Understanding EPIC Architectures and Implementations

Mark Smotherman
Dept. of Computer Science
Clemson University
Clemson, SC, 29634
mark@cs.clemson.edu

Abstract- HP and Intel have recently introduced a new style of instruction set architecture called EPIC (Explicitly Parallel Instruction Computing), and a specific architecture called the IPF (Itanium Processor Family). This paper seeks to illustrate the differences between EPIC architectures and former styles of instruction set architectures such as superscalar and VLIW. Several aspects of EPIC architectures have already appeared in computer designs, and these precedents are noted. Opportunities for traditional instruction sets to take advantage of EPIC-like implementations are also examined.

1. Introduction

Instruction level parallelism (ILP) is the initiation and execution within a single processor of multiple machine instructions in parallel. ILP is becoming an increasingly more important factor in computer performance. It was first used in the supercomputers of the 1960s, such as the CDC 6600 [34], IBM S/360 M91 [36], and IBM ACS [31], but such efforts were for the most part dropped in the 1970s due to the an apparent lack of parallelism in programs generated by then-existing compilers [35] and due to the less attractive performance / implementation-complexity tradeoffs necessary for ILP as compared to simpler cache-based processors, such as the IBM S/360 M85, and as compared to multiprocessor systems.

By the 1980s and 1990s, instruction level parallelism once again became an important approach to computer performance. Alan Charlesworth, Josh Fisher, and Bob Rau were leaders in experimenting with VLIW (very long instruction word) architectures, in which sophisticated compilers uncovered independent instructions within a program and statically scheduled these as multiple concurrent operations in a single wide instruction word. Charlesworth led efforts at FPS (Floating Point Systems) for attached array processors programmed in VLIW style [4].

Fisher led efforts at Yale on a VLIW machine called the ELI-512 and later helped found Multiflow, which produced the Multiflow Trace line of computers [6]. Rau led efforts at TRW on the Polycyclic Processor and later helped found Cydrome, which produced the Cydra-5 computer [25]. Other early VLIW efforts include iWarp and CHoPP.

In contrast to these VLIW-based efforts, other companies were exploring techniques similar to those used in the 1960s where extra hardware would dynamically uncover and schedule independent operations. This approach was called “superscalar” (the term was coined by Tilak Agerwala and John Cocke of IBM) to distinguish it from both traditional scalar pipelined computers and vector supercomputers. In 1989, Intel introduced the first superscalar microprocessor, the i960CA [21], and IBM introduced the first superscalar workstation, the RS/6000 [33]. In 1993 Intel introduced the superscalar Pentium, and since the mid-1990s the AMD or Intel processor in your desktop or laptop has relied on both clock rate and the superscalar approach for performance.

After a few years of operation, Cydrome and Multiflow both closed their doors after failing to establish a large enough market presence in the crowded minisupercomputer market of the 1980s. HP hired Bob Rau and Mike Schlansker of Cydrome, and they began the FAST (Fine-grained Architecture and Software Technologies) research project at HP in 1989; this work later developed into HP’s PlayDoh architecture. In 1990 Bill Worley at HP started the PA-Wide Word project (PA-WW, also known as SWS, SuperWorkStation). Josh Fisher, also hired by HP, made contributions to these projects [28].

In 1992, Worley recommended that HP seek a manufacturing partner for PA-WW, and in December 1993 HP approached Intel [8,28]. Cooperation between the two companies was announced in June 1994, and the companies made a joint presentation of their plans at the Microprocessor Forum in October 1997. The term EPIC (Explicitly Parallel Instruction Computing) was coined to describe the design philosophy and architecture style envisioned by HP, and the specific jointly designed instruction set architecture was named IA-64. More recently, Intel has preferred to use IPF (Itanium Processor Family) as the name of the instruction set architecture. Itanium is the name of the first implementation (it was previously called by the project codename Merced) [29], and currently Itanium-based systems can be purchased from HP, Dell, and Compaq, with many other system manufacturers committed to selling Itanium-based systems.

2. Three Major Tasks for ILP Execution

Processing instructions in parallel requires three major tasks: (1) checking dependencies between instructions to determine which instructions can be grouped together for parallel execution; (2) assigning instructions to the functional units on the hardware; and, (3) determining when instructions are initiated (*i.e.*, start execution) [27]. (Note: This departs from the earlier Rau and Fisher paper [24]; the three tasks identified there are: determine dependencies, determine independencies, and bind resources.) Four major classes of ILP architectures can be differentiated by whether these tasks are performed by the hardware or the compiler.

