HPL-PD Architecture Specification: Version 1.1

Vinod Kathail, Michael S. Schlansker, B. Ramakrishna Rau
Compiler and Architecture Research
HPL-93-80 (R.1)
February, 2000 (Revised)

{kathail, schlansk, rau}@hpl.hp.com

instruction-level parallelism, parametric architecture, EPIC, VLIW, superscalar, speculative execution, predicated execution, programmatic cache control, run-time memory disambiguation, branch architecture

HPL-PD is a parametric processor architecture conceived for research in instruction-level parallelism (ILP). Its main purpose is to serve as a vehicle to investigate processor architectures having significant parallelism and to investigate the compiler technology needed to effectively exploit such architectures. The architecture is parametric in that it admits machines of different composition and scale, especially with respect to the nature and amount of parallelism offered. The architecture admits EPIC, VLIW and superscalar implementations so as to provide a basis for understanding the merits and demerits of these different styles of implementation. This report describes those parts of the architecture that are common to all machines in the family. It introduces the basic concepts such as the structure of an instruction, instruction execution semantics, the types of register files, etc. and describes the semantics of the operation repertoire.

This is a revised version of “HPL PlayDoh Architecture Specification: Version 1.0”, Technical Report HPL-93-80, February, 1994.

1 Introduction

HPL-PD[1] is a parametric processor architecture conceived for research in instruction-level parallelism (ILP). HPL-PD defines a new philosophy of ILP computing, called Explicitly Parallel Instruction Computing (EPIC) [2-4], which represents an evolution of the VLIW architecture[2]. HPL-PD's main purpose is to serve as a vehicle to investigate processor architectures having significant parallelism and to investigate the compiler technology needed to effectively exploit such architectures.

When we released the first version of the architecture, we envisioned that a broad segment of the ILP research community would start using the architecture in their research, thus providing a single base to judge the merits of new architectural features and compiling technology. Since the publication of the first version, HP Labs and our university partners have been using the HPL-PD architecture for our own research. We, in conjunction with the University of Illinois' IMPACT project and NYU's ReaCT-ILP project, developed a set of tools, *e.g.*, a compiler and a simulator, for the HPL-PD architecture. In 1998, this compiler and simulation infrastructure, christened Trimaran[3], was released to universities to encourage wide-spread research in EPIC architecture and compiler technology.

HPL-PD is a parametric architecture in that it admits machines of different composition and scale, especially with respect to the amount of parallelism offered. This is to enable experiments that answer one of the important question in ILP research—how much parallelism is available in programs and how best to exploit it. Although the architecture started out as a VLIW architecture and evolved into the EPIC architecture because of our research interests, it takes no position as to the style of implementation. It admits both EPIC (which includes VLIW as a subset[4]) and superscalar implementations so as to provide a basis for understanding the merits and demerits of the two styles of implementation. However, certain of HPL-PD's features are better motivated in the EPIC context.

We emphasize that the main purpose of the HPL-PD architecture is to serve as a research vehicle. Its current definition represents a collection of ideas whose merits we wish to explore and should not be taken as frozen in concrete. We expect the architecture to evolve. Its evolution will be engineered to support the investigation of important research questions in instruction-level parallelism. New features will be added as and when required, and subsets of interest will be defined for specific studies. We will ensure that the evolution takes place in a controlled manner and that it is coordinated with our collaborators.

---

[1] This document is a revision of an earlier technical report published in 1994 [1]. In the intervening period, the name EPIC was coined to describe the style of processor architecture that was introduced in the earlier technical report. This document acknowledges the new name. Also, several new operations have been added to the operation repertoire.

[2] The Intel IA-64 instruction-set architecture [5] represents the first commercial instance of the EPIC style of architecture.

[3] More information about Trimaran can be found at `www.trimaran.org`.

[4] In the rest of this document, EPIC will be understood, implicitly, to include VLIW.

This report describes those parts of the architecture that are common to all machines in the family. It introduces basic concepts such as the structure of an instruction, instruction execution semantics, the types of register files, etc. and describes the semantics of the operation repertoire. The report is organized as follows. The rest of this section highlights the changes from the earlier version. Section 2 gives an overview of the architecture and discusses its parametric nature. It also discusses the EPIC and superscalar scheduling models. Section 3 discusses the instruction execution semantics including the semantics of multiple operations per instruction, speculative execution and predicated execution. Section 4 describes the data types and the types of the register files supported by the architecture. Section 5 introduces the format used for describing operations. The remaining sections except the last describe the operation repertoire provided by the architecture. Sections 6 and 7 describe integer and floating-point computation operations, respectively. Section 8 describes operations for data type conversion and for moving data between register files. Section 9 describes compare operations including compare-to-predicate operations. Section 10 describes the memory system and the load/store operation repertoire. Section 11 describes the branch architecture. The last section contains some concluding remarks.

1.1 Architectural enhancements / changes in Version 1.1

This sections highlights the new features as well the changes from the earlier version for readers who are already familiar with the earlier version of the architecture. The most visible change, of course, is the new name for the architecture. For various non-technical reasons, the architecture is now called HPL-PD. In addition, Sections 2.2, 3.1 and 3.2 have been modified substantially. We have also added a number of new references in order to bring them up-to-date.

Architectural enhancements in Version 1.1 are summarized below. These include several new integer, data conversion and move operations to allow greater experimentation. In addition, the architecture now provides a comprehensive set of load/store operations to save/restore and spill / unspill registers, something that was missing from the first version. Please note that we have added several new tables to accommodate the new operations, and thus, table numbers from the earlier version are not a good guide to look up the description of various operations in this version.

1. *Integer operations:* The architecture now includes five new integer operations -- ABS, MIN, MINL, MAX and MAXL. See Table 2 for their detailed specifications.
2. *Conversion operations:* Version 1.0 didn't include conversion operations between unsigned integers and floating point values. Thus, four new operations -- CONVLWS, CONVLWD, CONVLSW and CONVLDW, have been added for this purpose. Table 4 contains more details about these operations.
3. *Move operations:* The new version includes a number of new move operations. These include the following:

   MOVEGBP, MOVEB, MOVEGCM, which are described in Table 5;

   separate single and double precision moves between floating point registers, called MOVEFS and MOVEFD, which are described in Table 6;

   MOVEGBGT, MOVEGBFT, MOVEGBPT, MOVEGBBT, which are described in Table 6; and

   PRED_CLEAR_ALL, PRED_CLEAR_ALL_STATIC, PRED_CLEAR_ALL_ROTATING, which are described in Table 7.

   Some of these operations are used in generating efficient code to spill or save/restore

predicate registers and speculative tag bits (see Section 10.7).

4. *Load/store operations to spill and save/restore registers:* Section 10.7 describes these operations as well as their typical usage. The new operations are: SAVE, RESTORE, FSAVE, FRESTORE, BSAVE and BRESTORE.

2 Overview of the architecture

The HPL-PD opcode repertoire, at its core, is similar to that of a RISC-like load/store architecture, with standard integer, floating point (including fused multiply-add type of operations) and memory operations. In addition, it provides a number of advanced features, which we believe are important for enhancing and exploiting parallelism in programs.

