Very Large Instruction Word Architectures (VLIW Processors and Trace Scheduling)

Binu Mathew
{mbinu} @ {cs.utah.edu}

1 What is a VLIW Processor?

Recent high performance processors have depended on Instruction Level Parallelism (ILP) to achieve high execution speed. ILP processors achieve their high performance by causing multiple operations to execute in parallel, using a combination of compiler and hardware techniques. Very Long Instruction Word (VLIW) is one particular style of processor design that tries to achieve high levels of instruction level parallelism by executing long instruction words composed of multiple operations. The long instruction word called a MultiOp consists of multiple arithmetic, logic and control operations each of which would probably be an individual operation on a simple RISC processor. The VLIW processor concurrently executes the set of operations within a MultiOp thereby achieving instruction level parallelism. The remainder of this article discusses the technology, history, uses and the future of such processors.

2 Different Flavors of Parallelism

Improvements in processor performance come from two main sources: faster semiconductor technology and parallel processing. Parallel processing on multiprocessors, multicomputers and processor clusters has traditionally involved a high degree of programming effort in mapping an algorithm to a form that can better exploit multiple processors and threads of execution. Such reorganization has often been productively applied, especially for scientific programs. The general-purpose microprocessor industry on the other hand has pursued methods of automatically speeding up existing programs without major restructuring effort. This lead to the development of Instruction Level Parallel (ILP) processors that try to speed up program execution by overlapping the execution of multiple instructions from an otherwise sequential program.

A simple processor that fetches and executes one instruction at a time is called a simple scalar processor. A processor with multiple function units has the potential to execute several operations in parallel. If the decision about which operations to execute in an overlapped manner is made at run time by the hardware, it is called a super scalar processor. To a simple scalar processor, a binary program represents a plan of execution. The processor acts as an interpreter that executes the instructions in the program one at a time. From the point of view of a modern super scalar processor, an input program is more like a representation of an algorithm for which several different

plans of execution are possible. Each plan of execution specifies when and on which function unit each instruction from the instruction stream is to be executed.

Different types of ILP processors vary in the manner in which the plan of execution is derived, but it typically involves both the compiler and the hardware. In the current breed of high performance processors like the Intel Pentium and the MIPS R18000, the compiler tries to expose parallelism to the processor by means of several optimizations. The net result of these optimizations is to place as many independent operations as possible close to each other in the instruction stream. At run time, the processor examines several instructions at a time, analyses the dependences between instructions and keeps track of the availability of data and hardware resources for each instruction. It tries to schedule each instruction as soon as the data and function units it needs are available. The processor's decisions are complicated by the fact that memory accesses often have variable latencies that depend on whether a memory access hits in the cache or not. Since such processors decide which function unit should be allocated to which instruction as execution progresses, they are said to be dynamically scheduled. Often, as a further performance improvement, such processors allow later instructions that are independent to execute ahead of an earlier instruction which is waiting for data or resources. In that case the processor is said to be out of order.

Branches are common operations in general-purpose code. On encountering a branch, a processor must decide whether or not to take the branch. If the branch is to be taken, the processor must start fetching instructions from the branch target. To avoid delays due to branches, modern processors try to predict the outcome of branches and execute instructions from beyond the branch. If the processor predicted the branch incorrectly, it may need to undo the effects of any instructions it has already executed beyond the branch. If a super scalar processor uses resources that may otherwise go idle to execute operations the result of which may or may not be used, it is said to be speculative.

Out of order speculative execution comes at a significant hardware expense. The complexity and non-scalability of the hardware structures used to implement these features could significantly hinder the performance of future processors. An alternative solution to this problem is to simplify processor hardware and transfer some of the complexity of extracting ILP to the compiler and run time system—the solution explored by VLIW processors.

Joseph Fisher, who coined the acronym VLIW, characterized such machines as architectures which issue one long instruction per cycle, where each long instruction called a MultiOp consists of many tightly coupled independent operations each of which execute in a small and statically predictable number of cycles. In such a system, the task of grouping independent operations into a MultiOp is done by a compiler or binary translator. The processor freed from the cumbersome task of dependence analysis has to merely execute in parallel the operations contained within a MultiOp. This leads to simpler and faster processor implementations. In later sections, we will see how VLIW processors try to deal with the problems of branch and memory latencies and implement their own variant of speculative execution. But, first, we present a brief history of VLIW processors.

3 A Brief History of VLIW Processors

For various reasons which were appropriate at that time, early computers were designed to have extremely complicated instructions. These instructions made designing the control circuits for such computers difficult. A solution to this problem was microprogramming, a technique proposed by Maurice Wilkes in 1951. In a micro programmed CPU, each program instruction is considered a macroinstruction to be executed by a simpler processor inside the CPU. Corresponding to each macroinstruction, there will be a sequence of microinstructions stored in a microcode ROM in the CPU.