	Grouping	**Fn unit asgn**	**Initiation**
Superscalar	Hardware	Hardware	Hardware
EPIC	Compiler	Hardware	Hardware
Dynamic VLIW	Compiler	Compiler	Hardware
VLIW	Compiler	Compiler	Compiler

Table 1. Four Major Categories of ILP Architectures.

Table 1 identifies the four classes of ILP architectures that result from performing the three tasks either in hardware or the compiler. A superscalar processor is one with a traditional, sequential instruction set in which the semantics (*i.e.*, meaning) of a program is based on a sequential machine model. That is, a program's results should be the same as if the instructions were individually processed on a sequential machine where one instruction must be completed before the next one is examined. A superscalar processor includes the necessary hardware to speed up program execution by fetching, decoding, issuing, executing, and completing multiple instructions each cycle, but yet in such a way that the meaning of the program is preserved. The decoding and issuing of multiple instructions requires dependency-checking hardware for instruction grouping, decoding and routing hardware for assignment of instructions to function units, and register scoreboard hardware for timing the initiation of instruction execution. The dependency-checking hardware does not scale well ($O(n^2)$) and has been seen as a limit to the width of multiple instruction issue in superscalars.

At the opposite extreme is VLIW. The three responsibilities for ILP are each assigned to the compiler. The implementation of a VLIW computer uses long instruction words that provide a separate operation for each function unit on each cycle (similar to horizontal microprogramming). The width of the instruction word depends on the number of function units; *e.g.*, Multiflow produced machines with long instruction words up to 28 operations wide. Groups of independent operations are placed together into a single VLIW, and the operations are assigned to function units by position in the given fields within the long instruction word ("slotting"). The initiation timing is bound by the instruction word in which an operation appears; all operations in a VLIW start execution in parallel.

A sequence of long instruction words thus defines the *plan of execution* for a particular program on a particular implementation, the plan being specified by the sequence of VLIW instructions cycle by cycle [28]. It is the responsibility of the compiler to determine which operations can be grouped together and where they must be placed in this sequence of long instruction words. However, this also represents the Achilles heel of VLIW architectures: the problem of compatibility between implementations. Code compiled for one implementation with a certain set of function units with certain latencies will not execute correctly on a different implementation with a different set of function units and/or different latencies (although there have been studies directed at providing compatibility, *e.g.*, see [7]). In contrast, compatibility is not a problem with superscalars, and this is a major reason for their popularity.

Table 1 suggests intermediate architectures between superscalars and VLIWs with varying amounts of compiler responsibility (see also [27]). If the compiler determines the grouping of independent instructions and communicates this via explicit information in the instruction set, we have what Fisher and Rau termed an "independence architecture" [24] or what is now known as the EPIC architecture style [28]. EPIC retains compatibility across different implementations as do superscalars but does not require the dependency checking hardware of superscalars. In this manner, EPIC can be said to combine the best of both superscalar and VLIW architectures. The first EPIC architecture appears to be Burton Smith's Horizon (in 1988), which provided an explicit lookahead count field of the distance to the next dependent instruction [19], although Lee Higbie sketched an EPIC-like approach some ten years earlier (in 1978) in which concurrency control bits are added to the instruction format and set by the compiler or programmer [13].

Another category of ILP architecture is one in which the grouping and function unit assignment is done by the compiler, but the initiation timing of the operations is done by hardware scheduling. This style is called dynamic VLIW [26], and it has some advantage over traditional VLIW since it can respond to events at run time that cannot be handled by the compiler at compile time. For example, early VLIW designs did not include data caches, since a cache miss would disrupt the sequence of long instruction words by invalidating the compiler's assumption of latency

