HEWLETT PACKARD

Dynamic Scheduling Techniques for VLIW Processors

B. Ramakrishna Rau
Computer Research Center
HPL-93-52
June, 1993

instruction-level parallelism, VLIW processors, superscalar processors, pipelining, multiple operation issue, scoreboarding, dynamic scheduling, out-of -order execution

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.

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 [1-3], it has just not proven feasible to build superscalar products with this level of ILP [4-9]. 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 VLIW processors being perceived as 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, at least in principal, are better equipped to deal with variable latencies. In fact, it is fair to say that 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 [4, 10], the register renaming scheme, known as Tomasulo's algorithm, incorporated in the IBM 360/91 [11, 5], the history file, reorder buffer and future file [12], the register update unit [13] and checkpoint-repair [14].

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 a 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 statements about the operation latencies, the 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, where operations have 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. In addition, a VLIW architecture guarantees the absence of any flow dependences between the operations in a single instruction.

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.

A point of confusion that is worth clearing up is the distinction between a latency-cognizant program and a latency-cognizant compiler. In the case of the former, the semantics of the program can be understood only with full knowledge of the latencies. A latency-cognizant compiler, on the other hand, makes use of its knowledge of or estimates of the latencies in crafting a good schedule but then, if generating code for a UAL architecture, generates a program that is not latency-cognizant, i.e., the semantics of the program can be understood by assuming that each operation completes in a single cycle.

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 [15, 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, the execution latency of every operation has to be 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.

2.1 Data dependences

The discussion of non-sequential instruction issue policies is facilitated by a parallel micro-model of an operation. The traditional dyadic operation, such as an integer add, may be viewed as composed of four micro-operations: two which read the specified source registers, one which performs the computation once the first two micro-operations have completed, and a fourth that

writes the result to the destination register after the computation is complete. The precedence relationships between these micro-operations is represented by the partial ordering shown in Figure 1a. We shall assume that buffering is available on each arc in this graph to permit the micro-operations to be performed asynchronously with respect to one another, subject only to the stated precedence relationships.

The correct ordering between the micro-operations of distinct operations is obtained by requiring, on a register by register basis, that all accesses to a given register are performed in the same order as that specified by the sequential program. This total ordering of accesses to a given register can be relaxed somewhat; whereas read-write access pairs must maintain the original sequential order between each other, as must write-write pairs, there is no need to maintain the ordering between read-read pairs that were not separated by a write. Thus, the original total ordering of accesses to a register can be replaced by the partial ordering shown in Figure 1b. Although the set of reads that are sandwiched between two writes must maintain their ordering with respect to the two writes, they are unordered with respect to one another. If two writes occur one after the other, in the original sequential ordering, without an intervening read, the ordering between the two writes must be preserved.

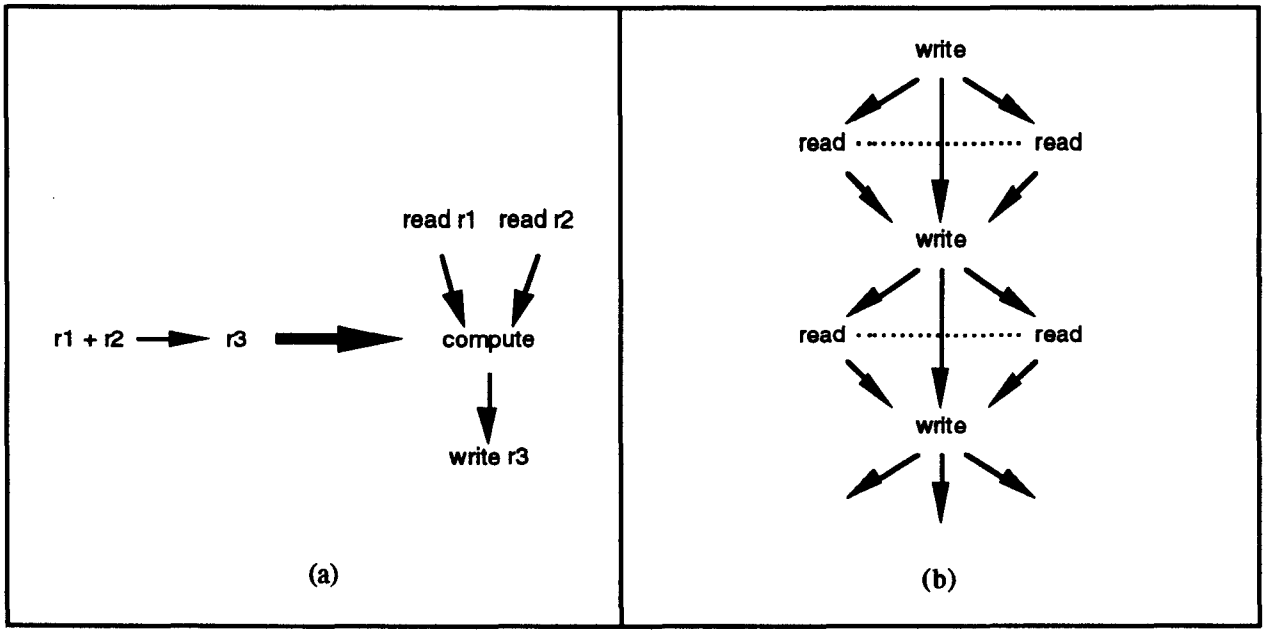

Figure 1. (a) The parallel micro-model of a dyadic operation. (b) The minimal partial ordering that must be maintained between the accesses to a given register to ensure correct semantics.

