Dynamically Scheduled VLIW Processors

B. Ramakrishna Rau

Hewlett-Packard Laboratories, 1501 Page Mill Road, Palo Alto, CA 94304

Abstract

VLIW processors are viewed as an attractive way of achieving instruction-level parallelism because of their ability to issue multiple operations per cycle with relatively simple control logic. They are also perceived as being of limited interest as products because of the problem of object code compatibility across processors having different hardware latencies and varying levels of parallelism. In this paper, we introduce the concept of delayed split-issue and the dynamic scheduling hardware which, together, solve the compatibility problem for VLIW processors and, in fact, make it possible for such processors to use all of the interlocking and scoreboarding techniques that are known for superscalar processors.

Keywords: VLIW processors, multiple operation issue, scoreboarding, dynamic scheduling, out-of-order execution

1 Introduction

Traditionally, VLIW processors have been defined by the following set of attributes.

- The ability to specify multiple, independent operations in each instruction. (We shall refer to such an instruction as a **MultiOp** instruction. An instruction that has only one operation is a **UniOp** instruction.)
- Programs that assume specific non-unit latencies for the operations and which, in fact, are only correct when those assumptions are true.
- The requirement for static, compile-time operation scheduling taking into account operation latencies and resource availability.
- Consequently, the requirement that the hardware conform exactly to the assumptions built into the program with regards to the number of functional units and the operation latencies.
- The absence of any interlock hardware, despite the fact that multiple, pipelined operations are being issued every cycle.

The original attraction of this style of architecture is its ability to exploit large amounts of instruction-level parallelism (ILP) with relatively simple and inexpensive control hardware. Whereas a number of VLIW products have been built which are capable of issuing six or more operations per cycle [4, 5, 3], it has just not proven feasible to build superscalar products with this level of ILP [18, 2, 14, 8, 7, 6]. Furthermore, the complete exposure to the compiler of the available hardware resources and the exact operation latencies permits highly optimized schedules.

These very same properties have also led to the perception that VLIW processors are of limited interest as products. The rigid assumptions built into the program about the hardware are viewed as precluding object code compatibility between processors built at different times with different technologies and, therefore, having different latencies. Even in the context of a single processor, the need for the compiler to schedule to a latency, that is fixed at compile-time, is problematic with operations such as loads which can have high variability in their latency depending on whether a cache hit or miss occurs. Because of this latter problem, VLIW products have rarely adhered to the ideal of no interlock hardware, whatsoever. Interlocking and stalling of the processor is common when a load takes longer than expected.

Superscalar processors and other dynamically scheduled processors are better equipped, at least in principal, to deal with variable latencies. In fact, when the variability is low, such processors are quite successful in dynamically scheduling around the mis-estimated latencies. A broad range of instruction issuing techniques, developed over the past three decades, can be brought to bear on this task. Examples include the CDC 6600 scoreboard [18, 19], the register renaming scheme, known as Tomasulo's algorithm, incorporated in the IBM 360/91 [2, 20], the history file, reorder buffer and future file [15], the register update unit [16] and checkpoint-repair [9].

The conventional wisdom is that dynamic scheduling using such techniques is inapplicable to VLIW processors. The primary objective of this paper is to show that this view is wrong, that dynamic scheduling is just as viable

with VLIW processors as it is with more conventional ones. A first step towards understanding how to perform dynamic scheduling on VLIW processors is to recognize the distinction between traditional VLIW processors and the concept of a VLIW architecture.

A VLIW processor is defined by a specific set of resources (functional units, buses, etc.) and specific execution latencies with which the various operations are executed. If a program for a VLIW processor is compiled and scheduled assuming precisely those resources and latencies, it can be executed on that processor in an instruction-level parallel fashion without any special control logic. Conversely, a VLIW processor that has no special control logic can only correctly execute those programs that are compiled with the correct resource and latency assumptions. VLIW processors have traditionally been built with no special control logic and this has led to the conclusion that VLIW processors must necessarily be designed in this fashion.

A different view of VLIW is as an architecture, i.e., a contractual interface between the class of programs that are written for the architecture and the set of processor implementations of that architecture. The usual view is that this contract is concerned with the instruction format and the interpretation of the bits that constitute an instruction. But the contract goes further and it is these aspects of the contract that are of primary importance in this paper. First, via its MultiOp capability, a VLIW architecture specifies a set of operations that are guaranteed to be mutually independent (and which, therefore, may be issued simultaneously without any checks being made by the issue hardware).

Second, via assertions about the operation latencies, an architecture specifies how a program is to be interpreted if one is to correctly understand the dependences between operations. In the case of a sequential architecture, all latencies are assumed to be a single cycle. So, the input operands for an operation are determined by all the operations that were issued (and, therefore, completed) before the operation in question.

In the case of programs for VLIW architectures, with operations having non-unit latencies, the input operands for an operation are not determined by all the operations that were issued before the operation in question. What matters is the operations that are supposed to have *completed* before the issuance of the operation in question. Operations that were issued earlier, but which are not supposed to have completed as yet, do not impose a flow dependence upon the operation in question.

We introduce the following terminology to facilitate our discussion. A program has **unit assumed latencies (UAL)** if the semantics of the program are correctly understood by assuming that all operations in one instruction complete before the next instruction is issued. A program has **non-unit assumed latencies (NUAL)** if at least one operation has a non-unit assumed latency, L, which is greater than one, i.e., the semantics of the program are correctly understood if exactly the next L-1 instructions are understood to have been issued before this operation completes. An architecture is UAL (NUAL) if the class of programs that it is supposed to execute are UAL (NUAL). We shall use the terms NUAL program and **latency-cognizant program** interchangeably.

This paper addresses the following questions:

- How does one determine the dependence semantics of latency-cognizant programs?
- How does one do dynamic scheduling for a latency-cognizant program?
- How do the unique aspects of VLIW architectures, namely, NUAL and MultiOp, affect the mechanisms used to effect scoreboarding and out-of-order execution?

Due to space considerations, this paper will not discuss the issue of how one provides precise interrupts for VLIW architectures. The mechanism developed in this paper, i.e., split-issue, supports precise interrupts. However, the issues involved are too numerous and subtle to be dealt with summarily. The hardware support needed for speculative execution is very closely related to that for providing precise interrupts. In both cases, it must be possible to back up instruction issue to an earlier point and then resume execution from there correctly. Since we are not addressing precise interrupts, we shall also ignore the topic of speculative execution. Lastly, we shall simplify our discussion by ignoring predicated execution [13, 3]. Predicated execution poses some difficult problems for out-of-order execution which are unrelated to whether the architecture in question is VLIW.