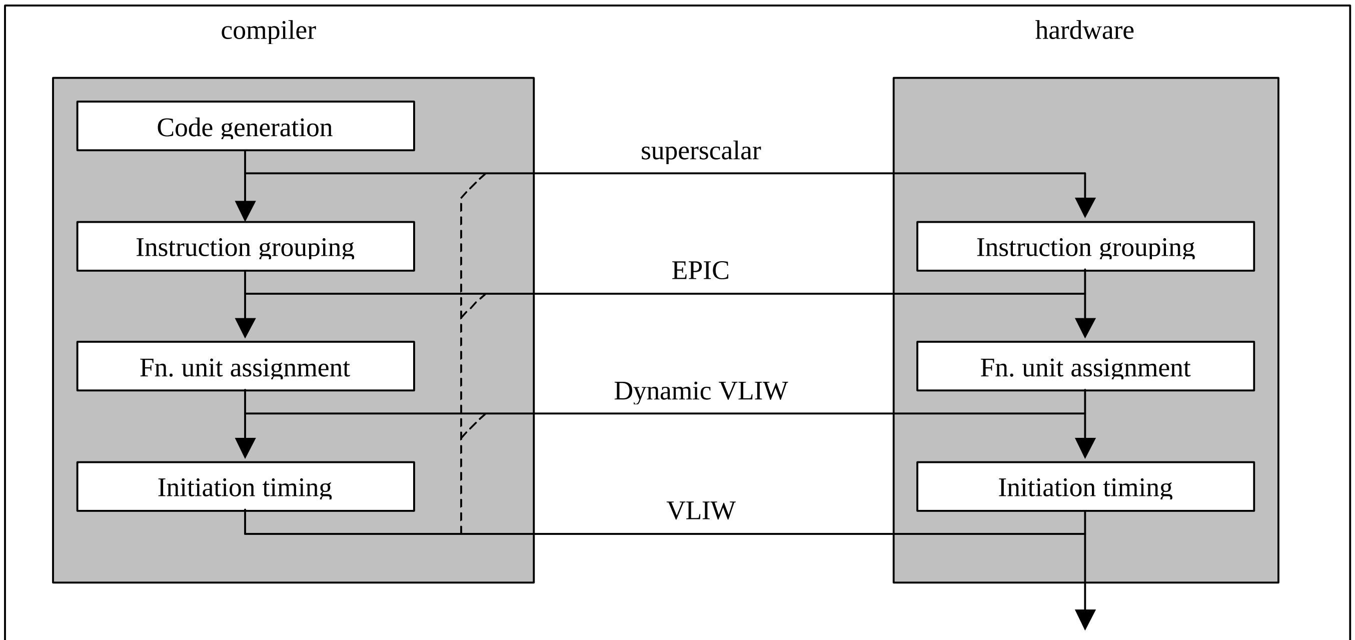

Figure 1. Graphical Depiction of the Three Major Tasks.

for load instructions. Thus in a simple dynamic VLIW approach, we can add load-miss interlock to otherwise bare hardware and stall the entire machine on a data cache miss.

Along these lines, Rau paid special attention to memory latency in the Cydra 5 design by use of a “memory collating buffer” which handled the early, and possibly out-of-order, arrival of values loaded from memory so as to preserve the static memory access latency assumptions made by the compiler; late arrivals delayed the entire machine [25,26]. (See also the discussion of LEQ semantics in [27].)

A more complicated architecture handles run-time events not by merely delaying the initiation of the next group, but by adding what is essentially dynamic scheduling hardware for the individual operations within each VLIW. (Although called dynamic VLIW by Rudd [26] and others, because of the complexity of the hardware this approach might actually be considered as a fifth category.) Instruction execution is split into two (or three) phases, with the first phase statically scheduled to read the registers, compute a result, and write the result to a temporary results buffer. The second phase will move results from the buffer into the register file. (Note the extra hardware and buffering, similar to what is found in a superscalar processor.) Rudd’s simulations suggest that there is little performance to be gained from introducing this level of complexity [26].

Figure 1 is a revision of Figure 2 from Rau and Fisher [24] using the responsibilities identified above, and shows the three responsibilities as performed by the compiler or by the hardware. The horizontal lines demonstrate the four levels at which information about the program can be given to the hardware. At the top level, a traditional instruction set is used and the hardware must perform the three tasks. There is no information in the instruction set to convey independent instruction groups, function unit assignment, or instruction timing.

The dashed lines within the compiler box indicate that for best performance, the compiler may go ahead and do all three tasks as required for best performance on a particular implementation, but supply the instructions in a less semantically-rich instruction set. In such a case, the hardware has to rediscover the independent groups among the instructions that the compiler has already arranged within the instruction stream, and it has to repeat the function unit assignment and instruction initiation timing.

As an example of the benefit of scheduling even for a superscalar processor, consider the HP PA-8000. It will run code generated for any PA-RISC 1.1 or 2.0 processor, but Holler reports that SPECint95 benchmarks ran 38% faster and SPECfp95 benchmarks ran 53% faster when specifically compiled for the PA-8000 as compared to running those same benchmarks but as compiled for the previous generation PA-7200 [14].

The other levels in Figure 1 at which programs can be conveyed to the hardware add more information to the instruction set and thus require less hardware. For example, let us assume a machine with two load/store units, an integer ALU, and a branch unit, with latencies 2, 2, 1, 2, respectively. If we wish to perform a simple addition, C = A + B, the code given to a superscalar would be something like this:

```
Load R1,A
Load R2,B
Add   R3,R1,R2
Store C,R3
```

The hardware has to determine that the loads are independent and can be grouped together, while the add is dependent on both and must be in a separate group. Likewise the store is dependent on the add and must be placed in a third group. The hardware will assign the instructions to the different function units based on the operation codes, and the register scoreboarding hardware will govern the initiation of the add and store. (Note: because of special forwarding paths or cascaded function units, some superscalar processors like the IBM RS/6000 and the TI SuperSPARC can start the execution of certain pairs of dependent instructions at the same time.)