The precedence relationships in Figure 1b have been classified as flow, anti- and output dependences [16]. The precedence relationship between two successive writes is an output dependence, that between a read and the immediately preceding write is a flow dependence, and the precedence relationship between a read and the immediately succeeding write is an anti-dependence. Collectively, these dependences are termed data dependences. They must all be honored.

2.2 Instruction issue policies

We shall use the following terminology in this paper. An operation is said to have been **issued** once its dependences upon previously issued, but as yet uncompleted, operations have been analyzed and it has been released to execute, with no further involvement from the instruction issue unit, as soon as those dependences have been removed. An operation is said to be **initiated** when it actually begins executing on some functional unit. The operation has **completed** once it has finished execution on the functional unit, and it has **retired** once the side-effects of the operation, i.e., the update of the destination register and the flagging of any exceptions, have been committed to the processor state. Since we are ignoring the subject of interrupts and exception in this paper, completion and retirement collapse into a single event. Between issuance and initiation, the operation waits in a buffer that we shall refer to generically as a reservation station.

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 (Figure 1b). A large amount of work has been done in this area, and it has been pulled together and analyzed admirably by Johnson [17]. 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. 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 [11]. 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.

---

[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. In view of the complexity of implementing such a strategy and the questionable benefits derived from it, we shall ignore such possibilities.

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.

The second policy leads one to mechanisms such as the dispatch buffer [18] or partial renaming [17]. 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 can 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.) SFSO is the simplest policy in which instruction issue stalls whenever either a source or destination register has a pending write against it. All that is needed is an invalid bit per register to indicate that the register has a pending write. RSSO does not stall issue when a source register's invalid bit is set. Instead, it places the operation in a reservation station which is the additional hardware required by this policy. However, it does stall issue if the destination register's invalid bit is set. RSRR does not stall issue in either situation. The reservation station hardware is not appreciably more complex than for RSSO. However, hardware is now needed to allocate distinct tags and to return them to the pool once the result has been computed. Also, the register file is more complex because a tag must be associated with each register.

With two policies each on the source and destination sides, we have four possible instruction issue policies. The missing one, SFRR, causes instruction issue to stall when a flow dependence is encountered even though all of the mechanisms to deal with output dependences have been provided. Since flow dependences are, typically, the more important obstacle to successful out-of-order execution than are output dependences, it seems unreasonable to cater to the latter and not to the former. Accordingly, we do not consider this policy in this paper.

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. 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. 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

Any form of dynamic scheduling requires the ability to re-create the dependence graph of the NUAL program. So, the question is how does one do this given the program and the assumed operation latencies? As with UAL operations (Figure 1a), a NUAL operation may be viewed as composed of multiple micro-operations. The difference is that the write micro-operation at the end of the operation is understood to occur after the read micro-operations with a time interval equal to the assumed latency. This difference completely alters the semantics of a program. Consider the fragment of a NUAL program shown in Figure 2a. 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 2b 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 2a has been split into two operations in Figure 2b. The Phase1 operation consists of the read micro-operations and the actual computation micro-operation. The Phase2 operation consists of the write micro-operation 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 2b, a Phase2 operation is interpreted as being 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 2. (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 2b, 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 2b 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 issued 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 3a). 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 3b). 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), the Phase2 operation is issued. Either immediately or eventually it executes, i.e., performs the copy from the anonymous register to the architectural register, and is retired.