The first generation of VLIW machines, such as the Cydra 5 [6] and the Multiflow TRACE [7], proved themselves as very successful in exploiting the parallelism present in applications rich in counted loops (DO-loops). The results on "scalar" code were somewhat mixed. Apart from the use of speculative execution, there was very little ability to accelerate the execution of codes in which the limiting factor was not a shortage of execution resources but, rather, the critical path through the computation. There were few architectural features to assist in reducing the length of this critical path and, in fact, the absence of data caches served to exaggerate it. The primary objective with HPL-PD has been to provide architectural features that permit the relaxation of ordering constraints between operations on the critical path, thereby shortening the critical path and facilitating schedules that exhibit higher levels of ILP than are generally expected for such applications.

Many of the architectural features in HPL-PD have their microarchitectural counterparts in conventional or superscalar processors. In much the same way that the first generation of VLIW machines made the execution resources and latencies architecturally visible, HPL-PD makes these heretofore invisible capabilities visible to, and controllable by, the compiler. A case in point is the cache memory hierarchy that has generally been controlled entirely by hardware and transparent to the compiler. A consequence of this is that certain applications that have poor data locality suffer massive performance degradation because the hardware continues to use the same default cache management policy. Most high performance processors aimed at the scientific processing community have, as a consequence, avoided the use of a data cache. Unfortunately, sequential performance suffers in the absence of a data cache. HPL-PD addresses this dilemma by providing a cache hierarchy whose default behavior is to function in the conventional manner, but with the additional ability for the program to explicitly instruct the hardware on how to manage data which behaves anomalously.

In the first part of this section, we preview some of the more advanced features of the architecture. Then, we describe the parametric nature of the architecture.

2.1 Advanced features of the architecture

The advanced features of the architecture can be grouped into the following major areas. Many of these features are well-known in the ILP research community. The Cydra 5 [6], the Multiflow Trace [7] and the HP PA-RISC [8] architectures included several of these features. Some others have been proposed in the literature (see, for example, [9-15]).

1. *Speculative execution*: Speculative execution is an important technique for enhancing parallelism in a program. It is used to break certain types of dependences between

operations. Section 3.4 describes speculative execution in more detail and identifies two forms of speculation: control speculation used to move operations above branches and data speculation used in the run-time disambiguation mechanism discussed later.

The architecture supports speculative execution of most operations; exceptions are stores and branches. To correctly handle exceptions generated by speculative operations, the architecture provides speculative and non-speculative versions of operations and provides speculative tag bits on registers (see Section 4).

2. *Predicated execution*: Predicated or guarded execution refers to the conditional execution of operations based on a boolean-valued source operand, called a predicate. It is a way to enforce the requirements of program control-flow, which is different from that provided by branch operations. Predicated execution is often an efficient method to handle conditional branches and provides much more freedom in code motion than possible otherwise. Section 3.3 describes predicated execution in more detail.

   To support predicated execution, the architecture provides 1-bit predicate register files and a rich set of compare-to-predicate operations which set predicate registers. In addition, most operations have a predicate input to conditionally nullify their execution (see Section 3.3 for exceptions).

3. *Memory system and load/store operations*: The architecture supports a hierarchical memory system consisting of first-level cache(s), a data prefetch buffer, second-level cache(s) and main memory. The main architectural features are:

   - Compiler control of the memory hierarchy: The architecture provides latency and cache-control modifiers with load/store operations, which permit a compiler to explicitly control the placement of data in the memory hierarchy. The default, in the absence of the use of these directives, is the conventional hardware management. The architecture also supports prefetching of data to any level in the memory hierarchy.
   - Run-time disambiguation mechanism: "Maybe" dependences between store and load operations are a limiting factor in exploiting parallelism. The run-time disambiguation mechanism is used to break these dependences. It permits a load and dependent operations to be issued before potentially aliasing stores even in the absence of conclusive compile-time aliasing information. The mechanism consists of three related families of operations, called data speculative loads (LDS), data verify loads (LDV) and data verify branches (BRDV).

   In addition, the memory operations in an instruction are executed in left-to-right prioritized order (see Section 3.2). This allows stores and dependent memory operations to be issued in the same cycle. Also, the architecture provides post-increment load/store operations similar to the ones in the HP PA-RISC architecture. For hardware efficiency, the architecture doesn't provide pre-increment load/store operations. Section 10 describes memory operations in more detail.

4. *Branch architecture*: The branch mechanism described in this report is a preliminary attempt to address the efficient implementation of branches in ILP machines. It permits different pieces of the information related to a branch to be specified as soon as they become available in the hope that the information can be used to reduce the adverse effect of the branch, *e.g.*, by prefetching instructions from the potential branch target.

   Prepare-to-branch operations are used to specify the target address and the static prediction for a branch ahead of the branch point, typically initiating instruction prefetch. The architecture provides a separate type of register file, called a branch target register file, to

store information about branches that have been prepared. Compare-to-predicate operations are used to compute branch conditions, which are stored in predicate registers. Branch operations test predicates and perform the actual transfer of control. The operation repertoire includes special branch operations to support software pipelining of loops. Section 11 describes branch architecture in more detail.

5. *Unusual semantics for simultaneous writes to registers:* The architecture permits multiple operations to write into a register simultaneously provided they all write the *same value*. In this case, the result stored in the register is simply the value being written. In the case of predicate registers, this atypical semantics is useful for efficient evaluation of boolean reductions (see below). In the case of other types of registers, the utility of this semantics is a research topic. See Section 3.2.
6. *Support for efficient boolean reduction*: We are interested in a class of parallelization techniques, which we generically call height-reduction of control dependences. In many cases, application of these techniques require a fast way to compute AND or OR of several boolean values to derive either a branch condition or a predicate to guard the execution of operations. The architecture provides a set of compare-to-predicate operations (OR and AND classes in Section 9.3) for efficient evaluation of boolean reductions.
7. *Register files*: The architecture supports rotating registers in integer, floating-point and predicate register files in order to generate "tight" code for software-pipelined loops. See Section 4.

2.2 Parametric nature of the HPL-PD meta-architecture

The EPIC philosophy is that it is the compiler, not the hardware, that is responsible for orchestrating the ILP of an executing program [4, 3]. The code for an EPIC processor reflects an explicit plan for how the program will be executed. This plan is created statically, i.e., at compile-time. It specifies when each operation will be executed, using which functional units, and with which registers as its operands. The EPIC compiler designs this plan, with full knowledge of the processor. The plan is communicated, via an instruction set architecture that can represent parallelism explicitly, to hardware which then executes the specified plan. The existence in the code of this explicit plan permits the EPIC processor to have relatively simple hardware despite high levels of ILP.

Static scheduling is necessary for superscalar processors as well, if the best performance is to be achieved at run-time. The definition of a good compile-time scheduling philosophy for superscalar processors is still an open problem. One strategy is to use the EPIC scheduling philosophy, an approach that has been adopted by the IMPACT project. After scheduling and register allocation, each parallel instruction is emitted as a series of sequential instructions. The superscalar hardware is expected to rediscover the parallelism at run-time and perform dynamic scheduling to achieve the intended ILP.

An EPIC code generator (the scheduler and register allocator) is responsible not just for the performance of the code, but for its correctness as well. Accordingly, it requires the following detailed information about the target EPIC processor [16]:

1. The register file structure. This includes the number of register files and, for each one, the number of registers in it and their bit width.
2. The operation repertoire. An *operation* consists of an opcode and a register tuple—one register each per operand. We treat any addressing modes supported by the target machine

as part of the opcode rather than as part of an operand. An operand is either a literal or a register in the target machine. For uniformity, we model a literal as the contents of a read-only "literal register". Multiple instances of an operation correspond to the presence in the machine of multiple functional units that can perform that operation. The operation repertoire specifies the opcode repertoire and, for each opcode, the sets of register files that it can access. Implicitly, and in a manner that is more directly useful to a compiler, this specifies the connectivity between the functional units and the register files.