Horizontal microprogramming is a particular style of microprogramming where bits in a wide microinstruction are directly used as control signals within the processor. In contrast, vertical microprogramming uses a shorter microinstruction or series of microinstructions in combination with some decoding logic to generate control signals. Microprogramming became a popular technique for implementing the control unit of processors after IBM adopted it for the System/360 series of computers.

Even before the days of the first VLIW machines, there were several processors and custom computing devices that used a single wide instruction word to control several function units working in parallel. However these machines were typically hand-coded and the code for such machines could not be generalized to other architectures. The basic problem was that compilers at that time looked only within basic blocks to extract ILP. Basic blocks are often short and contain many dependences and therefore the amount of ILP that can be obtained inside a basic block is quite limited.

Joseph Fisher, a pioneer of VLIW, while working on PUMA, a CDC-6600 emulator was frustrated by the difficulty of writing and maintaining 64 bit horizontal microcode for that processor. He started investigating a technique for global microcode compaction—a method to generate long horizontal microcode instructions from short sequential ones. Fisher soon realized that the technique he developed in 1979, called trace scheduling, could be used in a compiler to generate code for VLIW like architectures from a sequential source since the style of parallelism available in VLIW is very similar to that of horizontal microcode. His discovery lead to the design of the ELI-512 processor and the Bulldog trace-scheduling compiler.

Two companies were founded in 1984 to build VLIW based mini supercomputers. One was Multiflow, started by Fisher and colleagues from Yale University. The other was Cydrome founded by VLIW pioneer Bob Rau and his colleagues. In 1987, Cydrome delivered its first machine, the 256 bit Cydra 5, which included hardware support for software pipelining. In the same year, Multiflow delivered the Trace/200 machine, which was followed by the Trace/300 in 1988 and Trace/500 in 1990. The 200 and 300 series used a 256 bit instruction for 7 wide issue, 512 bits for 14 wide issue and 1024 bits for 28 wide issue. The 500 series only supported 14 and 28 wide issue. Unfortunately, the early VLIW machines were commercial failures. Cydrome closed in 1988 and Multiflow in 1990.

Since then, VLIW processors have seen a revival and some degree of commercial success. Some of the notable VLIW processors of recent years are the IA-64 or Itanium from Intel, the Crusoe processor from Transmeta, the Trimedia media-processor from Philips and the TMS320C62x series of DSPs from Texas Instruments. Some important research machines designed during this time include the Playdoh from HP labs, Tinker from North Carolina State University, and the Imagine stream and image processor currently developed at Stanford University.