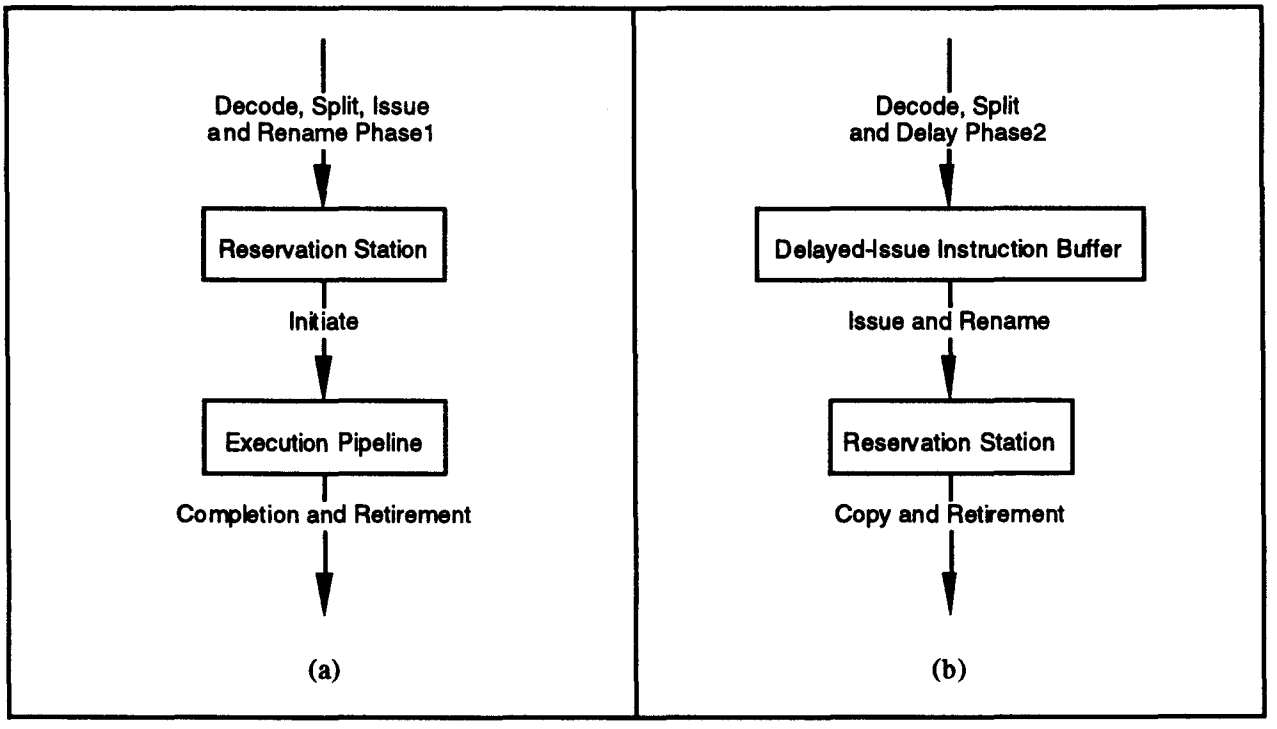

Figure 3: (a) Execution phases for a Phase1 operation. (b) Execution phases for a Phase2 operation

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

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 [19]. Presumably, the value specified must be the maximum 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 [20].)

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 [1-3].

It is worth emphasizing that the assumed latency need not be the same as the hardware latency (or even an estimate thereof) as long as some form of scoreboarding or dynamic scheduling is available. A few examples illustrate this point. Imagine that after scheduling code with full knowledge of the hardware latencies, one finds a large number of contiguous instructions

containing nothing but noops. It is perfectly correct to delete all those instructions while at the same time reducing the assumed latencies of all operations, whose execution spans those instructions, by the number of deleted instructions. (Doing so would require the ability to specify an assumed latency per operation.) At runtime, these noop cycles will be dynamically inserted by the instruction issue logic. There are two motivations for adjusting the assumed latencies in this manner. One is that the code size is reduced by eliminating the noop instructions. The second is that, if at some subsequent time this program is executed on a machine with shorter latencies, the noop cycles may not need to be dynamically inserted and the program's performance will benefit from these reduced latencies.

A second example is motivated by the possibility that latencies on a future machine might be longer than those on the current one. Assume a piece of code with adequate instruction-level parallelism such that the scheduled code has few noop instructions in it. Consider a particular operation whose result is first used well after it has been generated (assuming current latencies). This program is still correct if the assumed latency is increased beyond the current hardware latency to a value equal to the time interval between the start of this operation and its first use. The advantage so doing is that there will be no performance degradation on subsequent machines with longer latencies as long as the latencies do not exceed this assumed latency. This is the manner in which assumed latencies are used in Horizon [19].

Lastly, consider a loop with no loop-carried dependences. Over-estimating the hardware latencies contributes only to the startup penalty of the loop, whereas under-estimating them leads to stalling instruction issue on every iteration. For loops with large trip counts, it is preferable to specify assumed latencies that are large enough that they will rarely under-estimate the actual latencies that might be encountered across all members of this architectural family.

4 Dynamic scheduling techniques for NUAL programs

4.1 A machine model

The general machine model assumed in this paper is shown in Figure 4. 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 first-in-first-out **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 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 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 4 as the copyback unit. Phase1 operations access the architectural register file for their source operands and access the anonymous **delay register file** for writing the destination operand. Conversely, Phase2 operations access the delay register file for their source operand and the architectural register file for writing the destination operand.

---

[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.