3. Explicitly scheduled resources. This is the set of resources (functional units, buses, instruction fields, etc.) that the compiler must manage to ensure that two operations do not attempt to use the same resource at the same time.
4. The resource usage behavior of each operation. The compiler must use this information to ensure that the issue times of any two operations that use the same resource is such, relative to each other, that they will not end up using that resource simultaneously. This also determines which sets of operations can be issued simultaneously on this processor.
5. Latency descriptors. Every operation has a latency descriptor that specifies when, relative to the time that the operation is issued, each source operand is read and each destination operand is written.

HPL-PD is a meta-architecture, *i.e.*, a parametric processor architecture that encompasses a space of machines each of which has a different amount of ILP and a different instruction set architecture (ISA)[5]. Consequently, HPL-PD cannot be specific regarding all five types of information listed above. It only describes the first two, and in a manner that supersets the corresponding information for any specific ISA. In other words, HPL-PD (*i.e.*, this document) only describes the types of register files and the operation repertoire supported by the meta-architecture. Furthermore, the types of register files and the operation repertoire of any particular processor will be a subset, perhaps a proper subset, of what is defined in this document. Section 4 describes the types of register files supported by the architecture. Sections 6 through 11 describe HPL-PD's operation repertoire.

Architectures within the HPL-PD space consist of a set of register files, a set of functional units connected to register files, and a hierarchical memory system. A specific machine can have one or more of each type of register file except for the control register file[6]; there is exactly one control register file in each machine. Likewise, a specific machine can have one or more instances of each operation in HPL-PD's operation repertoire. Multiple instances of an operation means that multiple instances of that operation can be issued in parallel, *i.e.*, there are multiple functional units in the machine that can perform this operation.

All five types of information listed above, regarding the specific target processor, must be supplied to the compiler. A *machine-description database (mdes)* specifies this information to the compiler. The register file and operation repertoire information supplied in an mdes must be consistent with what is specified in this document. In collaboration with the University of

---

[5] Although HPL-PD is, strictly speaking, a meta-architecture, for convenience we shall refer to it as an architecture.

[6] We call a machine with one register file of each type a *single-cluster* machine, and a machine with more than one register file of at least one type a *multi-cluster* machine. Our focus has been on single-cluster machines; thus, it is possible that certain issues about multiple-cluster machines have not been properly addressed in this report.

Illinois' IMPACT project, we have developed an approach to describe an EPIC machine to a compiler [17, 18]. It was designed with the EPIC scheduling philosophy in mind and is based on the approach used by Cydrome in its compiler. The approach is supported by the following facilities:

1. A high-level machine description language: This is a human-readable version of the five types of information listed above.
2. A low-level machine description language: It specifies the same information as the high level machine description language but in a form that is suitable as an input to a compiler.
3. A macro processor and translator from the high-level machine description to the low-level machine description.
4. The *mdes Query System (mQS)*: This consists of the internal mdes data structures that hold the above information, and an associated query interface. The query interface is a collection of interface functions that a compiler can call to get the requisite information about the target machine and to manage a resource-usage map during scheduling.

A more detailed discussion of these topics is beyond the scope of this report. Interested readers should refer to the detailed description of the structure of the machine description database and the underlying theory presented in [16]. This paper describes an incremental model of code generation for EPIC processors that provides an efficient way to generate high quality code. The paper also relates this process to the structure of the machine description database that is queried by the code generator for the information that it needs about the target processor. A companion document [18] discusses an implementation of these ideas and techniques in Elcor, which is our compiler research infrastructure.

Elcor is a part of the Trimaran infrastructure. Trimaran, too, is designed such that it is parametric in nature. It reads the architectural parameters for a machine from its mdes. Thus, the various components of Trimaran will work either with all machines in the HPL-PD space or with a well-defined subclass, *e.g.*, EPIC machines.

The information discussed above is adequate for experiments that concern themselves with scheduling, register allocation and performance as a function of the parallelism and cost of the processor's datapath. For such purposes, the details of the instruction format are unimportant. However, if the issues under investigation are code size, scheduling that is sensitive to code size, instruction cache performance or instruction unit complexity and cost, then detailed knowledge of the instruction format is essential.

The components of the compiler infrastructure that depend upon this information are the assembler, the disassembler and the debugger. In keeping with the spirit of the rest of the infrastructure, it is desirable that they, too, be parametric and machine description driven. The nature of the description of the instruction format depends on the nature of the instruction format, for which there are many alternatives [5, 4]. The class of instruction formats that we have selected for our EPIC research is the class of what we term multi-template instruction formats [19]. We have also defined a strategy for dealing with instruction formats in a parametric fashion [20]. It consists of the following components:

1. A meta-grammar and meta-syntax for the class of multi-template instruction formats. The syntax of the instructions (i.e., the instruction format) for any given processor is a sentence of this meta-grammar.
2. An internal data structure, the Instruction Format Tree (IF-tree), for representing the description of an instruction format within the mdes.

3. A set of mQS interface functions that are needed during assembly and disassembly.

Our implementation of this strategy in Elcor is partially complete. When finished, it will be similar to the one described above, that is, there will be a high-level language for describing a multi-template instruction format, a low-level machine-readable counterpart, a translator from the high-level to the low-level description, and extensions to the internal mdes data structures and the query interface.

Even for experiments that do not require knowledge of the detailed instruction format, the compiler must still be able to specify branch addresses for the simulator's benefit. In such cases, one can assume an abstract machine whose program counter counts in units of an operation, *i.e.*, an operation's address is the number of operations preceding it in the program text, as laid out in memory. A point to note is that branch addresses must point to instruction boundaries; it is illegal to branch into the middle of an instruction.

3 Execution semantics

This section describes the basics of instruction execution. Sections 3.1 and 3.2 describe the logical structure of an instruction and the execution semantics of an instruction, respectively. Section 3.3 describes predicated execution, and Section 3.4 covers speculative execution. The last section contains brief comments about exceptions and exception handling in the case of normal execution of operations.

3.1 Logical instruction structure

In this section, we describe those aspects of the instruction structure that are necessary to understand HPL-PD. Since our discussion of HPL-PD is not, and cannot be, tied to any specific instruction format, we shall operate at what might, conceptually, be viewed as the assembly language level. Accordingly, we define a simple assembly language to facilitate our description of HPL-PD. This, or some equivalent syntax, would also be suitable as input to a simulator.

An *instruction* consists of a sequence of operations, each of which is terminated by a delimiter (a semi-colon). Each instruction is terminated by an end-of-instruction marker (the new line). The end-of-instruction marker can be viewed as a pseudo-operation, which performs actions typically associated with the end of an instruction, *e.g.*, advancing the program counter and advancing virtual time. For the convenience of a simulator, branch operations appear as the last operations within an instruction. There is no explicit NOP instruction in our assembly language. An empty instruction, *i.e.*, an instruction with no operations, serves the same purpose as a NOP.

In the superscalar version of the architecture, an instruction has zero operations (*i.e.*, a NOP) or one operation. In the EPIC version, the maximum number of operations in an instruction, the maximum number of operations of a specific type (*e.g.*, integer operations or memory operations), and the legal combinations in which they may appear are all specified by the mdes.

