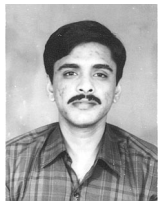
Very Long Instruction Word Processors

S Balakrishnan



S Balakrishnan is currently a research scientist at Philips Research, Eindhoven, The Netherlands. His research interests include design and performance evaluation of processors, compiler optimization, and operating systems.

Explicitly Parallel Instruction Computing (EPIC) is an instruction processing paradigm that has been in the spotlight due to its adoption by the next generation of Intel Processors starting with the IA-64. The EPIC processing paradigm is an evolution of the Very Long Instruction Word (VLIW) paradigm. This article gives an overview of VLIW processor architecture.

Introduction

Increase in speeds at which processors are clocked have led to higher performance benefits – applications now run faster; it is now possible to run realistic graphics, interactive games and simulators. This is primarily because of improvements in semiconductor technology in terms of both speed and circuit density. However, advances in chip fabrication technology alone do not account for the performance gains achieved by modern day processors. In addition to higher clock speeds, to achieve faster execution of conventional programs written for sequential machines, commercial processors like the Intel Pentium Processor have modified the processor architecture to exploit parallelism in a program. These processors seek out independent operations/instructions in a sequential program and execute them in parallel to exploit what is commonly called instruction level parallelism (ILP).

Before we attempt to understand what ILP is all about, let us consider the central processing unit (CPU) of a computer. In general, the CPU consists of two main subsystems – the data-manipulating subsystem and the control subsystem. The data-manipulating subsystem consists of basic arithmetic and logical units (ALUs) like the adders, multipliers, comparators, logic operation units, etc. The data-manipulating units are generally

Processor hardware detects dependencies between operations and schedules them for simultaneous execution to exploit Instruction Level Parallelism.

referred to as functional units. The control subsystem on the other hand steers computations through the data-manipulating subsystem after interpreting the instructions in a computer program. The complete data-manipulating subsystem including the interconnections and functional units is called the data path and the complete control subsystem is called the control path. In addition to the data and control subsystems the processor also consists of the system clock that synchronizes activities in the processor.

In the previous paragraph we had mentioned that the control subsystem interprets the instructions in a computer program. Elaborating further, the instructions that constitute a program can in general be encoded in a compact manner to specify the operation to be performed. To understand how this is done, suppose that the data path includes functional units to perform exactly four operations such as addition, subtraction, multiplication, and division. An instruction can select one operation and turn off the rest. A compact encoding of this can be achieved by setting aside two binary digits (bits) to specify each operation uniquely. The four possible values (00, 01, 10 11) represented using two bits can then be decoded using a circuit called the two-to-four decoder to activate one of its four output lines to select the appropriate functional unit.

Instruction Level Parallel Processors

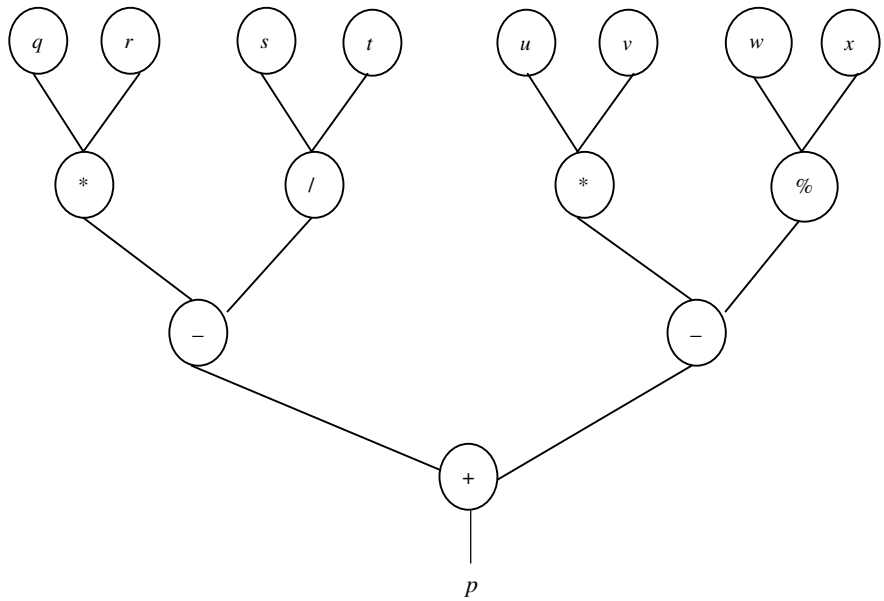
With this background, we can now attempt to explore the world of ILP processors and understand what makes them tick. *Box* 1 describes with an example the nature of instruction level parallelism. This also gives an idea of what processors look for when they attempt to exploit ILP. One approach is to delegate the job of detecting and exploiting ILP entirely to the control subsystem of the processor. The program is usually not expected to convey any explicit information regarding parallelism. The underlying hardware detects dependencies between operations and schedules them for simultaneous execution to exploit ILP. The number of instructions that can be examined simulta-