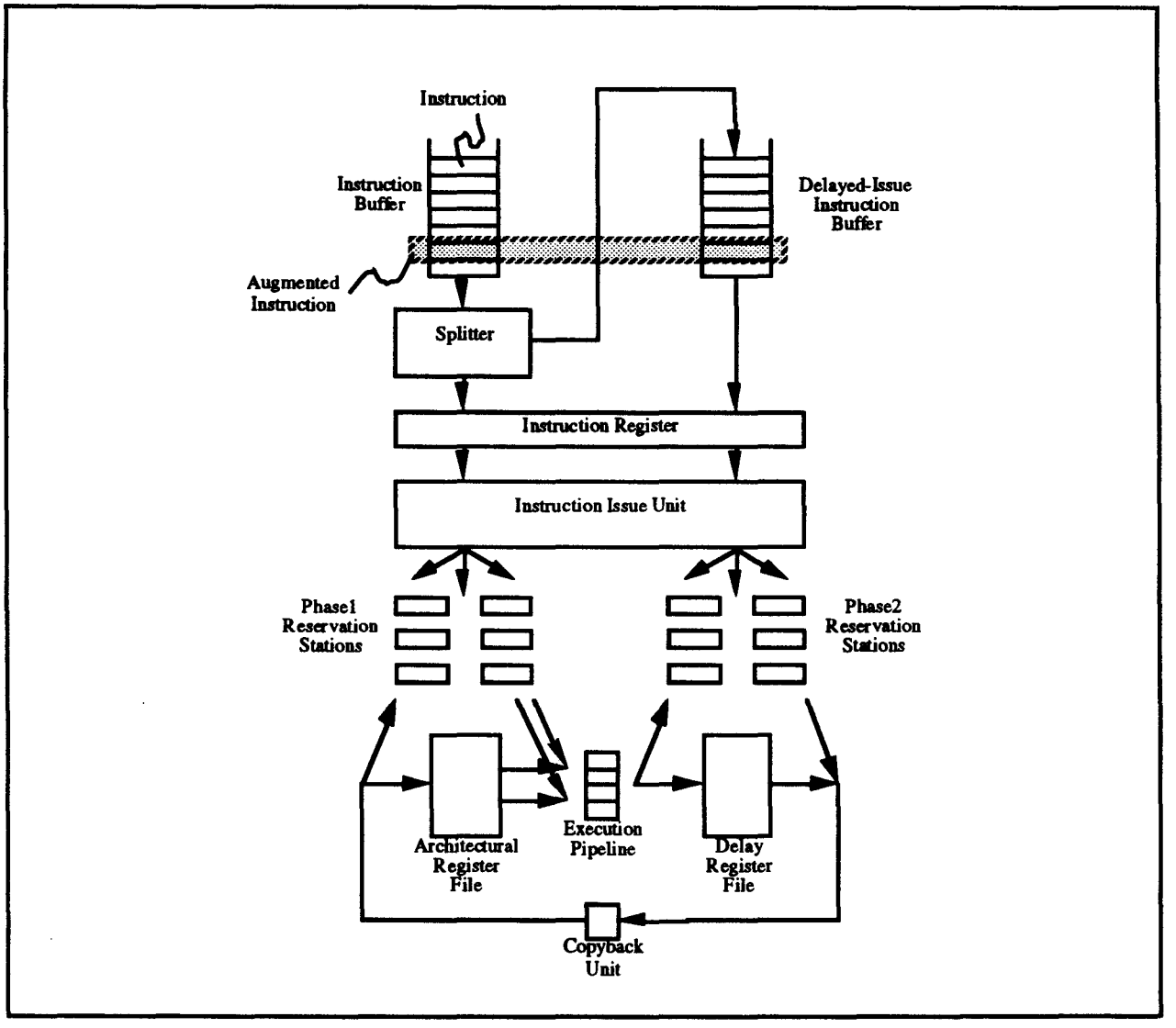

Figure 4. Processor organization for supporting split-issue with out-of-order execution.

Figure 4 displays a single architectural register file, a single execution pipeline, a single delay register file and a single copyback unit. In general, a processor might possess multiple architectural register files (e.g., integer and floating-point). There may be multiple functional units that access a given architectural register file. Furthermore, certain operations may access one architectural register file as the source and another one as the destination. Without loss of generality, we shall focus on a single architectural register file. We shall assume that there is a unique delay register file and copyback unit per functional unit. Thus, each delay register file is written to only by one specific functional unit and is read by a single copyback unit. In comparison to the architectural register file,

the delay register file can be implemented less expensively since it has but a single read port and a single write port.

4.2 Instruction issue strategies for NUAL programs

If we assume that all architectural register files have the same register access policy and, likewise, that all delay register files 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 architectural register file access policy (row) and delay register file 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 architectural register file, 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.

Three of these instruction issue policies turn out to be uninteresting. These are the ones in the first column corresponding to delay register file access policy B1 (SFSO) and architectural register file policies A1, A2 and B1. Since instruction issue stalls if the actual value of a Phase2 operation 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 when instruction issue is not stalled due to a Phase2 flow dependence, every Phase1 operation will find that its (architectural register file) source operands are available and that there are no outstanding writes against its (delay register file) destination. Therefore, it is unnecessary to do any dependence checking at all when accessing the architectural register file.

The instruction issue policy for which the architectural register file is not interlocked and the delay register file policy is B1 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

---

[1] We ignore the case in which the delay register file 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 architectural register file, too, would have no interlocks.

interlock hardware on the architectural register file which, in an ILP processor, may be expected to be highly multiported.

The other three policies in the first column degenerate to latency stalling and are not of interest as distinct policies. Also, if the architectural register file 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 architectural register file, which precludes A1 and A2.

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		
Architectural Register File Policy	**B1: SFSO**	**A2: RSSO**	**A1: RSRR**
Non-interlocked	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

This leaves seven policies of possible interest. Each of these policies may be combined with multiple instruction issue. Note that by this 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. 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. Other than the additional requirement of performing split-issue, the implementation issues are very similar to those for a multiple-issue superscalar processor [17].

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

4.3 General structure of the reservation stations for NUAL

Figure 5 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, second, the output section that pertains to the result operand (Figure 5a). 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 architectural register file and the output section is that for a delay register file. For a Phase2 operation, it is just the opposite. Figures 6b and 6c display the detailed structure of the input and output sections corresponding to an architectural register file and the delay register file, 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 architectural register file 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.