An *operation* corresponds to an elementary operation performed by a functional unit, *e.g.*, an add or a load. An operation specification consists of the following:

1. An opcode: Related opcodes are grouped into a family. Each family of opcodes has a major opcode and a set of modifiers. An opcode consists of a major opcode and a specific value for each of the modifiers associated with the family.
2. A list of sources: A source is either a register specifier or a literal. Conceptually, each register file in a machine has an unique name, and a register specifier consists of the name

of a register file and a register number within the register file. Literals are discussed below.

3. A list of destination register specifiers. Note that there are no destinations in the case of store operations and some of the branch operations.

The architecture permits integer literals in an operation. An operation that expects an integer value as an operand can specify either an integer register or an integer literal. Note that a machine can currently have literals of only one width: 32 bits[7]. Note, also, that there is no support for floating-point literals.

In this report, we use the following notations. An opcode is of the form a.b.c.d where a is the major opcode and b, c, d are modifiers. Modifiers are written in the same order in which they appear (top to bottom) in tables. In addition, the speculative version of an opcode is denoted by the modifier "E", which always appears at the end. The notation for writing register specifiers is as follows. Consider a register file named "R" containing n_S static registers and n_r rotating registers (see Section 4). Then, we specify a *static register* within the file as Ri where i is in the range $0 \le i \le n_S - 1$, and specify a rotating register as R[j] where j is in the range $0 \le i \le n_r - 1$. In some cases, we use a, b, c, etc., for register specifiers; their meaning will be clear from the context. The following example illustrates the notation used in writing assembly code.

Cycle	Instruction
1	GPR1 = ADD.W(GPR2, GPR3); GPR4 = SUB.W(GPR2, GPR3) if PR2;
2	– – – –
3	FPR2 = FADD.S(FPR1, FPR3); BRU(BTR2);

In this example, there are three instructions, each of which is on a separate line. The first instruction contains two operations; the second instruction is an empty instruction, which is denoted by ----; and the third instruction again contains two operations, one of which is an unconditional branch (BRU). The number in the cycle column indicates the cycle number in the program's virtual time when the corresponding instruction is issued. For example, the third instruction is issued at the third cycle in the program's virtual time. In some cases, when cycle numbers are not shown explicitly, they are assumed to be sequentially numbered.

To understand the notation used for writing operations, consider the first operation in the first instruction. ADD.W is its opcode, general-purpose registers GPR2 and GPR3 are its sources, and the general-purpose register GPR1 is its destination. If necessary, the guarding predicate for an operation is specified after the keyword "if"; for example, see the second operation in the first instruction. Each operation is terminated by a ";" delimiter.

3.2 Instruction execution

As mentioned in the introduction, we deliberately do not take a position about the style of implementation and admit both EPIC and superscalar implementations of HPL-PD. Broadly speaking, the characteristics of these two models of execution semantics are as follows.

---

[7] We recognize this as a major shortcoming of HPL-PD. It will be rectified in the next revision of this document, which will articulate a flexible way to define literals of various widths via the mdes.

The EPIC model [4, 3] is characterized by the following: MultiOp instructions and architecturally visible, non-unit assumed latencies (NUAL) for operations. EPIC code cannot be interpreted correctly without an understanding of these two features. The term MultiOp means that an instruction may explicitly indicate that multiple operations are to be issued in parallel. Within the EPIC model, we can identify two types of machines: "equals" (EQ) machines and "less-than-or-equals" (LEQ) machines. In an EQ machine, the latency(s) associated with an operation defines the exact time when the operation produces its output(s). In a LEQ machine, the latency(s) associated with an operation defines only the latest time by which the operation produces its output(s); the operation may produce its output(s) earlier than that. HPL-PD permits both types of machines. A detailed discussion of the merits of each type is beyond the scope of this report.

The superscalar model is characterized by the following. Instructions are UniOp, i.e., they have exactly one operation (which could be a NOP operation), operations have unit assumed latency (UAL), and dynamic scheduling is used to issue operations in parallel. That is not to say that static scheduling is not useful or that operations actually have unit latency. For performance reasons, compilers for superscalar machines, too, will schedule operations with full knowledge of the amount of parallelism in the machine and the actual latencies of operations. However, this knowledge is not necessary to correctly interpret code generated for a superscalar processor.

The number of instructions issued in a single cycle depends upon the implementation. In the typical EPIC version of the architecture, a single instruction is issued in each cycle. However, the architecture permits dynamically scheduled EPIC machines in which more than one instruction can be issued in a single cycle [21]. In the superscalar version, the number depends upon the capabilities of the dynamic scheduling hardware in a machine.

The execution semantics of a Multiop instruction is as follows. All operations in an instruction can be issued in parallel. Consequently, there can be no flow (*i.e.*, read-after-write) dependences between operations in a HPL-PD instruction with one notable exception. The architecture permits flow dependences between memory operations as long as they go from left-to-right in an instruction by ensuring that the following holds:

- The memory operations within an instruction are executed in an order that is consistent with their sequential left-to-right order of execution. This permits compilers to reduce the length of critical paths involving memory operations. For example, two store operations, one of which potentially depends upon the other, can be issued in the same instruction. Similarly, a store and a flow-dependent load can also be issued in the same instruction.

The semantics described above for a Multiop instruction permit two variations [4]. The first variation, called *Multiop-P* semantics requires that the correct execution is guaranteed only if all the operations in the instruction are issued simultaneously. The compiler can schedule code with the assurance that all operations in one instruction will be issued simultaneously. For instance, it can even schedule two mutually anti-dependent copy operations, which together implement an exchange copy, in the same instruction. Without this assurance, the exchange copy would have had to be implemented as three copy operations that require two cycles. Thus, the Multiop-P semantics permits admissible dependences between operations (*i.e.*, anti and output dependences) to be bi-directional across the instruction.

However, MultiOp-P semantics pose a problem with respect to the compatibility across a family of machines with differing amount of functional units. When code that was generated for a machine with a certain width (*i.e.*, number of functional units) has to be executed by a narrower machine, the narrow processor must necessarily issue the MultiOp instruction semi-sequentially,

one portion at a time. Unless care is taken, this will violate MultiOp-P semantics and lead to incorrect results. For instance, if the aforementioned copy operations are issued at different times, the intended exchange copy is not performed.

The other variation**,** called *MultiOp-S* semantics, simplify sequential execution by excluding bi-directional dependences across a MultiOp instruction. MultiOp-S instructions can still be issued in parallel, but they can also be issued sequentially from left to right. This permits the code compiled for a wide machine to be correctly executed on a narrow machine. On the other hand, the Multiop-S semantics excludes some of the benefits of the Multiop-P semantics. For example, an exchange-copy cannot be implemented as two copy operations in the same instruction. The compiler must ensure that admissible dependences between operations in a MultiOp instruction are only from left to right.

MultiOp-P and MultiOp-S bear similarities to EQ and LEQ, respectively. Both MultiOp-P and EQ guarantee that operations will not complete early in virtual time, whereas MultiOp-S and LEQ permit it. Although it need not necessary be the case, one would tend to use MultiOp-P in conjunction with EQ semantics, and to pair MultiOp-S with LEQ.

At present, the Elcor compiler in Trimaran supports only the Multiop-S semantics. It provides several versions of Multiop-S, which differ in the set of operations that are allowed to have intra-instruction dependences. For example, one version doesn't permit any intra-instruction dependences; another allows intra-instruction dependences only among memory operations.

The execution of an operation involves reading its inputs, computing the specified function and writing the results into the specified destination registers. The detailed timing constraints concerning the execution of an operation, *e.g.*, the latency of the operation, the times when inputs are sampled, the times when outputs are produced, are architectural parameters specified separately for each machine in its mdes.