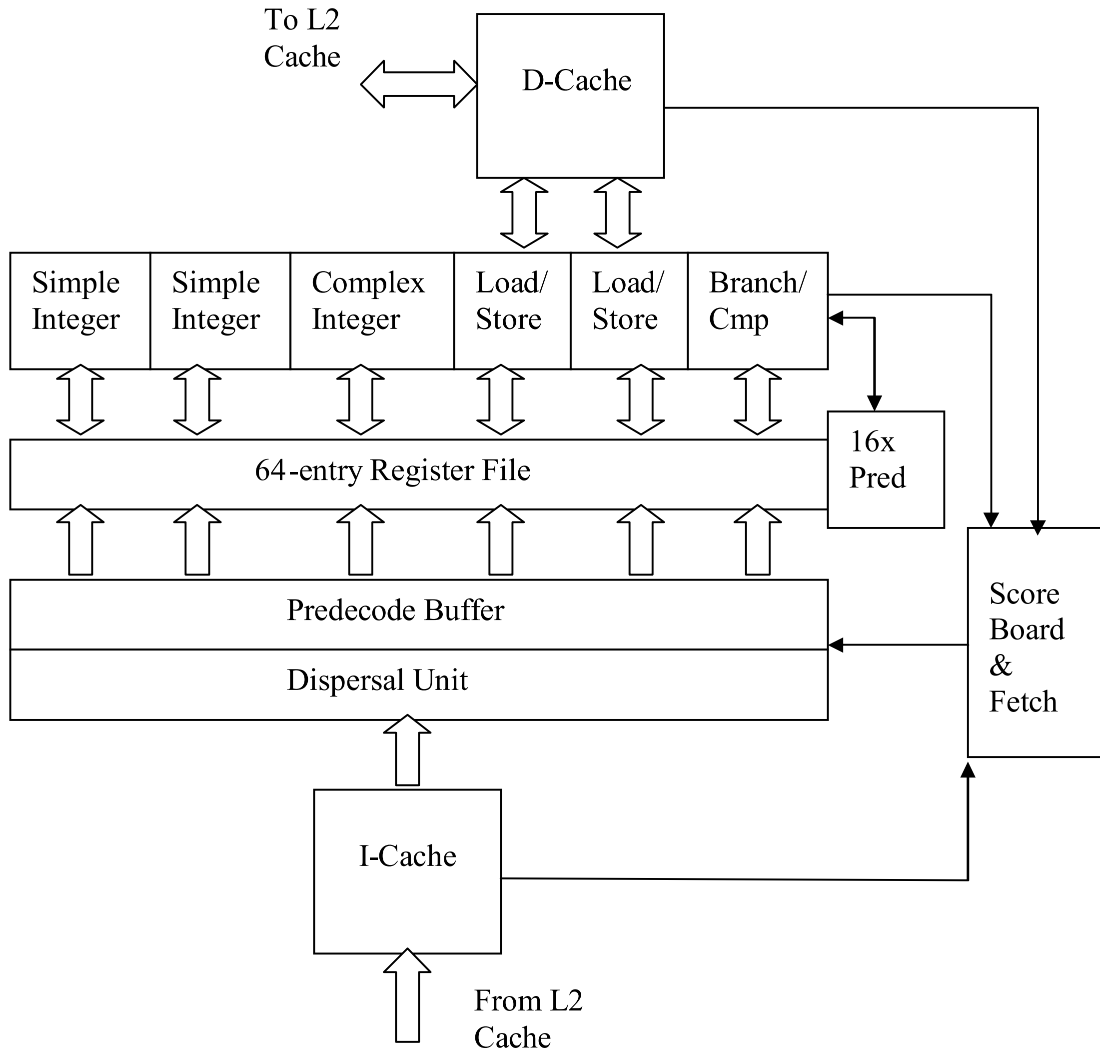

Figure 1: **Defoe Architecture**

4 Defoe: An Example VLIW Architecture

We now describe Defoe, an example processor used in this section to give the reader a feel for VLIW architecture and programming. Though it does not exist in reality, its features are derived from those of several existing VLIW processors. Later sections that describe the IA-64 and the Crusoe, will contrast those architectures with Defoe. Figure 1 shows the architecture of the Defoe processor.

4.1 Function units

Defoe is a 64-bit architecture with the following function units.

- Two load/store units.

Stop bit (1 bit)	Predicate (4 bits)	Opcode (9 bits)	Rdest (6)	Rsrc1 (6)	Rscr2 (6)

Figure 2: Instruction Encoding

- Two simple ALUs that perform add, subtract, shift and logical operations on 64-bit numbers and packed 32, 16 and 8-bit numbers. In addition, these units also support multiplication of packed 16 and 8-bit numbers.
- One complex ALU that can perform multiply and divide on 64-bit integers and packed 32, 16 and 8-bit integers.
- One branch unit that performs branch, call and comparison operations.

There is no support for floating point. Figure 1 shows a simplified diagram of the Defoe architecture.

4.2 Registers and Predication

Defoe has a set of 64 programmer visible general purpose registers which are 64 bits wide. As in the MIPS architecture, register R0 always contains zero. Predicate registers are special 1-bit registers that specify a true or false value. There are 16 programmer visible predicate registers in the Defoe named PR0 to PR15. All operations in Defoe are predicated, i.e., each instruction contains a predicate register field (PR) that contains a predicate register number. At run-time, if control reaches the instruction, the instruction is always executed. However, at execution time, if the predicate is false, the results are not written to the register file and side effects such as TLB miss exceptions are suppressed. In practice, for reasons of efficiency, this may be implemented by computing a result, but writing back the old value of the target register. Predicate register 0 always contains the value 1 and cannot be altered. Specifying PR0 as the predicate performs unconditional operations. Comparison operations use a predicate register as their target register.

4.3 Instruction Encoding

Defoe is a 64-bit compressed VLIW architecture. In an uncompressed VLIW system, MultiOps have a fixed length. When suitable operations are not available to fill the issue slots within a MultiOp, NOPs are inserted into those slots. A compressed VLIW architecture uses variable length MultiOps to get rid of those NOPs and achieve high code density. In the Defoe, individual operations are encoded as 32-bit words. A special stop bit in the 32-bit word indicates the end of a MultiOp. Common arithmetic operations have an immediate mode, where a sign or zero extended 8-bit constant may be used as an operand. For larger constants of 16, 32 or 64 bits, a special NOP code may be written into opcode field of the next operation and the low order bits may be used to store the constant. In that case, the predecoder concatenates the bits from 2 or more different words to assemble a constant. Figure 2 depicts the instruction format.

4.4 Instruction Dispersal and Issue

A traditional VLIW with fixed width MultiOps has no need to disperse operations. However, when using a compressed format like that of the Defoe, there is a need to expand the operations, and insert NOPs for function units to which no operation is to be issued. To make the dispersal task easy we make the following assumptions:

- A few bits in the opcode specify the type of function unit (i.e. load/store, simple arithmetic, complex arithmetic or branch) the operation needs.
- The compiler ensures that the instructions that comprise a MultiOp are sorted in the same order as the function units in the processor. This reduces the circuit complexity of the instruction dispersal stage. For example, if a MultiOp consists of a load, 32-bit divide and a branch, then the ordering (load, multiply, branch) is legal, but the ordering (load, branch, multiply) is not legal.
- The compiler ensures that all the operations in the same MultiOp are independent.
- The compiler ensures that the function units are not over subscribed. For example, it is legal to have two loads in a MultiOp, but it is not legal to have three loads.
- It is illegal to not have a stop bit in a sequence of more than 6 instructions.
- Basic blocks are aligned at 32-byte boundaries.