In Section 2 we review dynamic scheduling and out-of-order execution for UAL programs. In Section 3 we examine the manner in which the semantics of a NUAL program are to be interpreted and we introduce the concept of split-issue. Section 4 extends UAL dynamic scheduling techniques and mechanisms to the NUAL domain and evolves the structure of the delay buffer--the minimal additional hardware structure required to support scoreboarding and out-of-order execution of NUAL programs. Section 5 attempts to place these new ideas in perspective.

2 Dynamic scheduling of UAL programs

The semantics of a conventional, sequential program are understood by assuming that each instruction is completed before the next one is begun. If program time is

measured in units of instructions issued, the execution latency of every operation is one cycle. If the actual latency of all operations is in fact a single cycle, then an instruction may be issued every cycle for a UAL program without the need for any interlock hardware and without any danger of violating the semantics of the program.

If some or all of the actual execution latencies are greater than one cycle, or if one wishes to issue more than one instruction per cycle, then it is necessary to provide instruction issue logic to ensure that the semantics of the program are not violated. In particular, it is important for the issue logic to understand when an instruction is dependent upon another one as a result of their accessing the same register. The determination of such dependences relies upon the knowledge that a UAL program is being executed; the semantics of a given operation, and the data that it uses as its input, assume that every sequentially preceding operation has completed before it begins execution.

Since instruction issue policies for NUAL programs build upon those for UAL programs, we shall briefly review the latter. An instruction issue policy is defined by the types of dependences whose occurrence it precludes and by its actions when a particular type of dependence is encountered. All correct issue policies must honor the partially ordered dependence graph that exists between the reads and the writes to a particular register. A large amount of work has been done in this area, and it has been pulled together and analyzed admirably by Johnson [10]. We shall review these policies from a somewhat unusual viewpoint, one that is better suited to the extension of these policies to NUAL programs.

Instruction issue policies may be broadly divided into two approaches.

A. The contents of a register can either be an actual datum or a symbolic value, i.e., a surrogate for or the name of the as yet uncomputed datum.

B. The contents of a register may only be a datum.

In the first case, even though the result of an operation will not be available for some time, a tag can be allocated to represent its symbolic value and this tag can be "written" to the destination register immediately or, in other words, associated with that destination register. Since this happens in the same cycle that the operation was issued, the operation appears to have unit latency when viewed at the level of symbolic values. Furthermore, when an operation is issued, there is always a value available in the source registers, either an actual value or a symbolic one. Consequently, instruction issue need never be interrupted unless the pool of tags runs out.

Of course, with the exception of copy operations, operations cannot proceed until the actual values of their source operands are available[1]. In the meantime, they wait in reservation stations [20]. Each time an operation completes, the tag corresponding to the symbolic value for the result is broadcast along with the actual value. Every register or reservation station containing this symbolic value replaces it with the actual value. Also, at this time, the tag for the result is returned to the pool of tags available for re-allocation. When the actual values for both source operands are available, the reservation station contends for the functional unit on which the operation will be executed.

This approach can be sub-divided into two policies of interest based on the number of tags that are available to serve as the symbolic value of a given register.

A1. Multiple tags can be allocated to represent multiple, distinct symbolic values associated with a given register.

A2. Only a single tag is available to represent the multiple, distinct symbolic values associated with a given register. For convenience, and without loss of generality, we shall assume that this pre-allocated tag is identical to the address of the register.

Policy A1, with minor and insignificant differences, is what is commonly known as the Tomasulo algorithm [20]. This entails the use of reservation stations and register renaming.

Policy A2 corresponds to the use of reservation stations (without renaming) to enable the issuance and setting aside of operations, the actual values of whose source operands are not as yet available, or operations for which a functional unit is not immediately available. However, since there is only a single tag available to use as a symbolic value, there cannot be more than one outstanding update of a register at any one time. Thus, instruction issue must block on an output dependence.

In the case of approach B, since one deals only with actual values, the illusion of single cycle execution cannot be sustained. Two instruction issue policies can be defined for this approach

B1. Stall instruction issue if a dependence is encountered, i.e., if either the source or the destination register, for the operation that is about to be issued, has a pending write.

B2. Continue instruction issue even when dependences are encountered, but provide mechanisms that enforce the partial ordering of accesses to each register.

[1] This is not strictly true. For instance, an integer multiply operation can proceed even if one of its source operands is a symbolic value, if the actual value of the other source operand is known to be zero. However, the practical benefits of so doing are quite doubtful.

The second policy leads one to mechanisms such as the dispatch buffer [1] or partial renaming [10]. Johnson has argued persuasively that the second policy is not worth pursuing since it leads to implementations that are more expensive but less effective than those for A1 and A2. Consequently, we shall limit ourselves to considering only policies A1, A2 and B1.

Our discussion, thus far, has been in the context of a single register file. Since all operations source and sink the same register file, no distinction need be made between the register access policy and the instruction issue policy. When there are multiple register files and operations which source one register file but sink a different one, the instruction issue policy depends upon the register access policies of both register files. We need a way to talk about instruction issue policies and register access policies as distinct entities. The view that we shall adopt is that the policies A1, A2 and B1 are register access policies which describe the manner in which a register file and its contents can be manipulated. Each register access policy specifies certain actions and constraints that apply when that register file is a source or a destination of an operation.

Table 1 codifies the actions and constraints of each register access policy (column 1) when that register file is the source (column 2) and when it is the destination (column 3). As a source there are two possibilities: SF and RS. SF states that instruction issue stalls when a flow dependence is encountered. RS specifies the use of reservation stations to set the operation aside when a flow dependence is encountered. As a destination, too, there are two possibilities: SO and RR. SO states that instruction issue stalls when an output dependence is encountered. RR specifies the use of register renaming to eliminate all output dependences.

Table 1. The three instruction issue policies of interest for a UAL program.

Register File Policy	Instruction Issue Policy	
	Source Operands	Destination Operand
A1	RS	RR
A2	RS	SO
B1	SF	SO

When the source and destination register files are the same, register access policies A1, A2 and B1 correspond to the instruction access policies RSRR, RSSO and SFSO, respectively. (The first two and last two letters indicate the policies for the source operands and the destination operands, respectively.)

3 Semantics of NUAL programs

The semantics of a sequential program are understood by viewing each instruction as occurring atomically, within a single cycle, and concurrent with no other instruction. In contrast, the semantics of a NUAL program must recognize that each operation has two distinct events, in general, at two distinct points in time. These are the start of the operation, when the source registers are accessed, and the end of the operation, when the destination register is written[1]. Each of the pair of events for one operation may have a precedence relationship with either one of the pair of events for another operation. Correct execution of a NUAL program demands that all of these precedence relationships be honored. The time at which these events occur in a NUAL program is measured in units of instructions issued. Since an instruction is a set of operations that is intended to be issued in a single cycle, this is equivalent to measuring time in cycles if one instruction is issued every cycle. When there is the potential for confusion, we shall refer to this as the virtual time of the program to distinguish it from the real, elapsed time during execution.