The architecture provides unusual semantics for simultaneous writes to registers. Multiple operations may write into a register in a cycle provided they all write the *same value*. In this case, the result stored in the register is simply the value being written. On the other hand, if multiple operations attempt to write different values into a register simultaneously, then the result stored in the register is undefined. In the case of predicate registers, this atypical semantics is useful for efficient evaluation of boolean reductions; see the description of compare-to-predicate operations in Section 9.3. The utility of this semantics in the case of other types of registers such as general-purpose and floating-point registers is not well-understood. The full generality is provided to facilitate research in this area.

The architecture permits multiple branch operations in an instruction. Moreover, the latency of a branch operation is an architectural parameter that is specified for each machine in its mdes. A significant point to note is that a branch takes effect after exactly n cycles where n is the latency of the branch. That is, branch operations always have the "equals" semantics, even in "less-than-or-equals" machines.

Like integer or floating point operations, branch operations are executed in a parallel pipelined manner with the following implications. First, consider the execution of an instruction containing multiple branch operations. In this case, the result is well-defined only when at most one operation takes the corresponding branch. If more than one branch operations specify that the corresponding branches be taken, then the result of the execution is undefined. In other words, simultaneous multiple writes to the program counter (PC) are not permitted. It is the compiler's responsibility to ensure that, in an instruction, at most one branch takes. Second, branch

operations in the delay slots of a branch are executed in a pipelined fashion, which may give rise to "visits" (see Section 11).

3.3 Predicated execution

Predicated or guarded execution is a way to enforce the semantics of program control-flow, which is different from the one provided by branch operations. It refers to the conditional execution of operations based on a boolean-valued source operand, called a predicate. Predicated execution is frequently an efficient method to handle conditional branches and provides much more freedom in code motion than possible otherwise. For example, a compiler can use predicated execution to eliminate many of the conditional branches present in a program, a technique commonly referred to as *if-conversion*. If-conversion not only reduces the impact of branch latencies but has the benefit that operations can be moved freely across branch boundaries. If-conversion is used in software-pipelining of loops with conditionals [22] and in hyperblock scheduling [23]. Predicates may also be used to fill branch delay slots more effectively than possible otherwise. Note, however, that predicated execution is no substitute for branching. Its benefits are obtained at a price--operations that are nullified still consume machine resources in the EPIC model of execution. The Cydra 5 architecture [6] included a general form of predicated execution. Other architectures with some form of conditional execution include HP PA-RISC and DEC Alpha.

The model of predicated execution in the HPL-PD architecture is an enhanced version of the one provided by the Cydra 5 architecture. The architectural features relevant to predicated execution are as follows:

1. There are 1-bit predicate register files, which are partitioned into static and rotating portions (see Section 4).
2. The main operations on predicate registers are a set of compare-to-predicate operations, each of which can set up to 2 predicate registers. There are also operations to move data between predicate registers and integer registers and operations to clear predicate registers *en masse*. In addition, all of the integer computation operations and standard load/store operations can be used to operate on 32 predicates at a time by using control register aliases (see Section 4).
3. A class of compare-to-predicate operations and the unusual semantics of simultaneous writes to registers described in Section 3.2 provide an efficient method to evaluate boolean reductions. This provides an effective method for control-height reduction.
4. Most operations have a predicate input that guards their execution. We call such operations *predicated operations*. When the predicate has value 1 (true), the operation executes normally. When the predicate has value 0 (false), the execution of the operation is nullified *i.e.*, no change in the machine state takes place. The exact mechanism by which the execution of an operation is nullified depends upon the implementation. Predicated operations can be executed unconditionally by specifying the static predicate register PR1, which is permanently 1 when used as an input (see also Section 4.5).

   There are a few operations whose execution cannot be nullified. These operations also have one or more predicate inputs, but they use these inputs like data inputs. Branch operations related to software-pipelining of loops fall in this category. Compare-to-predicate operations are also an exceptional case for which it is hard to classify whether they are predicated or not (see Section 9.3 for more details).

3.4 Speculative execution

The architecture supports speculative execution of most operations; the exceptions are stores to memory and branch operations. Some of the architectural support for speculative execution is similar to the one described in [14]; see also[11, 15].

In this report, we use the term speculative execution in a broader sense than it is used in the literature. The notion of speculative execution includes two distinct forms of speculation.

1. *Control speculation*: This refers to the execution of operations before it has been determined that they would be executed in the normal control-flow of execution. Traditionally, speculative execution has been identified with control speculation.
2. *Data speculation*: This refers to the execution of operations with potentially incorrect operand values. An example of the data speculation is the execution of both a load and an operation that uses the loaded value before potentially aliasing stores that originally preceded them. The operation that uses the loaded value may execute with incorrect data. In the case of data speculation, the compiler must ensure program correctness by creating a code sequence, which is invoked to re-issue the operations that were executed with incorrect data. The run-time disambiguation mechanism described in Section 10.6 provides a form of data speculation.

The main issue in both forms of speculation is the correct handling of *architecturally visible* exceptions. If a speculatively executed operation generates such an exception, then the exception should be deferred and reported only after it has been determined that the exception would also occur in the normal program execution.

To support exception handling, the architecture provides speculative tag bits on registers and provides speculative and non-speculative versions for most operations except for store and branch operations, which don't have speculative versions. We use the modifier "E" (for eager) to denote the speculative version of an operation; that is, the name for the speculative version is obtained by appending E to the name of the non-speculative counterpart. A couple of additional points to note are as follows. First, we assume that both speculative and non-speculative version of an operation behave identically with respect to the types of exceptions generated and the conditions under which they are generated. Second, we defer all issues related to the processing of exceptions for non-speculative operations to a later version of the architecture.

In order to clearly explain speculative execution, we introduce terms for generating and signaling an exception. Generation is the detection and logging of an exception condition resulting from the execution of an operation. A generated exception causes an exception signal when it is known that the operation would have executed in the original non-speculative code sequence. Exception signaling causes the CPU to treat an exception condition by invoking exception processing, which may result in abnormal program termination, invoking an exception handler, or other special actions.

Prior to any speculative code motion, exception generation and signaling are simultaneous. Speculative code motion may separate the exception generation and signaling times using the speculative tag bit to propagate the exception condition from an operation which generates the exception to an operation which signals the exception. This allows an operation to execute speculatively and generate but not signal an exception. The exception is signaled only if it is determined later that the operation would have also executed in the original program. The means by which this is accomplished is described below.

Execution of a speculative operation: If the speculative tag of every source register is not set, then the execution proceeds normally when the operation doesn't generate an exception. When the operation does generate an exception, the speculative tag of the destination register(s) is set. If the speculative tags of one or more source register are set, then an exception propagation occurs. The operation simply sets the speculative tag bit of its destination register.

To report an exception, the value of PC and any other required state information at the time of the exception must be recorded and propagated. This report doesn't describe the mechanism for doing this; that is left for a later version of the report. (We will probably use an extension of the mechanism described in [14]. The main complication is that predicates are 1-bit registers.)

Execution of a non-speculative operation: If the speculative tag of every source registers is not set, then the execution proceeds normally, and any exception generated by the operation is immediately signaled. If the speculative tags of one or more source registers are set, then it indicates that an exception was generated by a speculative operation. The exception is, therefore, signaled using the recorded state information. If multiple source registers have their speculative tag bit set, the exception corresponding to the first operand is reported.