Apart from reducing wastage of memory, another reason to prefer a compressed format VLIW over an uncompressed one is that the former provides better I-Cache utilization. To improve performance, we use a predecode buffer that can hold up to 8 uncompressed MultiOps. The dispersal network can use a wide interface (say 512 bits) to the I-cache to uncompress up to 2 MultiOps every cycle and save them in the predecode buffer. Small loops of up to 8 MultiOps (maximum 48 operations) will experience repeated hits in the predecode buffer. It may also help lower the power consumption of a low-power VLIW processor. Defoe supports in-order issue and out of order completion. Further, all the operations in a MultiOp are issued simultaneously. If even one operation cannot be issued, issue of the whole MultiOp stalls.

4.5 Branch Prediction

Following the VLIW philosophy of enabling the software to communicate its needs to the hardware, branch instructions in Defoe can advise the processor about their expected behavior. A two bit hint associated with every branch may be interpreted as shown in Table 1.

Implementations of the Defoe architecture may provide branch prediction hardware, but a branch predictor is not required in a minimal implementation. If branch prediction hardware is provided, static branches need not be entered in the branch history table, thereby freeing up resources for dynamically predicted branches.

Opcode Modifier	Meaning
Stk	Static prediction. Branch is usually taken.
Sntk	Static prediction. Branch is usually not taken.
Dtk	Dynamic prediction. Assume branch is taken if no history is available.
Dntk	Dynamic prediction. Assume branch is not taken if no history is available.

Table 1: Branch Prediction Hints

4.6 Score board

To accommodate branch prediction and the variable latency of memory accesses because of cache hits and misses, some amount of score boarding is required. Though we will not describe the details of the scoreboard here, it should be emphasized that the scoreboard and control logic for a VLIW processor like the Defoe is much simpler than that of a modern super scalar processor because of the lack of out of order execution and speculation.

4.7 Assembly Language Syntax

The examples that follow use the following syntax for assembly language instructions.

(predicate_reg) opcode.modifier Rdest = Rsource1, Rsource2

If the predicate register is omitted, PR0 will be assumed. In addition, a semicolon following an instruction indicates that the stop bit is set for that operation, i.e., that operation is the last one in its MultiOp. The prefix "!" for a predicate implies that the opcode actually depends on the logical not of the value of the predicate register.

4.8 Example 1

This example demonstrates the execution model of the Defoe by computing the following set of expressions.

```
a = x + y - z
b = x + y - 2 * z
c = x + y - 3 * z
```

Register assignments: r1 = x, r2 = y, r3 = z, r32 = a, r33 = b, r34 = c

```
Line # Code Comments
1. add r4 = r1, r2 // r4 = x + y
2. shl r5 = r3, 1 // r5 = z << 1, i.e. z * 2
3. mul r6 = r3, 3 ; // r6 = z * 3. Stop bit.
4. sub r32 = r4, r3 // r5 = a = gets x + y - z
5. sub r33 = r4, r5 ; // r33 = b = x + y - 2 * z.
// Stop bit.
```

```
6. sub r34 = r4, r6 ; // r34 = c = x + y - 3 * z.
// Stop bit.
```

The first three lines are followed by a stop bit to indicate that those three operations constitute a MultiOp and that they should be executed in parallel. Unlike a super scalar processor where independent operations are detected by the processor, the programmer/compiler has indicated to the processor by means of the stop bit that these 3 operations are independent. The multiply operation will typically have a higher latency than the other instructions. In that case we have two different ways of scheduling this code. Since Defoe already uses score boarding to deal with variable load latencies, it is only natural for the scoreboard to stall issue till the multiply operation is done. In a traditional VLIW processor, the compiler will insert additional NOPs after the first MultiOp. Lines 4-6 show how structural hazards are handled in a VLIW system. The compiler is aware that Defoe has only two simple integer ALUs. Even though instruction 6 is independent of instructions 4 and 5, because of the unavailability of a suitable function unit, instruction 6 is issued as a separate MultiOp, one cycle after its two predecessors. In a super scalar processor, this decision will be handled at run-time by the hardware.

4.9 Example 2

This example contrasts the execution of an algorithm on Defoe and a super scalar processor (Intel Pentium). The C language function absdiff computes the sum of absolute difference of two arrays A and B which contain 256 elements each.

```
int absdiff(int *A, int *B)
{
  int sum, diff, i;
  sum = 0;
  for(i = 0; i<256; i++)
    {
        diff = A[i] - B[i];
        if(A[i] >= B[i])
          sum = sum + diff;
        else
         sum = sum - diff;       }
  return sum;
}
```

A hand assembled version of absdiff in Defoe assembly language is shown below. For clarity, it has been left unoptimized. An optimizing compiler will unroll this loop and software pipeline it.