Box 1. What is Instruction Level Parallelism?

Consider the following statement written in a high level language like C.

$p = q * r - s/t + u * v - w\%x;$

The operator % is the modulus operator in C. Here it yields the remainder of the division of w by x. the *, – and the + operators are the multiply, subtract and addition operators, respectively. The expression on the right hand side can be represented by the following tree (expression tree as compiler writers would call it).



The goal of parallel processing is to execute multiple operations simultaneously on independent hardware units to reduce the overall execution time of a program. Instruction level parallel processing is a combination of software and hardware techniques to reduce the overall program execution time by identifying and executing simultaneously independent instructions from a stream of sequential instructions. In the figure, observe that the four operations at the top level of the tree can be carried out in parallel provided there are multiple arithmetic units. The results of these operations are inputs for operations that can again be performed in parallel. However, the number of independent operations here are two. Current day processors indeed have multiple arithmetic units to support this kind of parallelism. Because this kind of parallelism exists at the level of instructions, it is called instruction level parallelism (ILP). Compiler technology also is advanced enough to perceive this parallelism and schedule instructions for parallel execution on the multiple functional units available in hardware.

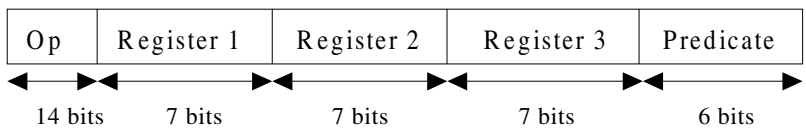
VLIW processors, relies exclusively on the compiler's ability to detect independence between instructions and schedule operations accordingly.

neously – this is called the window of instructions, for ILP is hence limited by the complexity of the hardware and the real-time nature of the problem. This is the approach taken by superscalar processors. The other approach – which is the one taken by VLIW processors, relies exclusively on the compiler's ability to detect independence between instructions and schedule operations accordingly. As there are many more independent operations than dependent operations in a program, it is impractical to explicitly specify all independent operations. Instead, a small subset of the independent operations is specified and packaged as a single VLIW instruction. Note that in contrast to the limited size of the window of instructions in a superscalar processor, the compiler for a VLIW processor can potentially examine the entire program to detect and schedule independent instructions. *Box* 2 gives an example of the instruction encoding of a VLIW processor.

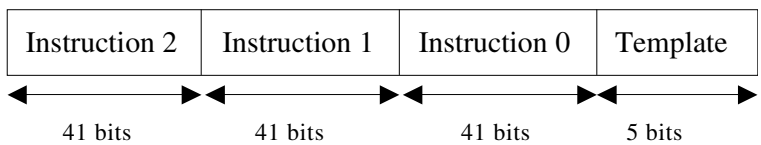
In essence, ILP processors differ in the proportion of the area of a silicon chip dedicated to the data and control paths. On one hand we could have a processor with a large and complex control path and a relatively small data path while on the other hand we could have a processor with a large data path and an extremely simple control path. The former approach is the one taken by superscalar processors while VLIW processors take the latter approach. VLIW processors with its relatively simple control subsystem are much easier to build and have been used successfully in digital signal processors. Parallelism in signal processing applications in contrast to general purpose applications are easy to detect at compile time. Again, the relatively simple control logic makes it convenient to support large amounts of hardware parallelism. On the other hand, parallelism in general purpose applications are hard to detect at compile time. The superscalar processor's hardware (control subsystem) for detecting parallelism and dynamically scheduling operations allows runtime detection and exploitation of parallelism in applications where parallelism is hard to detect at compile time. The Intel Pentium processor is an example of a processor that uses a

Box 2. The IA 64 Instruction Encoding.