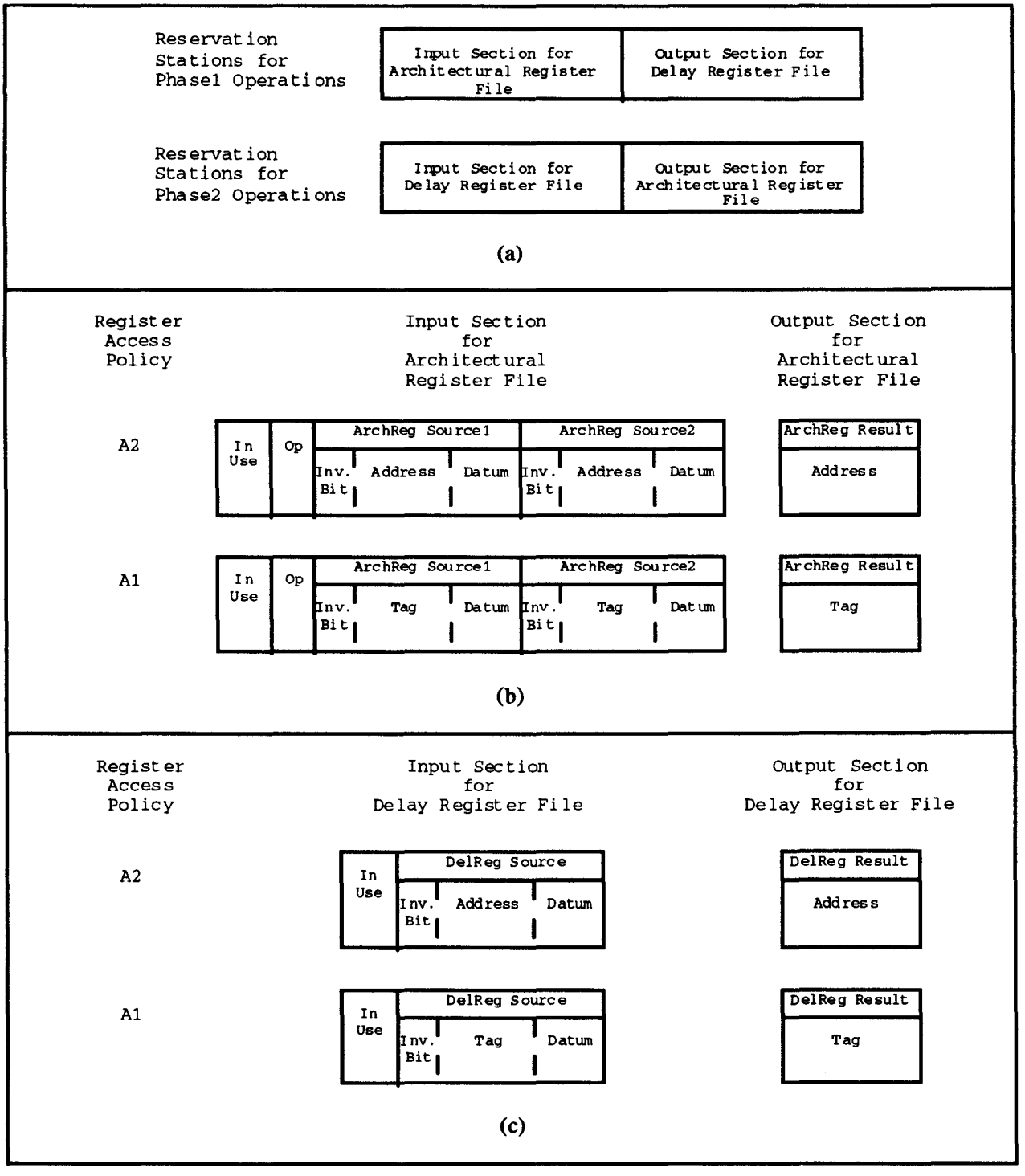

Figure 5. (a) The structure of reservation stations for Phase1 and Phase2 operations. (b) The detailed structure of the reservation station input and output sections that are associated with the architectural register files. (c) The detailed structure of the reservation station input and output sections that are associated with the delay register files.

In principle, the input and output sections associated with the delay register file are similar to those for the architectural register file, 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 5c 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. Generally, with out-of-order execution, multiple operations might use the result of a given operation. Consequently, a "linked list" of dependent operations must be created. This is implemented by broadcasting the result along with a tag and having every waiting operation perform an associative comparison between the broadcast tag and the tag that is being waited on. In the case of the delay register file, there is always exactly one dependent operation. So, instead of associative hardware, the destination register can directly point to the reservation station of the dependent operation and forward the result to it when available. Better yet, the delay register file and the Phase2 operation's reservation station can be combined into a single structure (Figure 6a). Effectively, a reservation station is allocated to a Phase2 operation at the same time that the delay register is allocated to the corresponding Phase1 operation, i.e., when the original operation is split and the Phase1 operation is issued.