Register assignment: On entry, r1 = a, r2 = b. On exit, sum is in r4.

```
Line # Code Comment
1. add r3 = r1, 2040 // r3 = End of array A
2. add r4 = r0, r0 ; // sum = r4 = 0
.L1:
```

```
3. ld r5 = [r1]      // load A[i]
4. ld r6 = [r2]      // load B[i]
5. add r1 = r1, 8    // Increment A address
6. add r2 = r2, 8    // Increment B address
7. cmp.neq pr1 = r1, r3 ; // pr1 = (i != 255)
8. sub r7, r5, r6 // diff = A[i] - B[i]
9. cmp.gte pr2 = r5, r6 ; // pr2 = (A[i] >= B[i])
10. (pr2) add r4 = r4, r7 // if A[i] >= B[i]
// sum = sum + diff
11. (!pr2) sub r4 = r4, r7 // else sum = sum - diff
12. (pr1) br.sptk .L1 ;
```

The corresponding code for an Intel processor is shown below. This is a snippet of actual code generated by the GCC compiler.

Stack assignment: On entry, 12(%ebp) = B, 8(%ebp) = A. On exit, sum is in the eax register.

```
Line # Code Comment
1. movl 12(%ebp), %edi // edi = B
2. xorl %esi, %esi // esi sum = 0
3. xorl %ebx, %ebx // ebx = 0
.p2align 2
.L6:
4. movl 8(%ebp), %eax // eax = A
5. movl (%eax,%ebx,4), %edx // edx = A[i]
6. movl %edx, %ecx // ecx = A[i]
7. movl (%edi,%ebx,4), %eax // eax = B[i]
8. subl %eax, %ecx // ecx = diff = A[i] - B[i]
9. cmpl %eax, %edx // A[i] < B[i]
10. jl .L7 // goto .L7 is A[i] < B[i]
11. addl %ecx, %esi // sum = sum + diff
12. jmp .L5
.p2align 2
.L7:
13. subl %ecx, %esi // sum = sum - diff
.L5:
14. incl %ebx // i++
15. cmpl $255, %ebx // i <= 255 ?
16. jle .L6
17. popl %ebx
18. movl %esi, %eax
```

The level of parallelism available in the Defoe listing lines 3-7 (five issue) can be achieved on a super scalar processor only if the processor can successfully isolate the five independent operations fast enough to issue them all during the same cycle. Dependency checking in h/w is extremely complex and adds to the delay of super scalar processors. The x86 being a register deficient CISC

architecture also incurs additional penalties because of register renaming and CISC to internal RISC format translation.

It is worth noticing that the Defoe listing contains only one branch (on line 12) whereas the x86 listing contains 3 branches. On a VLIW processor, we can often use predicated instructions to eliminate branches. In both listings, line 9 corresponds to the comparison of A[i] and B[i]. The Pentium version does a conditional jump based on the result of the comparison. On the other hand, the VLIW uses the result of the comparison to set a predicate. The predicate is then used to selectively write back the result of either the add or the subtract operation and the result of the other operation is discarded. This technique of converting a control dependence into a data dependence is called 'if conversion'. The benefits go beyond the single cycle saved by not doing a jump as in the case of the super scalar processor. The jumps on line 10 and 12 in the second listing depend on the condition code which in turn depends on the data. Such data dependent branches are difficult to predict. Assuming that A[i] $<$ B[i] and A[i] $>=$ B[i] are equally likely, the super scalar processor is likely to experience a branch misprediction and the resulting branch penalty half of the time.

Going by the VLIW philosophy of conveying performance critical information from the compiler to the hardware, the final branch on line 12 uses the opcode modifier "sptk" to inform the processor that the branch is statically predicted to be taken. For that particular loop, a VLIW processor can therefore predict the loop accurately 255 times out of 256 loop iterations without any hardware branch predictor. Even when a hardware branch predictor is available, the instruction advises the processor not to waste a branch history table entry on that branch since its behavior is already known at compile time.

5 The Intel Itanium Processor

The Itanium-1 processor is Intel's first implementation of the IA-64 ISA. IA-64 is an ISA for the Explicitly Parallel Instruction Computing (EPIC) style of VLIW developed jointly by Intel and HP. It is a 64-bit, 6 issue VLIW processor with four integer units, four multimedia units, two load/store units, two extended precision floating point units and two single precision floating point units. This processor running at 800 MHz on a 0.18 micron process has a 10 stage pipeline.

Unlike the Defoe, the IA-64 architecture uses a fixed-width bundled instruction format. Each MultiOp consists of one or more 128 bit bundles. Each 128 bit bundle consists of three operations and a template. Unlike the Defoe where the opcode in each operation specifies a type field, the template encodes commonly used combinations of operation types. Since the template field is only five bits wide, bundles do not support all possible combinations of instruction types. Much like the Defoe's stop bit, in the IA-64, some template codes specify where in the bundle a MultiOp ends. In IA-64 terminology, MultiOps are called instruction groups. Like Defoe, the IA-64 uses a decoupling buffer to improve its issue rate. Though the IA-64 registers are nominally 64 bits wide, there is a hidden 65th bit called NaT (Not a Thing). This is used to support speculation. There are 128 general purpose registers and another set of 128, 82-bit wide floating point registers. Like the Defoe, all operations on the IA-64 are predicated. However, the IA-64 has 64 predicate registers.