If we look at the corresponding VLIW program, we see that the compiler has completely planned out the grouping, function unit assignment, and initiation timing. This is the complete plan of execution and relies on a particular implementation and on particular latencies. Thus, this VLIW program would be invalid for an implementation with only one load/store unit, whereas the traditional code for a superscalar processor as given above would run without any changes being necessary.

ld/st unit 0	ld/st unit 1	integer alu	branch unit
Load R1,A	Load R2,B	nop	nop
nop	nop	nop	nop
nop	nop	Add R3,R1,R2	nop
Store C,R3	nop	nop	nop

The VLIW program also illustrates a difficulty of low utilization of the long instruction word fields. Of the 16 total fields in the four long instruction words, 12 are empty and have a no-operation placed in them. Multiflow recognized this inefficiency and provided a compression scheme, in which VLIW programs existed on disk and in main memory in an encoded format [6]. Long instruction words from the program were expanded to the traditional fixed-length VLIW format when they were fetched into the instruction cache. Several VLIW processors now use similar compression schemes, including the Lucent StarCore, Philips TriMedia, Sun MAJC, and TI C6x processors.

If we add register scoreboarding to handle dynamic events like cache misses, either on a long-instruction-word-wide basis or on a function-unit-by-function-unit basis, we move to a dynamic VLIW architecture. Note that in this case we can omit the second long instruction word above, since it only has nops. This further improves the utilization of instruction memory. Two computers that can be placed in this category appeared in the late 1980s: the Intel i860 microprocessor [18] and the Apollo Domain DN10000 workstation. In each computer, an integer pipeline can run in parallel with a floating-point pipeline. The instruction formats in these computers include a bit to specify an integer and a floating-point instruction pair that can be initiated in parallel (they are slotted in a fixed order in memory to correspond to the function unit assignment).

The EPIC style of instruction set for this example must have grouping information, such as a count of independent instructions. For example, if we add a Horizon-like lookahead count (given in parentheses for all but the last instruction), we obtain:

```
(2)  Load R1,A
(1)  Load R2,B
(1)  Add  R3,R1,R2
(.)  Store C,R3
```

The hardware can use the lookahead count to group the two loads together. Note that hardware function unit assignment and hardware instruction initiation timing are still required.

An analogy to these distinctions that might be helpful in presenting these ideas in the classroom is the example of designing and building a simple wooden stool. The design represents a program, and the construction and assembly of the stool (*e.g.*, top, two legs, cross-brace) represent the operations. The designer will send the plan to the woodshop, which represents the processor. If there are several machines and workers in the woodshop (*i.e.*, multiple function units), a shop foreman would set up a complete plan of building the stool. This plan would determine which parts could be constructed or assembled in parallel, which machines or tools would be used, and when construction or assembly activities would start. *E.g.*,

table saw	band saw	hand saw	hammer
Cut brace	Cut leg 0		
Cut top	Cut leg 1		Nail leg 0 to brace
			Nail leg 1 to brace
			Nail legs to top

To correspond to the superscalar approach, the designer walks to the wood shop and hands the design to the shop foreman, who must then do the planning there in the shop as construction proceeds. To correspond to the VLIW approach, the designer and the shop foreman are the same person; the design includes the detailed plan of building as illustrated above. This plan is necessarily shop-specific, since some shops might not have a band saw. Instead, in this second shop a hand saw must be used, and the time for cutting the legs will lengthen, thereby forcing the nailing to start later. The plan is thus not compatible across wood shops. (A dynamic VLIW analog might be where the time to complete hand sawing is unknown in the second shop and nailing activities start only when sawing on certain parts is completed.)

To complete the analogy for the EPIC approach, the dependencies can be given in the design. For instance, the cutting of the brace, top, and legs are all independent; but, nailing cannot start until at least two parts are completed. The design and independence information make no assumptions about what particular shops tools will be present (*i.e.*, at least one cutting-type tool is assumed, but the specific type is not necessarily set down in the plan), nor assumptions about the length of time required to construct or build. The independence information assists the foreman in the shop in setting up an efficient plan of building.