This combination yields further simplifications. Figure 6a shows a delay register concatenated with a Phase2 operation's reservation station. The tag and datum fields in the reservation station can be eliminated since they replicate identical fields in the delay register. The "in use" bit of the reservation station can be eliminated since its state is implied by the joint state of the two invalid bits. The resulting simplified structure is shown in Figure 6b with the invalid bit of the reservation station renamed the copyback bit. The invalid bit is set when the Phase1 operation is issued and reset when it completes. The copyback bit is set when the Phase2 operation is issued and is reset when it completes. If either bit is set, the delay register cum reservation station is in use.

Once the delay registers have been combined with the Phase2 reservation stations, register access policy A1 becomes pointless. This policy is motivated by the desire to permit multiple tags to be associated with the same register, i.e., to have multiple, concurrently active reservation stations associated with the same delay register. This is precluded once the delay registers and the

reservation stations have been combined; A1 degenerates to A2. At this point, the tag is the same as the address of the delay register and may be eliminated (Figure 6c).

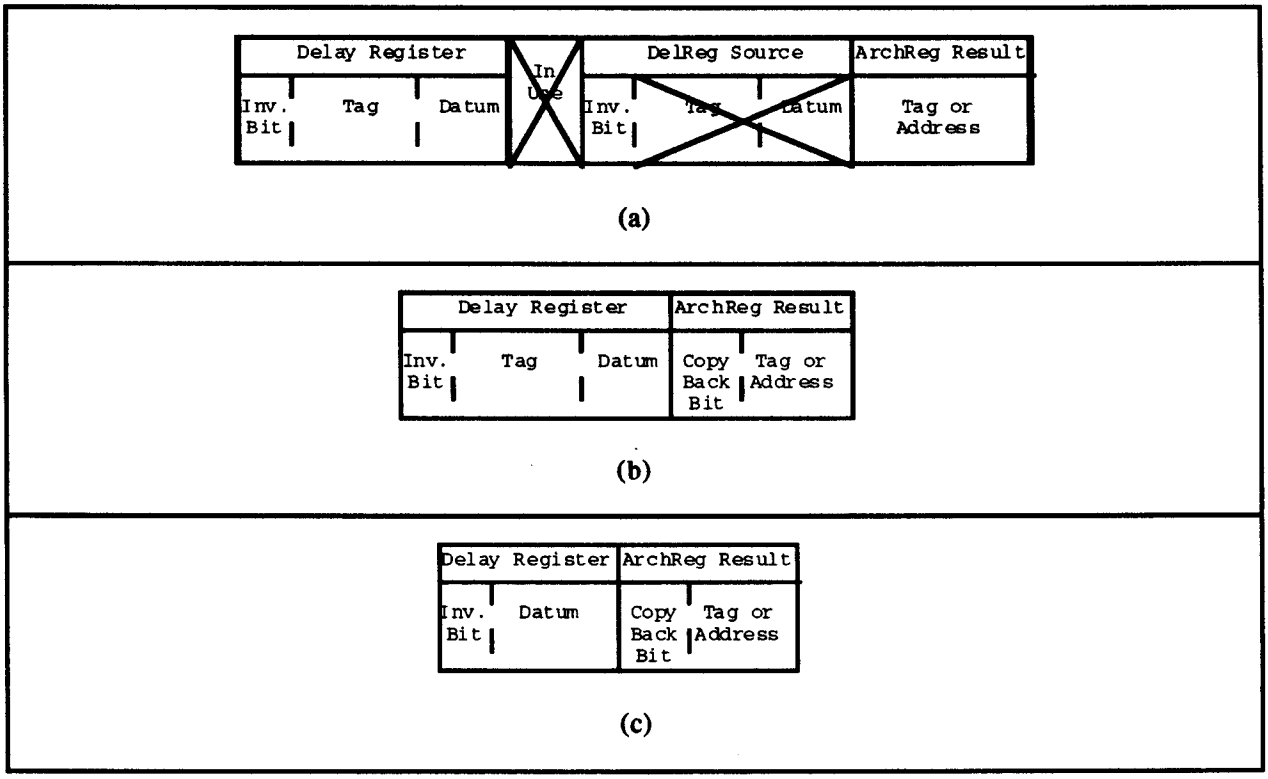

Figure 6. (a) The concatenation of the delay register with the reservation station for Phase2 operations. (b) The combined delay register and reservation station after optimizing away the redundant portion. (c) The combined delay register and reservation station after eliminating register renaming.

The Phase2 operation that is inserted into the delayed-issue buffer contains the addresses of the source delay register and the destination architectural register. In addition, by virtue of where the Phase2 operation is inserted into the delayed-issue buffer, the time of issue for this operation is specified implicitly. The final simplification is to combine the delayed-issue buffer with the delay register cum reservation station by noting that space is already provided in the latter for the tag or the architectural register address corresponding to the destination, and need not be replicated in the split-issue buffer. The delay register file address in the Phase2 operation need not be specified since, after combining, the delay register address is the same as the address that was used to access the delayed-issue buffer. For brevity, we shall refer to the combined delayed-issue buffer / delay register file / Phase2 reservation stations as the **delay buffer**, and each element of the delay buffer has the structure shown in Figure 6c.