3.1 Dependence semantics of a NUAL program

Consider the fragment of a NUAL program shown in Figure 1a. Since it is a UniOp program, we shall refer to each operation by the number of the instruction that it is in. If this is interpreted as a UAL program, the first load, operation #1, is irrelevant since r1 is immediately overwritten by the operation #2. Operations #11 and #12 are both flow dependent upon the operation #2, operation #11 is irrelevant, operation #14 is flow dependent upon operation #12 and operation #15 is flow dependent upon operation #14.

Figure 1b illustrates how this NUAL program fragment should be interpreted correctly to understand the actual semantics and dependence structure, assuming the latencies as specified. Each operation in Figure 1a has been split into two operations in Figure 1b. The Phase1 operation consists of the source register reads and the actual computation. The Phase2 operation consists of the destination register write and is understood to execute in a single cycle. Anonymous temporary registers (v1, ..., v5) convey the results of the Phase1 operations to the corresponding Phase2 operations. These temporary values, by their very nature, are written and read exactly once each. In Figure 1b, a Phase2 operation is interpreted as being

[1] This can be generalized to more than two events if, for instance, different inputs are sampled at different times or if different outputs are written at different times by an operation.

issued later than the corresponding Phase1 operation by an interval equal to the assumed latency less one cycle.

Instruction	Operation
1	r1 = load(r2)
2	r1 = load(r3)
3	
4	
5	
6	
7	
8	
9	
10	
11	r4 = fmul(r1, r5)
12	r4 = fadd(r1, r6)
13	
14	r7 = fmul(r4, r9)
15	r7 = fadd(r7, r8)
16	
17	

(a)

Instruction	Phase1 Operation	Phase2 Operation
1	v1 = load(r2)	
2	v2 = load(r3)	
3		
4		
5		
6		
7		
8		
9		
10		r1 = v1
11	v3 = fmul(r1, r5)	r1 = v2
12	v4 = fadd(r1, r6)	
13		r4 = v4
14	v5 = fmul(r4, r9)	r4 = v3
15	v6 = fadd(r7, r8)	
16		r7 = v6
17		r7 = v5

(b)

Figure 1. (a) A NUAL code segment and (b) its UAL code equivalent after splitting. The assumed operation latencies are 10 cycles for load, 4 cycles for floating-point multiply and 2 cycles for floating-point add. The empty instructions are understood to contain no-op operations.

From an inspection of Figure 1b, it is now clear that operation #11 is flow dependent upon operation #1, operation #12 upon operation #2, operation #14 upon operation #12, and operation #15 is not flow dependent upon any of the other five operations. Operation #15 is also irrelevant unless there is a Phase1 operation that reads r7 in instruction 17. Thereafter, the value in r7 is that deposited by operation #14. Furthermore, there is an anti-dependence *from operation #14 to operation #11*! Operation #11 may not write r4 before operation #14 reads it, otherwise operation #14 gets that value rather than the result of operation #12.

What Figure 1b illustrates is that we can interpret NUAL programs as if they are UAL programs once the operations have been split into their Phase1 and Phase2 components and the Phase2 component is understood to issue with a delay corresponding to the assumed latency. One might suspect that if a program can be interpreted as if it is a UAL program that it can also be dynamically scheduled using all the mechanisms and techniques that have been developed for UAL programs. This is, in fact, the case. The concept of splitting a NUAL operation and delaying the issuance of the Phase2 operation we shall refer to as **split-issue**. Also, we shall use the term **augmented (MultiOp) instruction** to refer to the set of Phase1 operations from a single MultiOp instruction along with all of the Phase2 operations (from earlier MultiOp instructions) that are supposed to issue concurrently with these Phase1 operations.

3.2 Split-issue

Split-issue is the mechanism which permits correct execution of a NUAL program even when the actual latencies do not agree with the assumed latencies. Furthermore, it enables well understood out-of-order-execution techniques to be employed with NUAL programs. We shall describe the concept here in its most general form. The general hardware model is described in Section 4.1. In certain special cases of interest, it simplifies to a rather inexpensive implementation.

With UAL programs, instruction interpretation comprises three steps (given that we are ignoring the precise interrupt issue). These are

1. decode and issue,
2. initiate, and
3. complete and retire

With NUAL programs we add one more action which is that of splitting. Once an instruction is in the instruction register, each operation is decoded and split into its Phase1 and Phase2 components. An anonymous register is assigned to be the destination of the Phase1 operation and the source for the Phase2 operation. The Phase1 operation is issued immediately (in virtual time) in accordance with the instruction issue policy that is being employed (Figure 2a). Either immediately or eventually, it is initiated, i.e., begins execution, then completes and is retired. The Phase2 operation is inserted into a list which is ordered by the virtual time at which the Phase2 operations should be issued[1] (Figure 2b). For each Phase2 operation this is computed as the virtual time at which the Phase1 operation is issued plus the assumed latency less one cycle. After the appropriate delay (measured in units of MultiOp instructions issued), the Phase2 operation is

[1] This is the conceptual view. There are any number of ways of actually implementing this.

issued. It executes, i.e., performs the copy from the anonymous register to the architectural register, either immediately or when appropriate, and is retired.

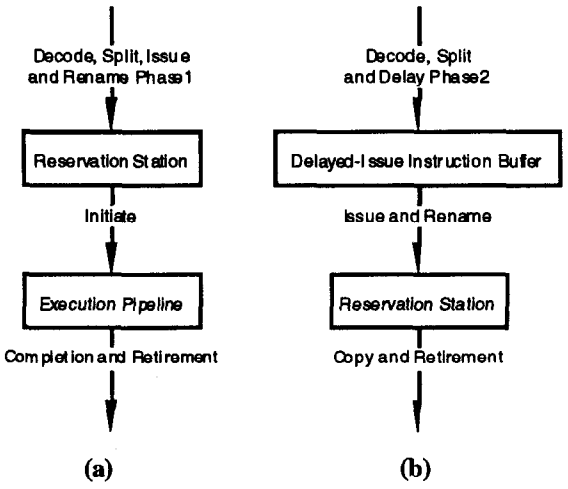

Figure 2: (a) Execution phases for a Phase1 operation. (b) Execution phases for a Phase2 operation

3.3 Specification of the assumed latency

The latency assumed by each operation may be specified in a number of ways. In decreasing order of generality and flexibility, these are:

- a field in each operation specifying the assumed latency,
- an **execution latency register (ELR)** per opcode or per set of opcodes which contains the assumed latency of that opcode or opcode set, and
- an architecturally specified latency for each opcode.