The IA-64 (named Itanium by Intel) is a VLIW processor. It has 128 general purpose and the same number of floating point registers. An operation encoding hence uses 7 bits each to specify the two source operands and another 7 bits to specify the destination operand. The type of operation itself is encoded using 14 bits. Many operations also use a predicate argument that takes up another 6 bits since there are 64 predicate registers. The predicate registers store a bit, depending on whose truth value the processor decides to either execute the concerned operation or skip its execution (execute a *noop*). This accounts for a total of 41 bits to specify an operation.



IA-64 Operation encoding



IA-64 VLIW instruction encoding

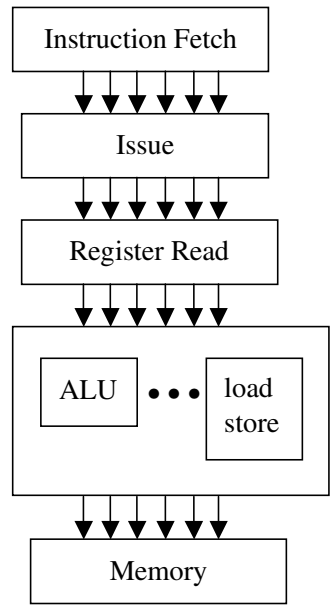
Each VLIW instruction (*bundle* in IA-64 terminology) consists of 128 bits. This means that each VLIW instruction can accommodate three 41 bit operations leaving five bits of the instruction encoding free. These five bits are used for the template which assists in decoding and routing the instructions and also the location of stops that mark the end of a group of instructions that can execute in parallel.

superscalar instruction execution engine at its core. Examples of VLIW processors include the Philips Trimedia processor used for media processing and the Texas Instruments' C6X processors for digital signal processing applications. *Box* 3 discusses the microarchitecture of a generic VLIW processor.

ILP techniques have been hugely popular because an application programmer need not be aware of these techniques i.e., the programmer need not explicitly specify the parallel portions of a code to exploit ILP. This allows code written for a processor to be carried over to the next generation of its implementation without any modification. A key advantage of superscalar

Box 3. VLIW Processor Organization

The schematic diagram depicts the various stages of a VLIW processor pipeline. The pipeline consists of a fetch unit, the issue stage, the register read stage, the execute stage and the memory stage. The fetch stage fetches instructions from the cache. In this stage, current day processors (like the IA-64) also incorporate a branch prediction unit. The branch prediction unit predicts the direction of branch instructions and speculatively fetches instructions from the predicted path. This is necessary to keep the processor pipeline active in the presence of branches in the code. The compiler can still provide hints to the hardware for determining the direction of a branch (taken or not-taken). Notably absent from the processor pipeline is the decode stage because of the extremely simple hardware to dispatch instructions to the functional units. The register read stage reads the contents of the source operands of the instruction from the large register file. The execute stage of the pipeline then executes the operation on the functional units and the memory stage reads/writes the results from/to the memory subsystem.



processors is that they provide object code compatibility over generations of a processor architecture. Object code compatibility allows object binaries compiled for previous generations of an architecture to run on new machines without modification. Providing object code compatibility is important because users would want to upgrade hardware while running applications that are not easily recompiled for a new machine. While a superscalar processor, by its very design, implicitly addresses binary compatibility across generations of a processor architecture, in a VLIW processor the run-time environment or the operating system should provide support to explicitly provide binary code compatibility over generations of a processor architecture. In addition, the aggressive compiler techniques used for code scheduling tend to bloat up the size of an object code compared to that of a superscalar processor. In the following sections we will discuss the important issue of binary compat-

Providing object code compatibility is important because users would want to upgrade hardware while running applications that are not easily recompiled for a new machine.

ibility that has to be expressly addressed before delivering a VLIW processor. Such a compatibility is essential for the commercial success of VLIW processors.