3. Characteristics of EPIC Architectures and Historical Precedents

3.1. Explicit Parallelism

As described above, explicit information on independent instructions in the program is a major distinguishing feature of EPIC architectures. In the IPF architecture, three 41-bit instructions are packaged together into a 128-bit "bundle", which is the unit of instruction fetch. A 5-bit template identifies the instruction type and any architectural stops between groups of instructions. In little-endian format, a bundle appears as:

Instruction 2	Instruction 1	Instruction 0	Template
127	86	45	4 0

Bundles can have zero, one, or at most two stops. Instruction groups (*i.e.*, sets of independent instructions) thus can span instruction bundles. Nops may be needed to pad out the bundles in some cases. The instruction type (one of six types: integer alu, non-alu integer, memory, floating-point, branch, and extended) can help in function unit assignment and routing during decoding, but this information provides type information rather than specific function unit identification. Thus, it is not in the dynamic or traditional VLIW category. (Note that not all combinations of instruction type and stop boundaries are available -- which would have required an 11-bit template to encode 6^3*2^3 cases.) S. Vassiliadis at IBM proposed a similar instruction bundling scheme, called SCISM, in the early 1990s [37].

Schlansker and Rau list five other attributes of EPIC architectures beyond instruction grouping [28]. The first two deal with eliminating and/or speeding up branching, the third with cache locality management, and the final two with starting load instructions as early as possible.

3.2. Predicated execution

To avoid conditional branches, each instruction can be conditioned or *predicated* on a true/false value in a predicate register. Only those instructions with a true predicate are allowed to write into their destination registers. Thus, if-then-else sequences can be compiled without branches (called "if conversion"). Instructions from each side of the decision are predicated with one of two inversely-related predicate registers and can be executed in parallel. (If predicate values are available in time, an implementation can delete instructions with false predicates in the decode or issue stages.)

IPF provides 64 predicate registers. Each register can hold one bit (true or false) and is set by compare instructions. In the normal case, a compare instruction writes to two predicate registers, one with the result of the compare and one with the inverted result, so that if-converted code can make use of this register pair.

The idea of predication dates back to at least 1952, when the IBM 604 plugboard-controlled computer included a suppression bit in each instruction format and programmers could provide if-converted segments of code [2]. Predication has been an important part of several instruction sets, including Electrologica X8 (1965), IBM ACS (1967), ARM (1986), Cydra-5 (1988), and Multiflow (1990) [2,24]. Other instruction sets without extra bits to spare in instruction formats have added a conditional move instruction, which provides for "partial predication".

3.3. Unbundled branches

Conditional branches are composed of three separate actions: (1) making a decision to branch or not; (2) providing the target address; and, (3) actual change of the PC. By separating these actions, multiple comparisons can be made in parallel, earlier in the instruction stream. Moreover, multiple targets can be specified, and instructions can be prefetched from those paths. Thus, the change of the PC can be delayed until an explicit branch instruction or set of branch instructions, having the effect of a multiway, prioritized branch.

The IPF architecture uses the predicate registers to record the results of comparisons and includes eight branch registers for use in prefetching. Branch instructions in IPF are made conditional by use of a predicate and can specify a branch register (action 3) or relative address (actions 2+3).

The decomposition of branches into separate actions is an idea that has been independently rediscovered several times, but the decomposition into actions 1+2 as a branch on condition instruction and action 3 as a separate exit instruction that chose among the currently active targets was part of the IBM ACS-1 instruction set in the mid-1960s [31].

3.4. Compiler control of the memory hierarchy

EPIC architectures should be able to provide hints to the hardware about the probable latency of load operation (*i.e.*, where in the memory hierarchy a data value will be found) and the probable locality of a loaded or stored data item (*i.e.*, where in the memory hierarchy to place a data value). These are hints rather than exact operation timings, so register interlocks or scoreboarding techniques are still used.

IPF provides hints as given in Table 2 and also provides prefetching stride information by use of base-update addressing mode. Because of low temporal locality of vector operands, data cache bypass was a feature of some vector processors. The Intel i860 was perhaps the first processor to offer two types of scalar load instructions, one of which would bypass cache [18]. In 1994, the HP 7200 included temporal locality hints as part of the normal load/store instructions [20]. Several instruction set architectures since that time have included locality hints, typically as part of software prefetch instructions (*e.g.*, Alpha, MIPS, SPARC v.9).

hint	**Store**	**Load**	**Fetch**
Temporal locality / L1	Yes	Yes	Yes
No temporal locality / L1		Yes	Yes
No temporal locality / L2			Yes
No temporal locality / all levels	Yes	Yes	Yes

Table 2. Cache Hints in IPF

3.5. Control speculation

To start loads (or other potentially-long-running instructions) early, they must often be moved up beyond a branch. The problem with this approach occurs when the load (or other instruction) generates an exception. If the branch is taken, the load (or other instruction) would not have been executed in the original program and thus the exception should not be seen. To allow this type of code scheduling, an EPIC architecture should provide a speculative form of load (or other long-running instruction) and tagged operands. When the speculative instruction causes an exception, the exception is deferred by tagging the result with the required information. The exception is handled only when a nonspeculative instruction reads the tagged operand (in fact, multiple instructions may use the tagged operand in the meantime and merely pass the tag on). Thus, if the branch over which the instruction is moved is not taken, no exception occurs, thereby following the semantics of the original program.