The first approach permits the specification of distinct assumed latencies for different occurrences of the same opcode. Although this can be quite useful, it is rather extravagant in its use of instruction bits. The Horizon architecture provides for such a latency specification per MultiOp instruction [17]. Presumably, the value specified is the minimum of the assumed latencies across all operations within a single instruction.

The second approach has two sub-cases depending on how the assumed latency is deposited into the ELR. One option is to provide all the assumed latencies in the program header. Prior to launching the program, the runtime system transfers this information into the ELR's which are part of the processor state, but inaccessible to user code. The second, more dynamic option is to make the ELR's visible to the program and to provide opcodes that load and, perhaps, store the contents of the ELR's. This second option provides the capability to keep changing the assumed latency of an opcode albeit not as flexibly as with the latency-field-per-operation approach. (Such a capability was provided in the Cydra 5 for specifying the assumed latency of load operations [12].)

With the third approach, there is no explicit specification of the assumed latencies. Instead, they are specified in the architecture specification and are fixed across all programs and across all processors within the architectural family. Only in this last case is it appropriate to use the term "architectural latencies" for the assumed latencies. This is the approach commonly used in the past by VLIW processors [4, 5, 3].

It is worth noting that there are a number of situations, having to do with the robustness of performance with respect to varying actual latencies, in which it is advisable for the program to assume latencies that are quite different from the actual hardware latencies [11].

4 Dynamic scheduling techniques for NUAL programs

4.1 A machine model

The general machine model assumed in this paper is shown in Figure 3. Instructions are fetched or prefetched into the instruction buffer as in any other processor. These instructions are assumed to be MultiOp (which includes UniOp instructions as a special case). An additional, post-decoding step, which we have termed splitting, exists. During this step, each operation in the instruction that is about to be issued is split into its Phase1 and Phase2 components by the splitter. The Phase2 operations are placed in the **delayed-issue instruction buffer**, appropriately far back, so as to be issued with a delay that is one less than the assumed latency[1]. The Phase1 operations are placed in the instruction register immediately along with any Phase2 operations which are at the front of the delayed-issue instruction buffer. The set of Phase1 and Phase2 operations that are placed in the instruction register during the same cycle constitute an augmented (MultiOp) instruction.

The instruction issue unit performs one of three actions upon each operation in the instruction register depending on the current situation. If the operation has no

[1] The assumption here is that the code has been scheduled such that two operations on the same functional unit never complete at the same time since this would constitute an over-subscription of the result bus. This also implies that there will never be the need to schedule two Phase2 operations on the same functional unit at the same time.

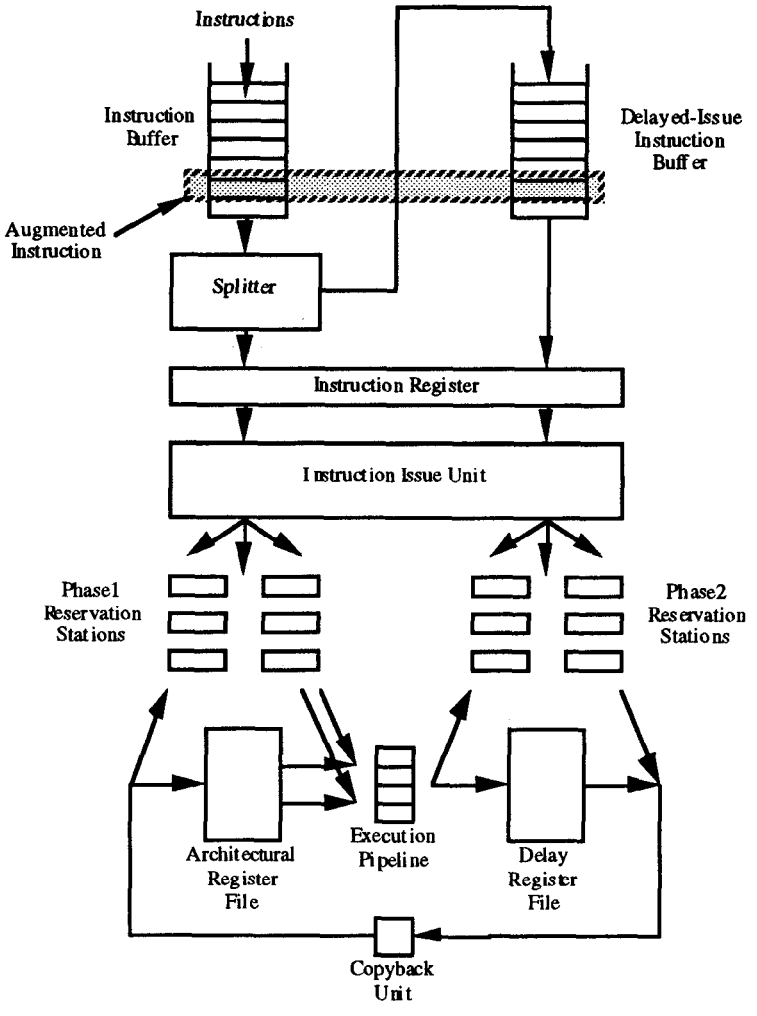

Figure 3. Processor organization for supporting split-issue with out-of-order execution.

dependence conflict with any previously issued operation, and if the appropriate functional unit is available, the operation goes directly into execution. If either a dependence conflict exists or if the required functional unit is not available, the operation is placed in a reservation station to await the condition under which it can go into execution. If no reservation station is available (including the case in which reservation stations are not provided in the hardware) the operation cannot be issued. All of the operations in a single augmented instruction are decoded and issued simultaneously from the instruction register. If even one operation cannot be issued, then none of them are issued. In this case, we shall say that instruction issue has stalled.

Phase1 and Phase2 operations are issued to distinct types of reservation stations, the structure of which we shall examine shortly. The functional units corresponding to the Phase1 operations are those that implement the functionality of the opcode repertoire, e.g., integer ALU's, floating-point adders and load-store units. In general, these functional units have latencies of one or more cycles and they may be pipelined. The implicit opcode for a Phase2 operation corresponds to a copy operation. This operation is assumed to complete in a single cycle. The "functional unit" that implements this is shown in Figure 3 as the copyback unit. Phase1 operations access the architectural register file, (ARF) for their source operands and access the anonymous **delay register file (DRF)** for writing the destination operand. Conversely, Phase2 operations access the DRF for their source operand and the ARF for writing the destination operand.