Table 1 summarizes the handling of exceptions; a --- indicates that the corresponding value is undefined. (For instance, the IEEE floating-point standard requires that the destination of an excepting instruction be left unchanged.)

Table 1: Semantics of speculative operations

Operation type	Tag bits of source registers	Operation generates an exception	Destina-tion's tag bit	Other actions	Signal exception if enabled
Speculative	0 for all sources	No	0	Update destination register with the result	No
		Yes	1	Record PC and other state information for exception reporting	No
	1 for one or more sources	Don't care condition	1	Record that an exception was propagated	No
Non-speculative	0 for all sources	No	0	Update destination register with the result	No
		Yes	0	---	Yes
	1 for one or more sources	Don't care condition	0	---	Yes

3.5 Exceptions and exception handling for non-speculative operations

A detailed description of all issues related to exceptions and exception handling for non-speculative operations are left to a later version of the architecture. These issues include the types of exceptions generated by an operation, conditions under which they are generated, masking of exceptions, and the exception handling mechanism. The only point we would like to emphasize is the one related to speculative execution. That is, a non-speculative operation also signals an exception when the speculative tag of any of its source registers is set.

4 Register files

This section describes the data types and the types of register files supported by the architecture. Section 4.1 describes the data types; Section 4.2 comments on static and rotating registers; the remaining sections describe the various types of register files.

A machine in the HPL-PD family can have one or more register files of each type except for the control register file; each machine contains exactly one control register file. The number of register files of each type and the number of registers in each file are architectural parameters, which are specified separately for each machine. We assume that each register file in a machine has a unique name and that a register specifier consists of a register file name and a register number within the file.

4.1 Data types

The data types supported by the architecture are byte, integer, floating-point number and predicate. The first three data types and their formats are the same as in HP PA-RISC.

- Byte: Bytes are signed or unsigned 8-bit quantities and are packed four to a word. They may be used to represent signed values in the range -128 through 127, unsigned values in the range 0 through 255, an arbitrary collection of 8 bits, or an ASCII character.
- Integer: There are signed and unsigned integers in two lengths: half-word (16 bits) and word (32 bits). 64-bit integers will be included in a later version. Half-word integers must be aligned at two-byte boundaries, *i.e.*, they must be stored in memory at addresses evenly divisible by two. Word integers must be aligned at four-byte boundaries. Note that unsigned integers can also be used to represent an arbitrary collection of 16 or 32 bits.
- Floating point: There are IEEE compliant 32-bit single precision and 64-bit double precision formats for floating point numbers. Single precision numbers must be aligned at four-byte boundaries, and double precision numbers at eight-byte boundaries.
- Predicate: 1-bit boolean values

4.2 Static & Rotating Registers

The architecture supports both static and rotating registers. Most register files (see the following sections) are partitioned into static and rotating portions with differing numbers of registers. The static registers are conventional registers; the rotating registers logically shift in register address space every time the rotating register base (RRB) is decremented by certain loop-closing branches; see Section 11.2.5.

Rotating registers have the special property that the register state prior to rotation is closely related to the register state after rotation. Consider a register file R with n_r number of rotating registers; that is, the file contains registers R[0] ... R[n_r - 1]. Assume that x is the value contained in some register R[j] prior to a "rotate" operation where j is in the range $0 \leq j \leq (n_r - 2)$. Then after a rotate operation R[j +1] has value x. Thus, for example, if R[0] has value x prior to rotation then R[1] has value x after a single rotation. Note that the value of R[0] after rotation is unspecified. We call this an *open loop* model of rotation, *i.e.*, there is a contiguous window of size n_r into a circular register file of unspecified size. This simplifies the definition of architectural families of processor with varying numbers of rotating registers in a file and provides better code compatibility across the family.

Typically, rotating registers are implemented in terms of a rotating register base (RRB), which is used to map register offsets to physical registers. A register access involves the modulo sum of the register offset to RRB in order to select a physical register. That is,

register address = (register offset + RRB) mod n_p

where n_p is the number of physical registers and $n_p \geq n_r$. Rotation is accomplished by simply decrementing the RRB. In such an implementation, precisely in the case where the number of physically addressed rotating registers is equal to n_r, we have a *closed loop* model. That is, if $R[n_r - 1]$ has value x prior to rotation, then R[0] has value x after rotation. We constrain code generation schemes to make no use of this fact and thus allow the number of physical registers, in general, to differ from n_r. In the rest of this report, we will assume the above implementation of rotation.

4.3 General purpose register file (GPR):

It is used to store signed or unsigned integers and has the following properties:

- Width = 32 bits + 1 bit for speculative tag.
- Partitioned into static and rotating portions.

By convention, static GPR0 is used as a bit-bucket.

4.4 Floating-point register file (FPR):

It is used to store single or double precision floating-point values, and its characteristics are as follows:

- Width = 64 bits + 1 bit for speculative tag
- Partitioned into static and rotating portions.
- Static FPR0 = 0.0, static FPR1 = 1.0.
- Writes to static FPR0 and FPR1 are ignored.

4.5 Predicate register file (PR):

It is used to store predicates and has the following properties:

- Size = 1 bit + 1 bit for speculative tag.
- Partitioned into static and rotating portions.
- Static PR0 = 0 (*i.e.*, false), static PR1 = 1 (*i.e.*, true).
- Writes to static PR0 and PR1 are ignored.

4.6 Branch-target register file (BTR):

This register file is used to store information about branches that have been prepared. The information for each prepared branch includes the branch target address and its static prediction.

- Size = 64 bits.
- 32 bit branch address + 1 bit for static prediction + 1 bit for speculative tag

4.7 Control register file (CR):

A machine in the HPL-PD family has exactly one register file of this type. Control registers provide a uniform scheme to access internal state within the processor. All integer operations can operate with control registers either as sources or as destinations. However, some of the operations may not make any sense on some of the control registers. We describe the registers in the control register file using mnemonics such as PC. The mapping of these mnemonics to actual register number in the control register file is part of a machine's description.

- Size = 32 bits.
- Control registers supported by the architecture are listed below.
  - PC: Instruction Counter
  - PSW: Processor status word
  - RRB: Register Relocate Base for rotating registers
  - LC: Loop counter
  - ESC: Epilog stage counter
  - PV(i, j): Aliases used to refer to 32 predicate registers at a time in a predicate register file. The first index identifies a register file, and the second index identifies a group of 32 registers in the file. For example, PV(1, 0) refers to the first 32 predicate registers in the predicate register file with name 1, PV(1, 1) to the second 32 registers in the same file, etc.
  - IT(i, j), FT(i, j), PT(i, j): Aliases used to refer to 32 speculative tag bits at a time in a general-purpose register file, a floating-point register file and a predicate register file, respectively. The first index identifies a register file, and the second index identifies a group of 32 tag bits in the file.
  - BTRL(i, j), BTRH(i, j): Aliases used to refer to low order 32-bits and high order 32 bits of a branch target register. The first index identifies a branch target register file and the second index identifies a register in that file.

5 Format of operation description

In the tables given in the subsequent sections, each row describes a single operation or a set of related operations. The description in each row consists of the following:

- Opcode: This specifies opcodes for the set of operations. It consists of a major opcode and a set of modifiers. The major opcode and each of the modifier sets are on separate lines. Each modifier consists of a set of mutually exclusive alternatives, separated by |. For example, the opcode for floating-point add operation is of the form:

  FADD
  S | D

This means that there are two floating-point add operations, one for single-precision (specified by S) and one for double-precision(specified by D). In some cases, there is only one alternative for a modifier, and the modifier may seem unnecessary. The reason for including such a modifier is to provide an easy way to add more alternatives in the future.