IPF provides speculative load and speculation check instructions. Integer speculative loads set a NaT (Not a Thing) bit associated with the integer destination register when an exception is deferred. Floating-point speculative loads place a NaTVal (Not a Thing Value) code in the floating-point destination register when an exception is deferred. These bits and encoded values propagate through other instructions until a speculation check instruction is executed. At that point a NaT bit or NaTVal special value will raise the exception.

The need to start loads early can be seen back in Konrad Zuse's Z4 computer constructed in Germany during the Second World War. The instruction stream was read two instructions in advance; and, if a load was encountered, it was started early [2]. The IBM Stretch (1961) used a separate index processor to pre-execute index-related instructions in the instruction stream and start loads early [3]. Moving instructions across branches was a vital aspect of Fisher's trace scheduling [24], and Ebcioglu discussed conditional execution of instructions based on branches in 1987 [9]. Multiflow introduced special non-faulting loads and used IEEE floating-point NaN propagation rules for supporting control speculation [6]. Deferring exceptions from speculative loads appears to have been first presented by Smith, Lam, and Horowitz in 1990 [30].

3.6. Data speculation

To be able to rearrange load and store instructions, the compiler must know the memory addresses to which the instructions refer. Because of aliasing, compilers are not always able to do this at compile time. In the absence of exact alias analysis, most compilers must settle for safe but slower (*i.e.*, unreordered) code. EPIC architectures provide speculative loads that can be used when an alias situation is unlikely but yet still possible. A speculative load is moved earlier in the schedule to start the load as early as possible; and, at the place where the loaded value is needed, a data-verifying load is used instead. If no aliasing has occurred, then the value retrieved by the speculative load is used by the data-verifying load instruction. Otherwise, the data-verifying load reexecutes the load to obtain the new value.

IPF provides advanced load and advanced load check instructions that use an Advanced Load Address Table (ALAT) to detect stores that invalidate advanced load values.

The IBM Stretch (1961) started loads early, as mentioned above. The lookahead unit checked the memory address of a store instruction against subsequent loads and on a match cancelled the load and forwarded the store value to the buffer reserved for the loaded value (only one outstanding store was allowed at a time) [3]. The CDC 6600 (1964) memory stunt box performed similar actions [34].

4. Alternate Translation Times

To this point, we have assumed a standard compilation model, which includes steps such as compilation, linking, loading, and execution. We have assumed that either the compiler or the hardware does the three major tasks of managing ILP. However, alternatives exist. For example, even within the compilation model, nontraditional points of translation and optimization have been proposed, such as reallocating registers and/or repositioning procedures at link time for better performance. Additionally, other nontraditional points are available during execution, such as software-based translation at page-fault time [7], hardware-based translation at icache-miss time [1,22], hardware-based capture and caching of parallel issue [11,23], or various dynamic optimizations during execution (*e.g.*, software