Figure 3 displays a single ARF, a single execution pipeline, a single DRF and a single copyback unit. In general, a processor might possess multiple ARF's (e.g.,

integer and floating-point). There may be multiple functional units that access a given ARF. Furthermore, certain operations may access one ARF as the source and another one as the destination. Without loss of generality, we shall focus on a single ARF. We shall assume that there is a unique DRF and copyback unit per functional unit. Thus, each DRF is written to only by one specific functional unit and is read by a single copyback unit. In comparison to the ARF, the DRF can be implemented less expensively since it has but a single read port and a single write port.

4.2 Instruction issue policies for NUAL programs

If we assume that all ARF's have the same register access policy and, likewise, that all DRF's have the same register access policy, then the instruction issue policies for the Phase1 and Phase2 operations are completely specified by the two register access policies (Table 2). For each pair of ARF access policy (row) and DRF access policy (column), the table entry specifies the corresponding Phase1 and Phase2 instruction issue policies. Given that we have restricted our discussion to three register access policies, there are nine possible NUAL instruction issue policies. We shall also find it useful to consider, for the ARF, the null access policy of having no interlocks whatsoever on accesses to the architectural registers[1]. This increases the number of possible instruction issue policies to twelve.

Consider first the four instruction issue policies that correspond to the use of B1 for the DRF. Since instruction issue stalls if the actual value of a Phase2 source operand is not available, and since Phase2 operations complete in a single cycle, a Phase2 operation can never complete late with reference to the program's virtual time. Consequently, at the beginning of every cycle in which instruction issue is not stalled due to a Phase2 flow dependence, every Phase1 operation will find that its (ARF) source operands are available and that there are no outstanding writes against its (DRF) destination. Therefore, it is unnecessary to do any dependence checking at all when accessing the ARF.

The instruction issue policy for which the DRF policy is B1 and the ARF is not interlocked is termed **latency stalling**. The nature of this policy is that instruction issue stalls when the assumed latency of an operation has elapsed, but the actual latency has not. In the program's virtual time, the assumed latency is never exceeded and so no dependence checking hardware is needed. This policy is of particular interest since it eliminates the requirement for interlock hardware on the ARF which, in an ILP processor, may be expected to be highly multiported.

The other three policies involving the use of B1 for the DRF degenerate to latency stalling and are not of interest as distinct policies. Also, if the ARF is not interlocked, the only acceptable policy for the delay registers is B1. Correctness requires that results never be written late, relative to the program's virtual time, into the ARF, which precludes A1 and A2. Since the DRF is an anonymous register file, it makes little sense to provide additional affiliated anonymous registers via register renaming of the DRF; one might as well have provided a larger DRF in the first place. Consequently, A1 is of little interest for the DRF. This leaves four policies of possible interest.

Table 2. Instruction issue policies of potential interest for NUAL programs. Entries specify the policies for Phase1 and Phase2 operations, respectively. The lightly shaded entries are of no interest. The heavily shaded entries are not feasible.

	Delay Register File Policy		
ARF Policy	**B1: SFSO**	**A2: RSSO**	**A1: RSRR**
No interlocks	Latency stalling		
B1: SFSO	SFSO / SFSO	SFSO / RSSO	SFRR/ RSSO
A2: RSSO	RSSO / SFSO	RSSO / RSSO	RSRR/ RSSO
A1: RSRR	RSSO / SFRR	RSSO/ RSRR	RSRR / RSRR

The operations that we have been considering thus far have been register-register operations. By generalizing from registers to all types of processor state, such as the program counter and the program status word, operations such as delayed branches can be included in this same framework. As far as their register-based dependences are concerned, loads and stores can also be viewed within the same framework. A load can be viewed as a register-register NUAL operation which takes an address from a source register and deposits the contents of the addressed memory location in the destination register. Likewise, a store has two source operands but no destination register. On the other hand, loads and stores also have dependences between one another via their accesses to the memory space. Conceptually, these too, can be treated by viewing the entire memory space as a register file. In practice, given the size of the memory space, different scoreboarding and out-of-order mechanisms (e.g., associative store buffers

[1] We ignore the case in which the DRF is not interlocked. This is feasible only if all actual latencies are guaranteed to be less than or equal to the assumed ones. In this case the ARF, too, would have no interlocks.