Section 3.1 describes the notation we use in this report for writing operation names. Note that the opcodes given in this column are for non-speculative versions. As described in Section 3.1, the opcode for the speculative version of an operation is derived by appending the modifier "E" to the corresponding non-speculative opcode. The Sp field described later indicates whether the architecture provides speculative versions for these operations or not.

- Operation description: This is a short description of the functionality provided by the major opcode and modifiers.
- I/O description: This is used to describe the following aspects of operations in the family. First, it provides a stylized description of the information about the register files from which the operation reads its *explicitly specified* operands and to which it writes its *explicitly specified* results. Second, it establishes an ordering on the sources and destinations, which is used in describing the semantics. Third, it is used to specify whether operations in the family are predicated or not.

  The format of the I/O description is as follows:

  - <predicated> <source>, ... , <source> : <destination>, ..., <destination>

  The first term <predicated> is either P? or empty with the following interpretation:

  - P?: This denotes that operations in the family are predicated. That is, they take a predicate register (static or rotating) as an input which is used to conditionally nullify the operation.
  - An empty field (*i.e.*, no P?) denotes that operations are not predicated, that is, they don't have a predicate input that is used to guard their execution.

  <source> or <destination> is a sequence consisting of one or more of the following:

  - I: General purpose register (static or rotating)
  - F: Floating point register (static or rotating)
  - P: Predicate register (static or rotating). The corresponding predicate input/output is treated like a data input/output.
  - B: Branch target register
  - L: Integer literal. This cannot be part of a <destination> sequence.
  - C: Control register

  Thus, for example, ICL as a <source> specifies that the operand can be an integer register, a control register or an integer literal.
- Sp: Y(es) in this field indicates that these operations have both speculative and non-speculative versions. N(o) in this field indicates that there are no speculative versions of these operations.
- Opcode semantics: This describes the action performed by operations in the family using a C like syntax. In some cases, this field simply refers to the text. For predicated operations, the action specified in this field corresponds to the case when the predicate input is 1 (true).

It is implicit in the specification that when the predicate input is 0, then the execution of operation is nullified.

In the semantic description of operations, we use the following notation. Src<n>, for <n> ≥ 1, denotes the n^{th} source operand excluding the guarding predicate (if there is one). Similarly, dest<n>, for <n> ≥ 1, denotes the n^{th} destination. For predicated operations, the guarding predicate is denoted by src0. For operation that are not predicated, src0 has no meaning.

In describing operations, we don't enumerate cases under which an operation generates exceptions. As mentioned in Section 3.4, all issues related to exceptions are left for a later version of the architecture.

6 Integer computation operations

The operation repertoire includes a relatively standard set of arithmetic and logical operations on integers; see Table 2 on the next page. All these operations operate on 32-bit words; there are no byte and half-word operations. Arithmetic operations come in two forms: signed and unsigned. In this report, unsigned operations are distinguished by the word "logical" in their descriptions.

All integer operations can be issued speculatively and have a predicate input to guard their execution. Each operation includes the length (or format) specifier, which seems unnecessary given that there is only one option, *i.e.*, W. It is included so that the operation repertoire can be easily extended to support both 32-bit and 64-bit integers in the future.

The repertoire includes opcodes for complex operations like multiply, divide, remainder, min and max. They may be provided in hardware or emulated in software. Shift and add logical operations are useful in address computation, which typically involves multiplying by a small constant.

Table 2: Integer computation operations

Opcode	Operation description	I/O description	Sp	Opcode semantics
ABS W	Absolute value word	P? ICL : IC	Y	dest1 = abs(src1)
ADD W	Add word	P? ICL, ICL : IC	Y	dest1 = src1 + src2
ADDL W	Add logical word	P? ICL, ICL : IC	Y	dest1 = src1 + src2
AND W	AND word	P? ICL, ICL : IC	Y	dest1 = src1 & src2
ANDCM W	AND complement word	P? ICL, ICL : IC	Y	dest1 = src1 & (! src2)
DIV W	Divide word	P? ICL, ICL : IC	Y	dest1 = src1 / src2
DIVL W	Divide logical word	P? ICL, ICL : IC	Y	dest1 = src1 / src2

Table 2 (cont.): Integer computation operations

Opcode	Operation description	I/O description	Sp	Opcode semantics
MAX W	Maximum value word	P? ICL, ICL : IC	Y	dest1 = max(src1, src2)
MAXL W	Maximum value logical word	P? ICL, ICL : IC	Y	dest1 = max(src1, src2)
MIN W	Minimum value word	P? ICL, ICL : IC	Y	dest1 = min(src1, src2)
MINL W	Minimum value logical word	P? ICL, ICL : IC	Y	dest1 = min(src1, src2)
MPY W	Multiply word	P? ICL, ICL : IC	Y	dest1 = src1 * src2
MPYL W	Multiply logical word	P? ICL, ICL : IC	Y	dest1 = src1 * src2
NAND W	Nand word	P? ICL, ICL : IC	Y	dest1 = !(src1 & src2)
NOR W	Nor word	P? ICL, ICL : IC	Y	dest1 = !(src1 | src2)
OR W	Or word	P? ICL, ICL : IC	Y	dest1 = src1 | src2
ORCM W	Or complement word	P? ICL, ICL : IC	Y	dest1 = src1 | (! src2)
REM W	Remainder word	P? ICL, ICL : IC	Y	dest1 = src1 % src2
REML W	Remainder logical word	P? ICL, ICL : IC	Y	dest1 = src1 % src2
SH1ADDL W	Shift 1 and add logical word	P? ICL, ICL : IC	Y	dest1 = (src1 << 1) + src2
SH2ADDL W	Shift 2 and add logical word	P? ICL, ICL : IC	Y	dest1 = (src1 << 2) + src2
SH3ADDL W	Shift 3 and add logical word	P? ICL, ICL : IC	Y	dest1 = (src1 << 3) + src2
SHL W	Shift left word	P? ICL, ICL : IC	Y	dest1 = src1 << src2
SHR W	Shift right word	P? ICL, ICL : IC	Y	dest1 = src1 >> src2
SHLA W	Shift left arithmetic word	P? ICL, ICL : IC	Y	dest1 = shla(src1, src2)
SHRA W	Shift right arithmetic word	P? ICL, ICL : IC	Y	dest1 = shra(src1, src2)
SUB W	Subtract word	P? ICL, ICL : IC	Y	dest1 = src1 - src2
SUBL W	Subtract logical word	P? ICL, ICL : IC	Y	dest1 = src1 - src2
XOR W	Exclusive OR word	P? ICL, ICL : IC	Y	dest1 = src1 $\oplus$ src2
XORCM W	Exclusive OR complement word	P? ICL, ICL : IC	Y	dest1 = src1 $\oplus$ (! src2)

7 Floating point computation operations

The operation repertoire provides standard arithmetic operations like add, multiply as well as combined multiply-add style of operations for both single and double precision floating point numbers. The repertoire includes opcodes for complex operations like divide, square root and reciprocal. They may be provided in hardware or emulated in software.

Table 3 lists the floating-point computation operations. All these operations can be issued speculatively and have a predicate input that guards their execution.