The IA-64 register mechanism is more complex than the Defoe's because it implements support for software pipelining using a method similar to the overlapped loop execution support pioneered by Bob Rau and implemented in the Cydra 5. On the IA-64, general purpose registers GPR0 to GPR31 are fixed. Registers 32-127 can be renamed under program control to support a register

stack or to do modulo scheduling for loops. When used to support software pipelining, this feature is called register rotation. Predicate registers 0 to 15 are fixed while predicate registers 16 to 63 can be made to rotate in unison with the general purpose registers. The floating point registers also support rotation.

Modulo scheduling is a software pipelining technique that can support overlapped execution of loop bodies while reducing tail code. In a pipelined function unit, each stage can hold a computation and successive items of data may be applied to the function unit before previous data is completely processed. To take advantage of pipelined operation, in a modulo scheduled loop, the loop body is unrolled and split into several stages. The compiler can schedule multiple iterations of a loop in a pipelined manner as long as data outputs of one stage flow into the inputs of the next stage in the software pipeline. Traditionally, this required unrolling the loop and renaming the registers used in successive iterations. IA-64 reduces the overhead of such a loop and avoids the need for register renaming by rotating registers forward, i.e., the rotating register base is incremented in the direction of increasing register index. After rotating by n registers, the value that was in register X+n can be accessed from register X. When used in conjunction with predication, this allows a natural expression of software pipelines similar to their hardware counterparts.

The IA-64 supports software directed control and data speculation. To do control speculation, the compiler moves a load before its controlling branch. The load is then flagged as a speculative load. The processor does not signal exceptions on a speculative load. If the controlling branch is taken, the compiler uses a special opcode named check.s to determine if an exception occurred. If an exception occurred, the check operation transfers control to exception handling code.

To support data speculation, the processor supports a special kind of load called an advance load. If the compiler cannot disambiguate between the addresses of a store and a later load, it can issue an advance load ahead of the store. The processor uses a special hardware structure called the ALAT to keep track of whether a later store wrote to the same location as the advance load. In the original location where the load might naturally have been placed, the compiler inserts a special check operation to see if a store invalidated the result of the advance load. If the advance load was invalidated, the check operation transfers control to recovery code.

As in the case of Defoe, the IA-64 too supports both static and dynamic hints for branches. It also makes use of hardware branch prediction. There are also hints in load and store instructions that inform the processor about the cache behavior of a particular memory operation.

The IA-64 also includes SIMD instructions suitable for media processing. Special multimedia instructions similar to the MMX and SSE extensions for 80x86 processors treat the contents of a general purpose register as two 32-bit, four 16-bit or eight 8-bit operands and operate on them in parallel.

To improve performance, the IA-64 architecture includes several features that are not found in a traditional VLIW architecture. The Intel Itanium processor is probably the most complex VLIW ever designed. It is a matter of debate whether some of the control complexity of the IA-64 is justifiable in a VLIW architecture and whether the enhancements deliver commensurate performance improvements. Next, we will look at a simpler VLIW processor that has been designed with a totally different goal — that of reducing power consumption.

6 The Transmeta Crusoe Processor

The Crusoe processor from Transmeta corporation represents a very interesting point in the development of VLIW processors. Traditionally, VLIW processors were designed with the goal of maximizing ILP and performance. The designers of the Crusoe on the other hand needed to build a processor with moderate performance compared to the CPU of a desktop computer, but with the additional restriction that the Crusoe should consume very little power since it was intended for mobile applications. Another design goal was that it should be able to efficiently emulate the ISA of other processors, particularly the 80x86 and the Java virtual machine.

The designers left out features like out of order issue and dynamic scheduling that require significant power consumption. They set out to replace such complex mechanisms of gaining ILP with simpler and more power efficient alternatives. The end result was a simple VLIW architecture. Long instructions on the Crusoe are either 64 or 128 bits. A 128-bit instruction word called a molecule in Transmeta parlance encodes 4 operations called atoms. The molecule format directly determines how operations get routed to function units. The Crusoe has two integer units, a floating point unit, a load/store unit and a branch unit. Like the Defoe, the Crusoe has 64 general purpose registers and supports strictly in order issue. Unlike the Defoe which uses predication, the Crusoe uses condition flags which are identical to those of the x86 for ease of emulation.

Binary x86 programs, firmware and operating systems are emulated with the help of a run time binary translator called code morphing software. This makes the classical VLIW software compatibility problem a non-issue. Only the native code morphing software needs to be changed when the Crusoe architecture or ISA changes. As a power and performance optimization, the hardware and software together maintain a cache of translated code. The translations are instrumented to collect execution frequencies and branch history and this information is fed back to the code morphing software to guide its optimizations.