The only loose end left is the specification of the correct time for issuing the Phase2 operation. This can be done by organizing the delay buffer 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 will 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, 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. Second, if access policy A1 is being employed for the architectural register file, 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. As a result, the copyback bit will be set at the time that the result is written into the delay buffer element. So, 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 architectural register file 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

---

[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.

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 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 as well, 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 [20]. 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.

The net outcome of this process of simplification is that the hardware for supporting the dynamic scheduling of NUAL programs, over and above that needed for UAL programs goes from three additional structures (the delayed-issue buffer, the delay register file and the Phase2 reservation stations) to just one relatively simple one per functional unit: the delay buffer. 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 architectural register file 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.

5 Discussion

In retrospect, the function and structure of the delay buffer are simple and obvious. The delay buffer causes results to be written to the architectural register file as soon as they are available just as long as it is not sooner than the assumed latency. The invalid bit and the datum field serve the function of buffering results that are computed too soon and the copyback bit records that the result is overdue and need not be buffered once available. As a side benefit, the address of the delay buffer element allocated to each Phase1-Phase2 operation pair serves as the tag if register renaming is employed in the architectural register file.

The complexity of the architectural register file and the Phase1 reservation stations, as a function of the register access policy, is unaffected by the fact that we are considering NUAL rather than UAL programs. For each architectural register access policy, the delay buffer guarantees that the result will not be written back any sooner than the NUAL semantics specify. In practical terms, the most interesting instruction issue policy is latency stalling since the architectural register files require no interlock hardware at all. Note that for UAL programs, this degenerates to sequential execution, a totally uninteresting option if ILP is the objective. NUAL, on the other hand, makes this a viable option and, in fact, the preferred one.

The objective of this paper was to establish that VLIW and dynamic scheduling are not contradictory concepts. This we have done. In fact, latency stalling is a simple and eminently practical scheme for use with VLIW and the delay buffer in conjunction with split-issue can support arbitrarily sophisticated access policies for the architectural register file. Though 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. On the other hand, 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.

VLIW has two main, unique attributes: MultiOp and NUAL. Let us examine the value of each one in the context of the following assumptions: that ILP is an essential feature for high-performance

microprocessors and that a significant and increasing fraction of a microprocessor's workload has levels of ILP that permit and make desirable the issuance of well over 4 operations per cycle. This is our belief and although there are those that would dispute one or both assertions, we do not have the luxury of debating these points in this paper. What are our options in light of these assumptions?

Superscalar is almost surely unrealistic if we wish to issue six, eight or ten operations per cycle. 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. If one attempts to issue N (UniOp) instructions per cycle, one needs 5N(N-1)/2 comparators to check for all possible flow, output and anti-dependences. Once some subset of the candidates have been issued, the remaining unissued instructions must be compacted and additional ones must be fetched in place of those that were issued. The hardware required is identical to the dispatch stack [18]. Apart from the cost in terms of logic, this can dilate the cycle time or require the insertion of an additional stage in the instruction issue pipeline. There is currently no existence proof of the ability to design a commercially successful product that issues eight or more instructions per cycle.

In contrast, the issuance of a single MultiOp instruction with N operations incurs no complexity since the compiler guarantees their independence. If N operations from multiple MultiOp instructions are to be issued simultaneously, then dependence checking logic is required once again. However, since it still is unnecessary to compare operations that are from the same instruction, the number of comparators is reduced by a factor of at most two. We conclude from this discussion that MultiOp is an essential feature of a highly parallel ILP processor. How valuable is NUAL?

NUAL architectures require latency-cognizant compilation and permit the use of the relatively simple latency stalling scheme. UAL architectures could just as well employ latency-cognizant compilers. This would minimize the frequency with which interlocks are invoked (if the noop cycles in the schedule are retained as noop instructions in the code). In this case the simplest access policy, B1, may be used for the architectural register file since interlocks are rarely expected to occur. In this form, the code is very much like NUAL code. Typically, code generation for UAL architectures eliminates the noop instructions, relying upon the dynamic scheduling capability of the hardware to re-insert them dynamically. In this case it is questionable whether policy B1 is adequate; however, we shall assume that it is. In the context of this paper, having ignored the topic of predicated execution, there is little to be said in favor of either UAL or NUAL without the benefit of a detailed analysis of the circuit design and timing of the interlock logic in both cases.

Predicated execution is a technique which enables the elimination of conditional branches in code and the overlapped scheduling of computations which previously contained conditional branches [21, 15, 22, 23]. Although originally developed in the context of NUAL (VLIW) architectures, in principle predicated execution is applicable to UAL architectures as well. However, interlocking or any form of dynamic scheduling lead to difficulties. The problem is that predicated execution squashes an operation if its predicate is false and allows it to execute normally otherwise. Thus, the value of the predicate determines whether an operation modifies its destination register. All of the register access policies available to a UAL architecture require knowledge of which register is modified by each operation. At the time of issuing an operation, if the value of the predicate is unknown, instruction issue must be stalled. Alternatively, with policies A2 and B1, instruction issue can proceed using the conservative assumption that the predicate is true and that the destination register, therefore, has a pending write to it. Subsequently, if the predicate turns out to be false, the invalid bit for the destination register must be reset, but the resulting temporary and spurious flow and output dependences will have compromised performance. Policy A1, register renaming, cannot be used since subsequent operations will be waiting for the tag corresponding to an operation that will never execute. Latency stalling, which applies to NUAL architectures, does not depend upon knowledge of which operation is modifying which register. It merely focuses on the discrepancy between the actual and the assumed latencies. Consequently, the presence of predicated execution has no impact upon it.