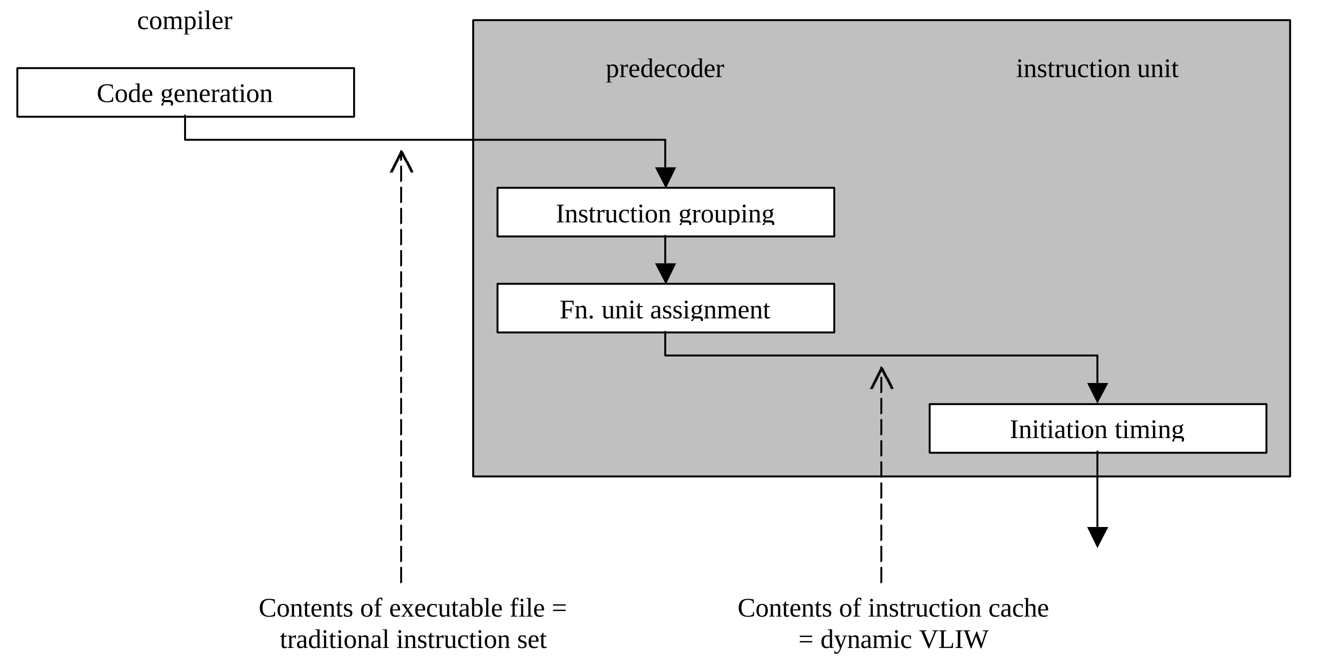

Figure 2. Nontraditional Assignment of ILP Management Tasks.

based [10] or hardware-based [5]). At any of these additional points, translation from a traditional instruction set into an EPIC or VLIW internal format is possible. Indeed, a typical compiler optimization step of if-conversion has been proposed as a run-time action using either software [12] or hardware [17]. Transmeta provides a run-time software approach that translates x86 instructions into an internal VLIW format, which they call "code-morphing" and which includes data and control speculation [15,16].

An early example of the run-time hardware approach is the National Semiconductor Swordfish processor (1990). Instructions from a traditional instruction set were examined at instruction cache miss by a hardware predecoder. The predecoder checked the instruction types and stored pairs of instructions with a grouping bit for parallel issue in the instruction cache [32]. Register scoreboarding was still performed at decode time, so this scheme looks like a traditional superscalar processor from the outside but is actually a dynamic-VLIW processor internally. Figure 2 illustrates this approach.

5. Conclusions

EPIC architectures are a new style of instruction set for computers. They are the skillful combination of several preexisting ideas in computer architecture along with a nontraditional assignment of the responsibilities in ILP processing between the compiler and the hardware. As such, EPIC architectures can claim to combine the best attributes of superscalar processors (compatibility across implementations) and VLIW processors (efficiency since less control logic). Through nontraditional translation, current traditional instruction sets can be used but the combined hardware and software system can exploit the efficiency of VLIW and EPIC implementations.

References

[1] S. Banerjia, *et al.*, "MPS: Miss-Path Scheduling for Multiple Issue Processors," *IEEE Transactions on Computers*, December 1998, pp. 1382-1397.

[2] G. Blaauw and F. Brooks, Jr., *Computer Architecture: Concepts and Evolution*. Reading, MA: Addison-Wesley, 1997.

[3] W. Buchholz (ed.), *Planning a Computer System*. New York: McGraw-Hill, 1962.

[4] A. Charlesworth, "An Approach to Scientific Array Processing: The Architectural Design of the AP-120B/FPS-164 Family," *IEEE Computer*, September 1981, pp. 18-27.

[5] Y. Chou, *et al.*, "PipeRench Implementation of the Instruction Path Coprocessor," in *Proceedings Micro-33*, December 2000, pp. 147-158.

[6] R. Colwell, *et al.*, "A VLIW Architecture for a Trace Scheduling Compiler," *IEEE Transactions on Computers*, August 1988, pp. 967-979.

[7] T. Conte and S. Sathaye, "Dynamic Rescheduling: A Technique for Object Code Compatibility in VLIW Architectures," in *Proceedings of Micro-28*, Ann Arbor, November 1995, pp. 208-217.

[8] J. Crawford, "Introducing the Itanium Processors," *IEEE Micro*, September-October 2000, pp. 9-11.

[9] K. Ebcioglu, "A Compilation Technique for Software Pipelining of Loops with Conditional Jumps," in *Proceedings of Micro-20*, Colorado Springs, December 1987, pp. 69-79.

[10] K. Ebcioglu and E. Altman, "DAISY: Dynamic Compilation for 100% Architectural Compatibility," in *Proceedings of ISCA-24*, Denver, June 1997, pp. 26-37.

[11] M. Franklin and M. Smotherman, "A Fill-Unit Approach to Multiple Instruction Issue," in *Proceedings of Micro-27*, San Jose, December 1994, pp. 162-171.