To correctly model the precise exception semantics of the x86 processor, the part of the register file that holds x86 register state is duplicated. The duplicate is called a shadow copy. Normal operations only affect the original registers. At the end of a translated section of code, a special commit operation is used to copy the working register values to the shadow registers. If an exception happens while executing a translated unit, the run time software uses the shadow copy to recreate the precise exception state. Store operations are implemented in a similar manner using a store buffer. As in the case of IA-64, the Crusoe provides alias detection hardware and data speculation primitives.

7 Scheduling Algorithms for VLIW

The difficulty of programming VLIW processors by hand should be evident even from the simple Defoe programming examples. One reason programming VLIWs is more difficult than writing code for a super scalar processor is that the program for a super scalar processor is inherently sequential and it is left to the hardware to extract parallelism from the sequential program. On the other hand, when generating code for a VLIW processor, the assembly language programmer or the compiler is faced with the task of extracting parallelism from a sequential algorithm and scheduling independent operations concurrently. For this reason, instruction scheduling algorithms are critical to the performance of a VLIW processor. We next describe three important scheduling algorithms

starting with the classic trace scheduling algorithm.

7.1 Trace Scheduling

Many compilers for first-generation ILP processors used a three phase method to generate code. The phases were:

- Generate a sequential program. Analyze each basic block in the sequential program for independent operations.
- Schedule independent operations within the same block in parallel if sufficient hardware resources are available.
- Move operations between blocks when possible.

This three phase approach fails to exploit much of the ILP available in the program for two reasons.

- Often times, operations in a basic block are dependent on each other. Therefore sufficient ILP may not be available within a basic block.
- Arbitrary choices made while scheduling basic blocks make it difficult to move operations between blocks.

Trace scheduling is a profile driven method developed by Joseph Fisher to circumvent this problem. In trace scheduling, a set of commonly executed sequence of blocks is gathered together into a trace and the whole trace is scheduled together.

The trace scheduling algorithm works as follows.

1. Generate a possibly unoptimized version of the program, run it on sample input and collect statistics. Estimate the probability of each conditional branch.

2. From the basic block level data precedence graph of the program (also commonly called DAG for Directed Acylic Graph), select a loop free linear sequence of basic blocks which have a high probability of execution. Such a sequence is called a trace. The compiler may use other optimizations like loop unrolling or procedure inlining to generate DAGs from which suitable traces can be selected.

3. Consider the trace as if it were a basic block. Build a DAG for it considering branches like all other operations. If an operation controlled by a conditional jump could over write a value that is live on the off-trace edge, add an edge that makes the operation dependent on the branch so that the operation cannot be moved ahead of the branch. Also, add edges to preserve the relative order of conditional branches.

4. Schedule the resulting DAG as if it were a basic block doing register allocation and function unit selection as each operation is scheduled.

5. Generate compensation code for mistakes made by considering the trace as a basic block. In particular:

a. If an operation that used to precede a conditional branch in the sequential code is moved after that branch, then add a copy of that operation preceding the off-trace target of the conditional jump.

b. If an operation that succeeded a point of entry into the trace from outside the trace is moved ahead of that point of entry, place a copy of that operation outside the trace, on the path that leads to that point of entry.

c. Ensure that rejoins that used to enter the trace enter the new trace only at a point after which no operation is found in the new trace that were not below the rejoin point in the old trace.

6. Link the new trace back into the old DAG.

7. After scheduling the very first trace, new operations would have been added to the original DAG. Pick a different frequent trace and schedule it. Repeat till the DAG has been covered using disjoint traces and no unscheduled operations remain.

7.2 Trace Scheduling - 2

Trace scheduling - 2 goes beyond the original trace scheduling in that it allows nonlinear code motion, i.e. it allows operations from both sides of a conditional branch to be moved above the branch. Trace scheduling usually misses code motions that are speculative or moves operations from one trace to another. Trace scheduling - 2 on the other hand uses an expected value function called speculative yield to consider the cost of speculative execution and decide whether or not to move operations from one block to another. Unlike trace scheduling which operates on a linear sequence of blocks, the newer algorithm works by picking clusters of operations where each cluster is a maximal set of operations that are connected without back edges in the flow graph of the program. The actual details of the algorithm are beyond the scope of this article.