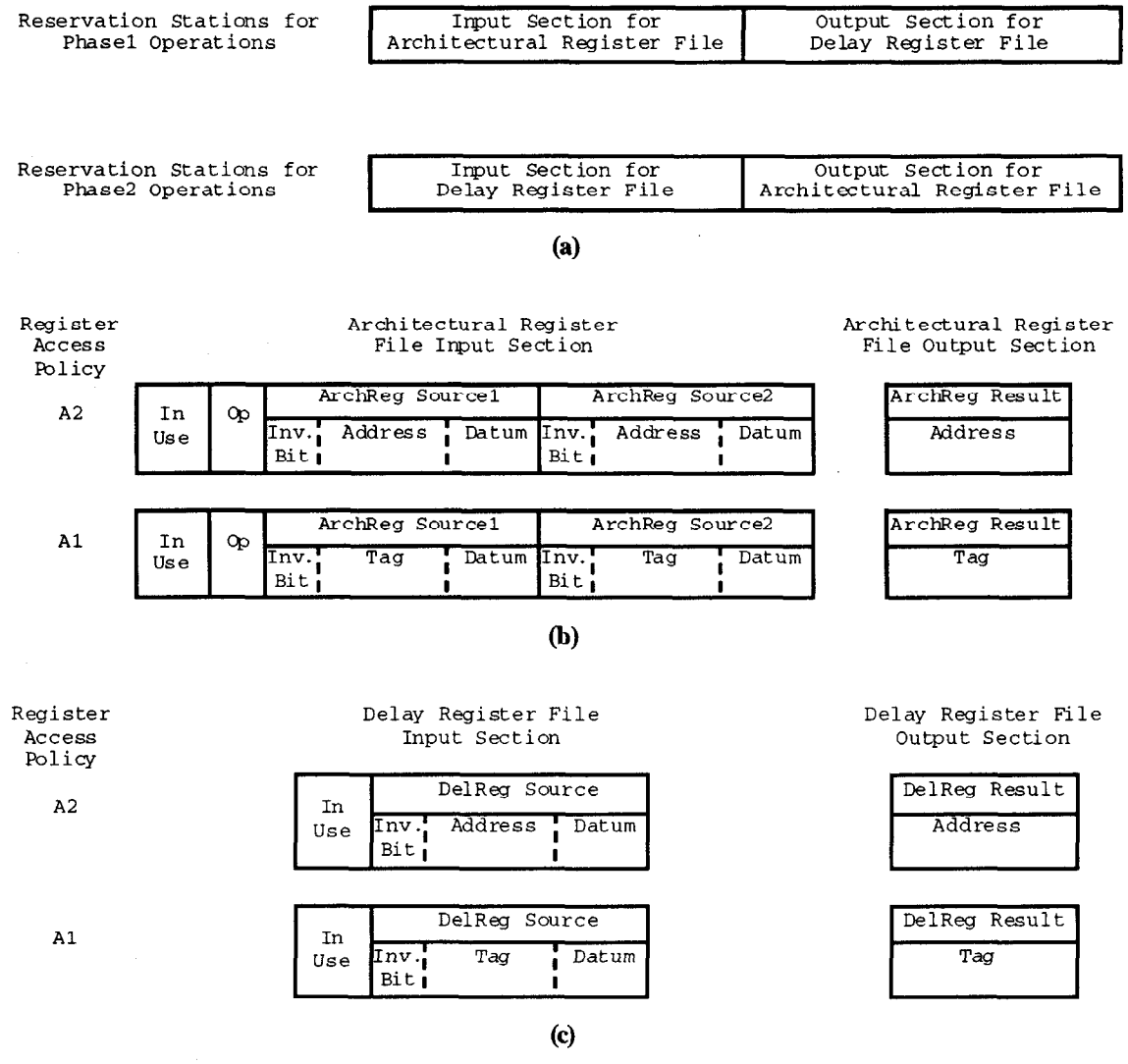

Figure 4. (a) The structure of reservation stations for Phase1 and Phase2 operations. (b) The detailed structure of the reservation station input and output sections that are associated with the ARF's. (c) The detailed structure of the reservation station input and output sections that are associated with the DRF's.

[10]) must be employed than those that are used for register-register operations.

4.3 General structure of the reservation stations for NUAL

Figure 4 shows the structure of the reservation stations. A reservation station consists of two parts: the input section which relates to the source operands of the operation, along with the opcode and the output section which pertains to the result operand (Figure 4a). The detailed structure of each section depends upon the instruction issue policy as well as on the type of register file to which it corresponds. For a Phase1 operation, the input section is that for an ARF and the output section is that for a DRF. For a Phase2 operation, it is just the opposite. Figures 4b and 4c display the detailed structure of the input and output sections corresponding to the ARF and the DRF, respectively.

With access policy A1, each architectural register consists of either the datum itself or the tag for its symbolic value and an invalid bit. The invalid bit is set if the register contains a tag. That portion of the input section corresponding to each source operand has the same structure as an architectural register. Additionally, the input section contains the opcode and a bit to indicate that

this reservation station is in use. The output section for the ARF under policy A1 consists of the tag that represents the symbolic value of the result (so that it can be broadcast along with the actual value once it is available). (In the case of UAL and the absence of split-issue, the reservation station under policy A1 would consist of this pair of input and output sections.)

Under policy A2, the tag is identical with the address of the architectural register on both the source and destination sides. Other than this, the structures of the input and output sections are identical. The register structure is simplified; since the tag is identical to the register's address, it need not be explicitly represented and no storage is required. Only space for the datum and the invalid bit are needed. With policy B1, no reservation stations are needed and the register structure is the same as that under policy A2.

In principle, the input and output sections associated with the DRF are similar to those for the ARF, except for a couple of differences First, since there is only one possible opcode for a Phase2 operation (copy), it need not be explicit in the input section. Second, a Phase2 operation has a single source operand. Together, these simplifications yield the input and output sections shown in Figure 4c for policies A1 and A2.

4.4 The delay buffer: simplified hardware support for NUAL

Certain simplifications result from the stylized manner in which the delay registers and the Phase2 reservation stations are used and the fact that register access policy A1 is of little interest for the DRF. Through a series of simplifications, that are discussed in detail in an expanded version of this paper [11], three structures--the delayed-issue buffer, the DRF and the Phase2 reservation stations--are combined into one hardware structure which we shall refer to as the **delay buffer**. This results in a number of redundant fields that can be optimized away. Each element of the delay buffer has the resulting structure shown in Figure 5.

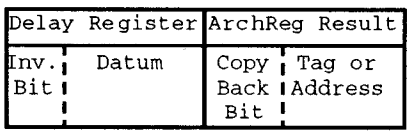

Figure 5. Structure of an element of the delay buffer.

The delay buffer is organized as a circular buffer. The **Phase2 Issue Pointer (PIP)** points to an element of the delay buffer and, each time an augmented instruction is issued, the PIP moves forward by one element. On each instruction issue cycle, the Phase2 operation that has just been split off is allocated the delay buffer element that is ahead of the PIP by the assumed latency less one. The address of this element is used by the Phase1 operation to specify its destination. The invalid bit in this element is set at this time (to be reset when the Phase1 operation completes and the result is written into that delay buffer element). At this time, the copyback bit in this element should be in the reset state. The Phase2 operation is issued when the PIP arrives at this element.

If the result is computed sooner than the assumed latency, it is written into the delay buffer element, the invalid bit is reset but, since the copyback bit is not set, it is not written to the destination architectural register. When the PIP reaches this delay buffer element, the invalid bit is not set and, so, the datum is written to the destination architectural register.

If the result is computed later than the assumed latency, then the invalid bit is still set when the PIP arrives at this element. Two actions are taken. First, if access policy A1 is being employed for the ARF, the address of the destination architectural register in the delay buffer element is replaced by the tag for the symbolic value that is placed in the destination architectural register. Second, the copyback bit is set to indicate to the hardware that the result should be written to the destination architectural register as soon as it is available. As a result, the copyback bit will be set at the time that the result is eventually written into the delay buffer element. Accordingly, the result will also be written to the destination architectural register and both the invalid and the copyback bits will be reset. When both the invalid and the copyback bits are reset, the delay buffer element is no longer in use and may be reallocated.

If the access policy for the ARF is A1, the copyback can be performed immediately upon the PIP reaching a delay buffer element. Since the Phase2 operation is a copy, it can be executed immediately even though its source datum is not available. This is done by copying the symbolic value in the delay buffer element (i.e., the address of that element[1]) into the destination architectural register and resetting the copyback bit. When the corresponding Phase1 operation completes, it broadcasts the delay buffer address (as the tag) along with the datum. The architectural register and any Phase1 reservation stations that contain that tag replace their tags with the datum. At the same time, the invalid bit in the delay buffer element is reset. The copyback bit can be eliminated since it would be reset the very cycle it is set. Thus, the delay buffer element

[1] More precisely, the tag should consist of the delay element address prefixed by the functional unit identifier so that the tags corresponding to different functional units are distinct.

is no longer in use as soon as the PIP has passed over it and the invalid bit is reset. Also, note that the allocation of a tag for the destination architectural register is no longer needed since the address of the delay buffer element serves that function.