[12] K. Hazelwood and T. Conte, "A Lightweight Algorithm for Dynamic If-Conversion During Dynamic Optimization," in *Proceedings PACT*, 2000, pp. 71-80.

[13] L. Higbie, "Overlapped Operation with Microprogramming," *IEEE Transactions on Computers*, March 1978, pp. 270-275.

[14] A. Holler, "Compiler Optimizations for the PA-8000," in *Proceedings of Compcon 97*, San Jose, February 1997, pp. 87-94.

[15] E. Kelly, R. Cmelik, and M. Wing, "Memory Controller for a Microprocessor for Detecting a Failure of Speculation on the Physical Nature of a Component Being Addressed," US Patent 5,832,205.

[16] A. Klaiber, "The Technology Behind Crusoe Processors," Transmeta Corporation, January 2000.

[17] A. Klauser, *et al.*, "Dynamic Hammock Predication for Non-predicated Instruction Set Architectures," in *Proceedings PACT*, 1998, pp. 278-285.

[18] L. Kohn and N. Margulis, "Introducing the Intel i860 64-bit Microprocessor," *IEEE Micro*, August 1989, pp. 15-30.

[19] J. Kuehn and B. Smith, "The Horizon Supercomputing System: Architecture and Software," in *Proceedings of Supercomputing 88*, Orlando, November 1988, pp. 28-34.

[20] G. Kurpanek, *et al*., "PA 7200: A PA-RISC Processor with Integrated High-Performance MP Bus Interface," in *Proceedings of Compcon 94*, San Francisco, February 1994, pp. 375-382.

[21] S. McGeady, "The i960CA Superscalar Implementation of the 80960 Architecture," in *Proceedings of Compcon 90*, San Francisco, January 1990, pp. 232-239.

[22] K. Minagawa, M. Saito, and T. Aikawa, "Pre-decoding Mechanism for Superscalar Architecture," in *Proceedings of IEEE Pacific Rim Conference on Communications*, Victoria B.C., May 1991, pp. 21-24.

[23] R. Nair and M. Hopkins, "Exploiting Instruction Level Parallelism in Processors by Caching Scheduled Groups," in *Proceedings ISCA-24*, June 1997, pp. 13-25.

[24] B. Rau and J. Fisher, "Instruction-Level Parallel Processing: History, Overview, and Perspective," *Journal of Supercomputing*, July 1993, pp. 9-50.

[25] B. Rau, *et al.*, "The Cydra 5 Departmental Supercomputer," *IEEE Computer*, January 1989, pp. 12-35.

[26] K. Rudd, "VLIW Processors: Efficiently Exploiting Instruction Level Parallelism," Ph.D. dissertation, Computer Science Dept., Stanford University, 1999.

[27] M. Schlansker, *et al.*, "Achieving High Levels of Instruction-Level Parallelism with Reduced Hardware Complexity," HP Labs Tech. Rept. HPL-96-120, November 1994.

[28] M. Schlansker and B. Rau, "EPIC: Explicitly Parallel Instruction Computing," *IEEE Computer*, February 2000, pp. 37-45. (See also "EPIC: An Architecture for Instruction-Level Parallel Processors," HP Labs Tech. Rept. HPL-1999-111, February, 2000.)

[29] H. Sharangpani and K. Arora, "Itanium Processor Microarchitecture," *IEEE Micro*, September-October 2000, pp. 24-43.

[30] M. Smith, M. Lam, and M. Horowitz, "Boosting Beyond Static Scheduling in a Superscalar Processor," in *Proceedings of ISCA-17*, Seattle, May 1990, pp. 344-354.

[31] M. Smotherman, "IBM Advanced Computing Systems – A Secret 1960's Supercomputer Project," available online http://www.cs.clemson.edu/~mark/acs.html

[32] M. Smotherman, "National Semiconductor Swordfish," available online http://www.cs.clemson.edu/~mark/swordfish.html

[33] Special issue, "IBM RISC System/6000 Processor," *IBM Journal of Research and Development*, January 1990.

[34] J. Thornton, *Design of a Computer – The Control Data 6600*. Glenview, IL: Scott, Foresman, and Co., 1970.

[35] G. Tjaden and M. Flynn, "Detection and Parallel Execution of Independent Instructions," *IEEE Transactions on Computers*, October 1970, pp. 889-895.

[36] R. Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units," *IBM Journal of Research and Development*, January 1967, pp. 25-33.

[37] S. Vassiliadis, R. Blaner, and R. Eickemeyer, "SCISM: A Scalable Compound Instruction Set Machine," *Journal of Research and Development*, January 1994, pp. 59-78.