in x86 emulation. First the processor decodes the x86 code in per-byte interpretation mode, but if the code is going to be executed several times, the morphing mechanism translates it into the optimal sequence of molecules, and the result of the translation is cached for reuse.

Transmeta tried to jump over the technology that goes ahead of the time. The Transmeta's solution is a really fantastic technical achievement. They have actually made a real-time working technology of dynamic compilation of cross platform software. If the system is able to work so efficiently on the fly translating the x86 code into the internal form, then just imagine what results can be obtained with an optimized program. So, binary compatibility is possible and quite effective.

On the other hand, even Gordon Moore claimed that growth of the processor frequencies would soon cease to follow the rule of thumb he had formulated and will become much slower. In this situation the performance of classic systems can soon become limited on the physical level, which will make for more rapid development of other ways of performance increasing, one of which is development of new well-paralleling algorithms, - and this will allow the VLIW show itself in all its beauty.
Final words

Comparison of advantages of the VLIW and superscalar architectures has been being discussed much by experts in realization of instruction-level parallelism (ILP). Supporters of both conceptions bring the discussion to contraposition of simplicity and limited capabilities of the VLIW and dynamic capabilities of the superscalar systems. But such contraposition is wrong. Both approaches have their advantages and it doesn’t make sense to bring up the issue of their competition. Development a plan of execution during compilation is important for a high instruction-level paralleling degree, even for a superscalar processor. Besides, compilation comes along with ambiguity, which can be resolved only in course of execution, and the processor requires dynamic mechanisms to solve this problem. The supporters of the EPIC agree with both positions - the EPIC supports these mechanisms on the architectural level so that the compiler can control the architectural mechanisms using them selectively where it’s possible. Such wide abilities help the compiler use rules of controlling these mechanisms more optimally than the equipment.

The basic principles of the EPIC, coupled with capabilities of the architecture, provide means for determining ILP architectures and processors which allow for a higher ILP degree with less complicated equipment in various fields of application. The IA-64 is an example of how principles of the EPIC can be used for general-purpose computational systems where compatibility of the object code is critical. But with time the EPIC will take a more significant share on the market of high-performance integrated systems. This sphere makes stronger requirements for the price/performance ratio and lower requirements for compatibility on the object modules level, which forces using more adjustable architectures. Thus, HP Labs started the research project PICO (Program In, Chip Out) developing a prototype which can automatically design an architecture and a microarchitecture of the EPIC processor adapted for a certain task using integrated applications written in C. So, the EPIC favours a stable growth of performance of general-purpose microprocessors executing certain applications without rewriting the code fundamentally.

The main problem of the VLIW processors is evidently a contradiction between the sequential logic of applications and nature of most computational algorithms, and a parallel nature of execution of the VLIW processor. Limitation of the number of functional units in classical processors is caused not just by impossible effective out-of-order execution but also by the limited number of instructions, which can be executed simultaneously. But if the optimiser had no strict time constrains (like in classical superscalar processors), it would be possible to pack instructions quite densely, but only on some algorithms. For example, there are event-driven programs with a constantly changing data set, which do not easily undergo static optimization. Such a situation when changing a data set results into a fatal performance losses, for example, in operation with databases, is quite possible. To prevent it it’s necessary to go along all algorithm’s branches in all possible combinations of data and generate an optimal code for each possible situation. But such solution will make a compiled application much greater, and swapping of different (in situation) parts of the optimal code for execution from RAM will increase the memory load. That is why the Itanium2 has the increased size of the cache memory in comparison with the Itanium, despite the several times wider data bus.

It could be interesting to pack into one-bundle instructions from different algorithm branches to execute them simultaneously to get an operable result. This approach excludes an idea of branch prediction and is much simpler in terms of performance, but the code size and memory load are evidently much greater. On the other hand, if the branching is higher than binary, the instruction decoding units will get a greater load as well - at the greater density of instruction swapping (both branches instead of one) there can be competition for the maximum possible bandwidth of the Front Side Bus and/or cache memory (Back Side Bus).

The preliminary planning of sequence of operations will allow using the entire bandwidth of the data bus in more cases than for classical processors. But such a high and predictable effectiveness of the interface requires a new approach for building the system.

The most interesting (and dangerous for the conception) aspect in development of the VLIW architectures is an issue of static code optimization in combination with possible changes in the internal processor architecture. It’s obvious that a program optimized for one device can turn out to be useless for operation on the next generation of processors. Nevertheless, there is a roundabout way. The idea is that a distributive of a certain application contains not a code ready for execution or source texts (OSF) but results of a certain compilation stage: with described dependences, formalized description of processes, unrolled cycles, i.e. the most difficult part of operation of a compiler which also defines quality of its operation. During installation of a program an installer invokes a specific for a given processor compiler which converts this intermediate code into an executable one the most optimal for the given processor (and configuration) way. So, it is possible to make a final compiler (and options of its operations) based on libraries describing used subsystems. Such approach can squeeze out the maximum performance from each configuration of a computer and do away with such conception as a soft hardware program. Any OS can be installed on any processor if required drivers (descriptions) are provided. But this is a too ideal scheme.
An alternative solution can be a Crusoe-like system with dynamic compilation of applications delivered during execution in the form of an intermediate code. On the one hand, a performance level will be lower because of the additional load in the form of dynamic compilation, but on the other hand, simplicity and flexibility of such solution (the dynamic translator can account for behavior of applications executed simultaneously in the system) make this approach quite attractive. Besides, such dynamic compiler can use information on the equipment during its operation.

Well, I can assume that Transmeta ruined the unwillingness to use current modern memory technologies. It's obvious that the RAM load will rise several times with dynamic compilation of applications. In other words, using a memory system with a low peck throughput Transmeta loses much in applications addressing intensively the system RAM, while it achieves a parity in performance in the applications working mostly inside the processor. In the future, Transmeta must be aimed at extension of the number of functional units of the processor with possible parallel execution of two and more program branches (of different programs) in the same core using more scalable and efficient system memory.