7.3 Super Block Scheduling

Super block scheduling is a region scheduling algorithm developed in conjunction with the Impact compiler at the University of Illinois. Like trace scheduling, super block scheduling is based on the premise that extracting ILP from sequential programs requires code motion across multiple basic blocks. Unlike trace scheduling, super block scheduling is driven by static branch analysis, not profile data. A super block is a set of basic blocks in which control may enter only at the top, but may exit at more than one point. Super blocks are identified by first identifying traces and then eliminating side entries into a trace by a process called tail duplication. Tail duplication works by creating a separate off-trace copy of the basic blocks in between a side entrance and the trace exit and redirecting the edge corresponding to the side entry to the copy. Traces are identified using static branch analysis based on loop detection, heuristic hazard avoidance and heuristics for path selection. Loop detection identifies loops and marks loop back edges as taken and loop exits as not taken. Hazard avoidance uses a set of heuristics to detect situations like ambiguous stores and procedure calls that could a cause a compiler to use conservative optimization strategies and then predicts the branches so as to avoid having to optimize hazards. Path selection heuristics use the opcode of a branch, its operands and the contents of its successor blocks to predict its direction if no other method can be used to predict the outcome of the branch. These are based on common programming patterns like the fact that pointers are unlikely to be NULL, floating point comparisons are unlikely to be equal etc. Once branch information is available, traces are grown and super blocks created by tail duplication followed by scheduling of the super block. Studies have shown that static analysis based super block scheduling can achieve results that are comparable to profile based methods.

7.4 The Future of VLIW Processors

VLIW processors have enjoyed moderate commercial success in recent times as exemplified by the Philips Trimedia, TI TMS320C62x DSPs, Intel Itanium and to a lesser extend the Transmeta Crusoe. However, the role of VLIW processors has changed since the days of Cydrome and Multiflow. Even though early VLIW processors were developed to be scientific super computers, newer processors have been used mainly for stream, image and digital signal processing, multimedia codec hardware, low power mobile computers etc. VLIW compiler technology has made major advances during the last decade. However, most of the compiler techniques developed for VLIW are equally applicable to super scalar processors. Stream and media processing applications are typically very regular with predictable branch behavior and large amounts of ILP. They lend themselves easily to VLIW style execution. The ever increasing demand for multimedia applications will continue to fuel development of VLIW technology. However, in the short term, super scalar processors will probably dominate in the role of general purpose processors. Increasing wire delays in deep sub micron processes will ultimately force super scalar processors to use simpler and more scalable control structures and seek more help from software. It is reasonable to assume that in the long run, much of the VLIW technology and design philosophy will make its way into main stream processors.

7.5 References

1. Joseph A. Fisher, *Global code generation for instruction-level parallelism: Trace Scheduling-2*. Tech. Rep. HPL-93-43, Hewlett-Packard Laboratories, June 1993.

2. Joseph A. Fischer, *Very Long Instruction Word Architectures and the ELI-512*, Proceedings of the 10'th Symposium on Computer Architectures, pp. 140-150, IEEE, June, 1983.

3. Joseph A. Fisher, *Very Long Instruction Word Architectures and the ELI-512*. 25 Years ISCA: Retrospectives and Reprints 1998: 263-273

4. M. Schlansker, B. R. Rau, S. Mahlke, V. Kathail, R. Johnson, S. Anik, and S. G. Abraham, *Achieving high levels of instruction-level parallelism with reduced hardware complexity*. Technical report, Technical Report HPL-96-120, Hewlett Packard Laboratories, Feb. 1997.

5. M. Schlansker, B. R. Rau. *Epic: An Architecture for Instruction Level Parallel Processors*. Technical report, Technical Report HPL-1999-111, Hewlett Packard Laboratories, Feb. 2000.

6. Scott Rixner, William J. Dally, Ujval J. Kapasi, Brucek, Lopez-Lagunas, Abelardo, Peter R. Mattson, and John D. Owens. *A bandwidth-efficient architecture for media processing*. In Proc. 31st Annual International Symposium on Microarchitecture, Dallas, TX, November 1998.

7. Intel Corporation. *Itanium Processor Microarchitecture Reference for Software Optimization.* Intel Corporation, March 2000

8. Intel Corporation. *Intel IA-64 Architecture Software Developer's Manula, Volume 3: Instruction Set Reference.* Intel Corporation, January 2000

9. Intel Corporation. *IA-64 Application Developer's Architecture Guide.* Intel Corporation, May 1999

10. P.G. Lowney, S.M. Freudenberger, T.J. Karzes, W.D. Lichtenstein, R.P. Nix, J.S. O'Donnell, and J.C. Ruttenberg. *The Multiflow trace scheduling compiler*. Journal of Supercomputing, 7, 1993.

11. R. E. Hank, S. A. Mahlke, J. C. Gyllenhaal, R. Bringmann, and W. W. Hwu, *Superblock formation using static program analysis*, in Proc. 26th Ann. Int'l. Symp. on Microarchitecture, (Austin, TX), pp. 247–255, Dec. 1993.

12. S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann, *Effective compiler support for predicated execution using the hyperblock*, in Proceedings of the 25th International Symposium on Microarchitecture, pp. 45–54, December 1992.

13. James C. Dehnert, Peter Y. T. Hsu, Joseph P. Bratt, *Overlapped Loop Support in the Cydra* 5 in Proc. ASPLOS 89, pp. 26-38.

14. Alexander Klaiber, *The Technology Behind Crusoe Processors.* Transmeta Corp, 2000.