Binary Compatibility

As mentioned earlier, in a VLIW processor, the entire responsibility of scheduling correctly the instructions of a program lies with the compiler. The techniques used for scheduling instructions require the compiler to be aware of the functional unit latencies (The number of clock cycles between the initiation of an operation and its completion). Problems hence arise with binary compatibility between generations of the same architecture. Various techniques have been suggested to solve this problem. Techniques like *split-issue*, which are hardware-based solution have been mostly restricted to research studies. We will not delve into this technique due to limitations of space. Similarly, problems arise if different generations of the same architecture have different hardware parallelism. This has not been considered as a problem since new generations have a tendency of having greater hardware parallelism.

Binary Translation: This is another technique applicable to the problem of binary compatibility but has been used in the context of designing simple architectures with low power consumption and execute x86 binaries (see *Box* 4). This technique uses software to translate an existing binary into a binary that can be executed on a new target architecture (A VLIW processor in the Crusoe processor). Translation can be either static or dynamic. Static translators translate program offline (when the program is not running) using the execution profiles of programs. Dynamic translators have runtime overheads but are more attractive since it can adapt to the runtime behavior of the program by caching pieces of translated code that are executed frequently. Moreover, the cached pieces of translated code can be subject to optimizations that would have been otherwise impossible because of the lack of information about the runtime behavior of the program.

Suggested Reading

[1] T M Conte, Superscalar and VLIW Processors, in *Handbook of Parallel and Distributed Computing*, (A Y Zomaya, ed.), McGraw-Hill, New York, 1995.

[2] V Rajaraman and C Siva Ram Murthy, *Parallel Computers – Architecture and Programming*, (Chapter 3 – Instruction Level Parallel Processing), Prentice Hall of India, New Delhi, 2000.

[3] Harsh Sharangpani and Ken Arora, Itanium Processor Architecture, *IEEE Micro*, pp. 24-43, Sept.-Oct. 2000.

[4] Linda Geppert and Tekla S Perry, Transmeta's Magic Show, *IEEE Spectrum*, Vol. 37, No. 5, May 2000.

[5] Jerry Huck and others, Introducing the IA-64 Architecture, *IEEE Micro*, pp. 12-22, Sept.-Oct. 2000.

Box 4. The TRANSMETA Crusoe VLIW Processor

The TRANSMETA Crusoe processor can run software that runs on IBM PC compatible personal computers which use a Pentium processor even though the architecture of a Crusoe processor nowhere resembles that of an Intel Pentium processor. The Crusoe processor is a VLIW processor with special hardware to support x86 emulation. This is in contrast with the core of an Intel Pentium processor which is a superscalar execution engine.

One of the primary design goal of the Crusoe processor is low power consumption. This would enable the use of these processors in mobile systems like laptop personal computers allowing many hours of operation times between battery recharges. A conventional Pentium processor's superscalar core has hardware to detect dependencies between operations, schedule and retire operations. This greatly complicates the design of the system, increasing clock rate, adding costs and increasing the power consumption. The simplicity of a VLIW processor on the other hand ameliorates these problems significantly.

To achieve binary compatibility with an x86 programs, the Crusoe processor relies on a software technique called dynamic binary translation. An x86 program is dynamically translated to execute on the TRANSMETA VLIW processor. This translation scheme has been dubbed *code morphing* by TRANSMETA. In this scheme, the software (also called the virtual machine manager since it presents a x86 virtual machine to an x86 program) uses a combination of interpretation and translation to speed up program execution.

Conclusion

Microprocessor vendors and computer manufacturers are now targeting processors using the VLIW paradigm for their next generation products. For example, COMPAQ, a company that already has processors like the Alpha 21264 and 21364 RISC processors in their computers have recently announced that after 2003 all their servers are going to be Itanium (IA64) processor based. This can be considered a wise move on the part of COMPAQ. This is because the 'bells-and-whistles' that go into a current day superscalar processor to support runtime detection and exploitation of ILP reaches a point of diminishing returns for high instruction issue rates (more than eight instructions per clock cycle). It is highly likely that VLIW processors might be resurrected precisely for the reason why RISC computers were first advocated – control simplicity.

Address for Correspondence
S Balakrishnan
Philips Research The
Netherlands (Natlab)
Prof.Holstlaan 4 (WDC-3.10)
5656 AA Eindhoven
The Netherlands.
Email: balki@c4.com