Instruction issue must stall either if the delay buffer element that is about to be allocated is still in use or if, in a wraparound sense, it is more than one lap ahead of the PIP. The latter constraint implies that there must be at least as many delay buffer elements as the longest assumed latency, else deadlock will occur. Thus, the maximum possible assumed latency, for the operations that execute on each functional unit, is an architectural parameter.

With latency stalling, instruction issue must stall when the PIP points to an element whose invalid bit is still set and can only resume once that bit is reset. In this case, too, the copyback bit is redundant (it will never be set) and may be eliminated from the structure of the delay buffer element. A delay buffer element is in use from the time that its invalid bit is set until the PIP passes over it. Interestingly, this is almost exactly what the collating buffers associated with the Cydra 5's memory pipelines were [12]. In the Cydra 5, this capability was motivated by the variability of the load latency due to interference in the interleaved main memory. The assumed latency for loads was specified by writing the assumed latency into the memory latency register (MLR). It is of obvious value to extend this concept to all the functional units in order to address the variability of hardware latencies across multiple implementations of a NUAL architecture. An attractive property of latency stalling is that it scales well to large numbers of functional units since each functional unit independently decides whether instruction issue should be stalled.

The delay buffer has only a single read port, a single write port, no associative hardware and a very simple allocation/deallocation process. All of these contribute to its being relatively inexpensive. Furthermore, most of this hardware is already present in some other form. The normal staging of pipelined operations in UAL architectures requires that the destination address be buffered from the time of issue until the result is written to the architectural register. This hardware is subsumed by the delay buffer structure that we have developed. The delay registers provided in the delay buffer to hold data until it is time to write them to the ARF would show up as additional architectural registers in a UAL architecture. It would appear to be a poor trade-off to replace delay registers which have one port each for reads and writes with architectural registers which must necessarily be highly multiported.

4.5 Multiple instruction issue

Multiple instruction issue is important either if the hardware has the ability to issue more operations in parallel or if the hardware latencies are shorter than those assumed by the compiler. In the latter case, the opportunity to execute operations earlier in real time than in the program's virtual time can only be realized if instructions are scanned (and issued) faster than one per cycle. (Note that by multiple instruction issue we mean the issuance, each cycle, of more than one MultiOp instruction, each one containing multiple operations.) The program's virtual time must advance by multiple cycles on each unstalled real cycle. There is a variety of ways to accomplish this, but in each case the net effect must be identical to that obtained by issuing one MultiOp instruction at a time while running the instruction issue logic with a cycle time that is a sub-multiple of the actual instruction issue cycle time.

Each of the above instruction issue policies may be combined with multiple instruction issue. If policy A2 is used for the delay buffer, the implementation issues are very similar (other than the additional requirement of performing split-issue) to those for a multiple-issue superscalar processor using the same policy as that used for the ARF [10]. Although dependence checking is unnecessary between the multiple operations within any one of the instructions being issued, every operation must be checked against every operation from a preceding instruction. Data path complexities to support multiple instruction issue are common to both VLIW and superscalar. One example is the need for register forwarding between operations that are writing to the ARF and flow dependent successor operations that are being issued in the same cycle.

If latency stalling is employed, the determination of which instructions may be issued together is greatly simplified. The only condition that can prevent the issue of the next instruction on a given cycle is if the invalid bit pointed to by the PIP is set in one or other of the functional units. By extension, the only condition that can prevent the issue of the next n instructions is if the invalid bit is set in any functional unit for the element pointed to by the PIP or in any of the next n-1 elements. Thus, the equivalent of a priority encoder determines the maximum number of instructions that may be issued simultaneously. This would appear to be considerably less complicated than what is entailed with a superscalar processor. In particular, no dependence checking is required when accessing the ARF. However, the data path complexity alluded to above is unchanged. The PIP is advanced by n on each non-stalled cycle in which n augmented instructions are issued.

5 Discussion

The purpose of this paper was to establish that VLIW and dynamic scheduling are not contradictory concepts. This we have done. Whereas latency stalling is a simple and eminently practical scheme for use with VLIW, the delay buffer in conjunction with split-issue can support arbitrarily sophisticated access policies for the ARF, leveraging off all the dynamic scheduling techniques that have been, or will be, developed for superscalar processors.

Although possible, there is still the question of how desirable it is to perform dynamic scheduling that is more complex than latency stalling. The author is not a proponent of using dynamic scheduling to effect large-scale code re-ordering at runtime, whether for VLIW or superscalar processors. The hardware penalties are just too high. This function is best performed by a latency-cognizant compiler. Nevertheless, it is clear that small-scale re-ordering or, at the very least, some form of interlocking is unavoidable in the face of variable delays and the need for object code compatibility across a family of machines. Let us consider the complexity of dynamic scheduling as we escalate it in three steps.

The simplest scenario is when the hardware has exactly the same latencies and functional units as were assumed by the compiler. In this case, neither from a correctness viewpoint nor from a performance viewpoint is it necessary to have any dynamic scheduling logic. This, of course, was the original attraction of VLIW. However, it suffers from the problem of object code compatibility, which was the motivation for this paper.

The next step is to ensure correctness in the face of actual latencies that could be longer (or shorter) than those assumed. This is the minimum one must do to guarantee object code compatibility. This is provided by the delay buffer which, as pointed out in Section 4.4, requires little additional harware. The simplest mechanism available for VLIW is latency stalling which entails no interlocks on ARF access. The delay buffer can support arbitrarily complex access policies for the ARF, but we claim below that this is impractical.

The last step is to realize the performance benefits that result from taking advantage of shorter than assumed latencies or of greater hardware parallelism than expected. This requires multiple (MultiOp) instruction issue. To be effective, this most likely implies out-of-order execution of Phase1 operations, i.e., either policy A1 or A2 for the ARF.

The practicality of each tier of complexity hinges on the practicality of two capabilities: multiple instruction issue and ARF access policies B1, A2 and A1. Practicality must be evaluated in the context of the level of ILP desired. Of interest to proponents of ILP are processors capable of issuing eight or more operations per cycle. (It might help the reader to view the following discussion in light of a more concrete processor. For instance, one that has separate integer and floating-point register files and permits the issue of two load/store operations, three integer operations, two floating-point operations and a branch operation per cycle.)

Johnson [10] has shown that multiple instruction issue for superscalar requires a number of comparators that is proportional to N(N-1). The source and destination registers for each instruction that is a candidate for issue must be compared with those for all the sequentially preceding instructions which also are candidates for issue. This is required so that one can determine, in parallel, which of those instructions may, in fact, be issued without violating any dependences.