6 Conclusions

The commonly held view has been that object code compatibility is impossible with VLIW processors. However, VLIW is the only practical option if the goal is to exploit instruction-level parallelism at the level of issuing six or more operations per cycle. So, it is important to develop dynamic scheduling techniques for VLIW processors.

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 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.

Having laid out the concepts and framework for the dynamic scheduling of VLIW processors, the next steps for this research activity are to understand the hardware costs and performance

characteristics of the various instruction issue policies. Of particular interest are those that utilize the delay buffer, especially latency stalling.

Acknowledgments

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

References

1. A. E. Charlesworth. An approach to scientific array processing: the architectural design of the AP-120B/FPS-164 Family. Computer 14, 9 (1981), 18-27.
2. R. P. Colwell, R. P. Nix, J. J. O'Donnell, D. B. Papworth and P. K. Rodman. A VLIW architecture for a trace scheduling compiler. IEEE Transactions on Computers C-37, 8 (August 1988), 967-979.
3. G. R. Beck, D. W. L. Yen and T. L. Anderson. The Cydra 5 mini-supercomputer: architecture and implementation. The Journal of Supercomputing 7, 1/2 (1993), 143-180.
4. J. E. Thornton. Parallel operation in the Control Data 6600. Proc. AFIPS Fall Joint Computer Conference (1964), 33-40.
5. D. W. Anderson, F. J. Sparacio and R. M. Tomasulo. The System/360 Model 91: machine philosophy and instruction handling. IBM Journal of Research and Development 11, 1 (January 1967), 8-24.
6. J. E. Smith, G. E. Dermer, B. D. Vanderwarn, S. D. Klinger, C. M. Roszewski, D. L. Fowler, K. R. Scidmore and J. P. Laudon. The ZS-1 central processor. Proc. Second International Conference on Architectural Support for Programming Languages and Operating Systems (Palo Alto, California, October 1987), 199-204.
7. R. D. Groves and R. Oehler. An IBM second generation RISC processor architecture. Proc. 1989 IEEE International Conference on Computer Design: VLSI in Computers and Processors (October 1989), 134-137.
8. K. Diefendorff and M. Allen. Organization of the Motorola 88110 superscalar RISC microprocessor. IEEE Micro 12, 2 (April 1992), 40-63.
9. E. DeLano, W. Walker, J. Yetter and M. Forsyth. A high speed superscalar PA-RISC processor. Proc. COMPCON '92 (February 1992), 116-121.

10. J. E. Thornton. Design of a Computer - The Control Data 6600. (Scott, Foresman and Co., Glenview, Illinois, 1970).

11. R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. IBM Journal of Research and Development 11, 1 (January 1967), 25-33.

12. J. E. Smith and A. R. Pleszkun. Implementing precise interrupts in pipelined processors. IEEE Transactions on Computers C-37, 5 (May 1988), 562-573.

13. G. S. Sohi and S. Vajapayem. Instruction issue logic for high-performance, interruptable pipelined processors. Proc. 14th Annual Symposium on Computer Architecture (Pittsburgh, Pennsylvania, June 1987), 27-36.

14. W. W. Hwu and Y. N. Patt. Checkpoint repair for out-of-order execution machines. IEEE Transactions on Computers C-36, 12 (December 1987), 1496-1514.

15. B. R. Rau, D. W. L. Yen, W. Yen and R. A. Towle. The Cydra 5 departmental supercomputer: design philosophies, decisions and trade-offs. Computer 22, 1 (January 1989), 12-35.

16. D. J. Kuck. The Sructure of Computers and Computations. (John Wiley & Sons, New York, 1978).

17. M. Johnson. Superscalar Microprocessor Design. (Prentice-Hall, Englewood Cliffs, New Jersey, 1991).

18. R. D. Acosta, J. Kjelstrup and H. C. Torng. An instruction issuing approach to enhancing performance in multiple function unit processors. IEEE Transactions on Computers C-35, 9 (September 1986), 815-828.

19. M. R. Thistle and B. J. Smith. A processor architecture for Horizon. Proc. Supercomputing '88 (Orlando, Florida, November 1988), 35-41.

20. B. R. Rau, M. S. Schlansker and D. W. L. Yen. The Cydra 5 stride-insensitive memory system. Proc. 1989 International Conference on Parallel Processing (August 1989), 242-246.

21. J. C. Dehnert, P. Y.-T. Hsu and J. P. Bratt. Overlapped loop support in the Cydra 5. Proc. Third International Conference on Architectural Support for Programming Languages and Operating Systems (Boston, Mass., April 1989), 26-38.

22. 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. Proc. 25th Annual International Symposium on Microarchitecture (1992), 45-54.

23. J. C. Dehnert and R. A. Towle. Compiling for the Cydra 5. The Journal of Supercomputing 7, 1/2 (1993), 181-228.