In contrast, the issuance of a single MultiOp instruction with N operations incurs little complexity since the compiler guarantees their independence. This is one of the original attractions of VLIW. However, if N operations from multiple MultiOp instructions are to be issued simultaneously, similar dependence checking logic is required. Since it still is unnecessary to compare operations that are from the same instruction, the number of comparators is reduced, but only by a factor of at most two. In either case, multiple instruction issue involving large numbers of operations is impractical. (As shown in Section 4.5, these comparators are unnecessary if latency stalling is used. However, the benefits derived from multiple instruction issue, when latency stalling is used, are unclear.)

On the basis of such considerations, the author is led to the conclusion that multiple instruction issue is unrealistic when the number of operations per cycle is large. This, regardless of whether the underlying architecture is VLIW or sequential. VLIW does have the advantage via its MultiOp capability to issue multiple operations per cycle without issuing multiple instructions (i.e., a single MultiOp instruction) whereas with superscalar processors multiple operation issue must *always* involve multiple instruction issue.

If one retreats to the objective merely of ensuring correctness in the face of longer actual latencies, one enters the realm of the feasible. VLIW, via its NUAL attribute, can employ latency stalling which involves low complexity and is quite scalable to the high levels of ILP that are made practical by MultiOp. If the use of policies B1, A2 and A1 are practical for the ARF at high levels of ILP, then they can be employed by VLIW just as well as they could by superscalar. (In addition, superscalar must take on the full complexity of multiple instruction issue, even if the only possible error in the compiler's

assumptions have to do with latency and not with the number of functional units.)

Unfortunately, there is little hard data to support or refute the position taken by the author other than at the "existence proof" level; although there have been a number of VLIW products built that were able to issue eight or more operations per cycle [4, 5, 3], there are no similar examples for superscalar products at the same levels of ILP. This is not to suggest that igenuity will never lead to a breakthrough in the area of dynamic scheduling. If, in fact, multiple instruction issue and ARF interlocking turn out to be practical at high levels of ILP superscalar may become a viable alternative at high levels of ILP. By the same token, these breakthroughs could just as well be exploited by VLIW.

6 Conclusions

We have demonstrated that VLIW processors are as capable of out-of-order execution and multiple instruction issue as are superscalar architectures. The attributes of VLIW that are central to successfully achieving high levels of instruction-level parallel execution along with object code compatibility are MultiOp and NUAL. The key mechanism for enabling the dynamic scheduling of NUAL programs is split-issue, and the preferred hardware support for it is the delay buffer. Latency stalling is a particularly simple interlock technique that can be used with NUAL programs.

Acknowledgments

The distinction between VLIW as a processor implementation and VLIW as an architecture developed in the course of discussions with Josh Fisher, as did the understanding of the fact that program semantics with NUAL are defined by the assumed latencies. The recognition, that the memory latency register concept from the Cydra 5 could be extended to all functional units and applied to solve the problem of code compatibility, is due to Mike Schlansker. This paper has benefited greatly from discussions with Mike, Vinod Kathail, Phil Kuekes and Dennis Brzezinski.

References

1. Acosta, R.D., Kjelstrup, J., and Torng, H.C. An instruction issuing approach to enhancing performance in multiple function unit processors. *IEEE Transactions on Computers C-35*, 9 (September 1986), 815-828.
2. Anderson, D.W., Sparacio, F.J., and Tomasulo, R.M. The System/360 Model 91: machine philosophy and instruction handling. *IBM Journal of Research and Development 11*, 1 (January 1967), 8-24.
3. Beck, G.R., Yen, D.W.L., and Anderson, T.L. The Cydra 5 mini-supercomputer: architecture and implementation. *The Journal of Supercomputing 7*, 1/2 (May 1993), 143-180.
4. Charlesworth, A.E. An approach to scientific array processing: the architectural design of the AP-120B/FPS-164 Family. *Computer 14*, 9 (September 1981), 18-27.
5. Colwell, R.P., Nix, R.P., O'Donnell, J.J., Papworth, D.B., and Rodman, P.K. A VLIW architecture for a trace scheduling compiler. *IEEE Transactions on Computers C-37*, 8 (August 1988), 967-979.
6. DeLano, E., Walker, W., Yetter, J., and Forsyth, M. A high speed superscalar PA-RISC processor. In *Proc. COMPCON '92*, (February 1992), 116-121.
7. Diefendorff, K., and Allen, M. Organization of the Motorola 88110 superscalar RISC microprocessor. *IEEE Micro 12*, 2 (April 1992), 40-63.
8. Groves, R.D., and Oehler, R. An IBM second generation RISC processor architecture. In *Proc. 1989 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, (October 1989), 134-137.
9. Hwu, W.W., and Patt, Y.N. Checkpoint repair for out-of-order execution machines. *IEEE Transactions on Computers C-36*, 12 (December 1987), 1496-1514.
10. Johnson, M. *Superscalar Microprocessor Design*. Prentice-Hall, Englewood Cliffs, New Jersey, 1991.
11. Rau, B.R. Dynamic scheduling techniques for VLIW processors. Technical Report HPL-93-52. Hewlett-Packard Laboratories, 1993.
12. Rau, B.R., Schlansker, M.S., and Yen, D.W.L. The Cydra 5 stride-insensitive memory system. In *Proc. 1989 International Conference on Parallel Processing*, (August 1989), 242-246.
13. Rau, B.R., Yen, D.W.L., Yen, W., and Towle, R.A. The Cydra 5 departmental supercomputer: design philosophies, decisions and trade-offs. *Computer 22*, 1 (January 1989), 12-35.
14. Smith, J.E., Dermer, G.E., Vanderwarn, B.D., Klinger, S.D., Roszewski, C.M., Fowler, D.L., Scidmore, K.R., and Laudon, J.P. The ZS-1 central processor. In *Proc. Second International Conference on Architectural Support for Programming Languages and Operating Systems*, (Palo Alto, California, October 1987), 199-204.
15. Smith, J.E., and Pleszkun, A.R. Implementing precise interrupts in pipelined processors. *IEEE Transactions on Computers C-37*, 5 (May 1988), 562-573.
16. Sohi, G.S., and Vajapayem, S. Instruction issue logic for high-performance, interruptable pipelined processors. In *Proc. 14th Annual Symposium on Computer Architecture*, (Pittsburgh, Pennsylvania, June 1987), 27-36.
17. Thistle, M.R., and Smith, B.J. A processor architecture for Horizon. In *Proc. Supercomputing '88*, (Orlando, Florida, November 1988), 35-41.
18. Thornton, J.E. Parallel operation in the Control Data 6600. In *Proc. AFIPS Fall Joint Computer Conference*, (1964), 33-40.
19. Thornton, J.E. *Design of a Computer - The Control Data 6600*. Scott, Foresman and Co., Glenview, Illinois, 1970.
20. Tomasulo, R.M. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of Research and Development 11*, 1 (January 1967), 25-33.