Table 3: Floating-point computation operations

Opcode	Operation description	I/O description	Sp	Opcode semantics
FADD S | D	Add single or double precision	P? F, F : F	Y	dest1 = src1 + src2
FABS S | D	Absolute value single or double precision	P? F : F	Y	dest1 = abs(src1)
FDIV S | D	Divide single or double precision	P? F, F : F	Y	dest1 = src1 / src2
FMAX S | D	Maximum value single or double precision	P? F, F : F	Y	dest1 = max(src1, src2)
FMIN S | D	Minimum value single or double precision	P? F, F : F	Y	dest1 = min(src1, src2)
FMPY S | D	Multiply single or double precision	P? F, F : F	Y	dest1 = src1 * src2
FMPYADD S | D	Multiply and add single or double precision	P? F, F, F : F	Y	dest1 = src1 * src2 + src3
FMPYADDN S | D	Multiply, add and negate single or double precision	P? F, F, F : F	Y	dest1 = - (src1 * src2 + src3)
FMPYRSUB S | D	Multiply and reverse subtract single or double precision	P? F, F, F : F	Y	dest1 = src3 - src1 * src2
FMPYSUB S | D	Multiply and subtract single or double precision	P? F, F, F : F	Y	dest1 = src1 * src2 - src3
FRCP S | D	Reciprocal single or double precision	P? F : F	Y	dest1 = 1.0 / src1
FSQRT S | D	Square root single or double precision	P? F : F	Y	dest1 = sqrt(src1)
FSUB S | D	Subtract single or double precision	P? F, F : F	Y	dest1 = src1 - src2

8 Conversion and move operations

This section describes operations for converting data types and for moving data from one register file to another. The architecture provides the following types of conversion operations; see Table 4.

1. Integer to floating-point: These operations convert signed or unsigned (logical) integer values to single or double precision floating-point values. They take their arguments from general purpose registers and leave their results in floating-point registers. That is, a move from general-purpose registers to floating-point registers is implicit in these operations.

2. Floating-point to integer: These operations convert single or double precision floating point values to signed or unsigned (logical) integer values. They take their arguments from floating-point registers and leave their results in general purpose registers. Again, a move is implicit in these operations.

3. Floating-point to floating point: These operations are used for conversion between single and double precision values.

4. Sign extension: These operations convert byte or half-word quantities into sign-extended 32-bit format. Note that byte and half-word load operations are unsigned loads in the sense that they zero-extend (not sign-extend) the loaded values to convert them into 32-bit format (see Section 10). Thus, to load a byte or a half-word and operate on it as a signed quantity, it is necessary to explicitly sign-extend the loaded value.

All these operations can be issued speculatively and have a predicate input that guards their execution.

Table 4: Conversion operations

Opcode	Operation description	I/O description	Sp	Opcode semantics
CONVWS	Convert an integer to single precision floating point	P? I : F	Y	dest1 = (float) src1
CONVWD	Convert an integer to a double precision floating point	P? I : F	Y	dest1 = (double) src1
CONVLWS	Convert an unsigned integer (logical) to single precision floating point	P? I : F	Y	dest1 = (float) src1
CONVLWD	Convert an unsigned integer (logical) to a double precision floating point	P? I : F	Y	dest1 = (double) src1
CONVSW	Convert a single precision floating point to an integer	P? F : I	Y	dest1 = (int) src1
CONVDW	Convert a double precision floating point to an integer	P? F : I	Y	dest1 = (int) src1
CONVLSW	Convert a single precision floating point to an unsigned integer (logical)	P? F : I	Y	dest1 = (unsigned int) src1
CONVLDW	Convert a double precision floating point to an unsigned integer (logical)	P? F : I	Y	dest1 = (unsigned int) src1
CONVSD	Convert a single precision to a double precision floating point	P? F : F	Y	dest1 = (double) src1
CONVDS	Convert a double precision to a single precision floating point	P? F : F	Y	dest1 = (float) src1
EXTS B | H	Extend sign byte or half-word	P? I : I	Y	See the description

Table 5 lists operations used to transfer data between various register files. All these operations can be issued speculatively and have a predicate input to guard their execution.

Operations that move data between a GPR file and a FPR file put no interpretation on the value being transferred. For example, MOVEGF.L and MOVEGF.U simply take 32 bits in a GPR and store it into the lower or upper half of a FPR, respectively. Similar comments apply to MOVEFG.L and MOVEFG.U.

This version of HPL-PD provides separate moves for single-precision and double-precision

floating point values in order to be consistent with other floating point operations. Both these operations may be implemented as a simple 64-bit move between FPRs.

Table 5: Move operations

Opcode	Operation description	I/O description	Sp	Opcode semantics
MOVE	Move a literal, GPR or CR to a GPR or CR	P? ICL : IC	Y	dest1 = src1
MOVEGF L | U	Move a GPR to lower or upper half of a FPR	P? I : F	Y	dest1. (lower or upper half) = src1
MOVEF S | D	Move a FPR to a FPR single or double precision	P? F : F	Y	dest1 = src1
MOVEFG L | U	Move to a GPR lower half or upper half of a FPR	P? F : I	Y	dest1 = src1.(lower or upper half)
MOVEPG	Move a predicate register to a GPR and clear higher order bits	P? P : I	Y	dest1.lsb = src1 dest1.(other bits) = 0
MOVEGBP	Move a GPR bit to a predicate register	P? I, IL : P	Y	dest1 = src1[src2], *i.e.*, src2th bit of src1
MOVEB	Move a BTR to another BTR	P? B : B	Y	dest1 = src1
MOVEGCM	Move a GPR to a CR using a mask	P? IL, IL, C : C	Y	dest1 = (src1 & src2) | (src3 & ~src2)

MOVEGBP moves a specified bit from a GPR to a predicate register; one of its uses is in loading a predicate register from the main memory, for example, to unspill the register (see Section 10.7). MOVEGCM initializes a subset of bits in a control register and leave the other bits unchanged. Please note that MOVEGCM replaces the LDCM operation in the earlier version. Its main use is in writing to a group of predicate registers or a group of speculative tag bits in order to restore or unspill them from the main memory. The second source is the mask that specifies the indices of predicate registers or tag bits to restore.

Note that there is no operation to move data between predicate registers, since these moves can be performed using the appropriate type of compare-to-predicate operations (see Section 9.4.3). For convenience, one can define an operation (say MOVEPP) to move data between predicate registers for the internal use of a compiler; the operation is then translated to an appropriate type of compare-to-predicate operation during the code emission phase.

Table 6 lists operations that transfer a specified bit from a GPR to the speculative tag associated with a register. All these operations can be issued speculatively and have a predicate input to guard their execution. These operations are used to write to speculative tag bits while restoring or unspilling registers from the main memory (see Section 10.7).

Table 6: Operations to move a GPR bit to a speculative tag bit

Opcode	Operation description	I/O description	Sp	Opcode semantics
MOVEGBGT	Move a GPR bit to GPR speculative tag	P? I, IL : I	Y	dest1 = src1[src2], i.e., src2th bit of src1
MOVEGBFT	Move a GPR bit to FPR speculative tag	P? I, IL : F	Y	dest1 = src1[src2], i.e., src2th bit of src1
MOVEGBPT	Move a GPR bit to PR speculative tag	P? I, IL : P	Y	dest1 = src1[src2], i.e., src2th bit of src1
MOVEGBBT	Move a GPR bit to BTR speculative tag	P? I, IL : B	Y	dest1 = src1[src2], i.e., src2th bit of src1

Table 7 lists operations that clear predicate registers *en masse*. These operations clear ALL predicate registers or a subset, namely, static or rotating. They are used to clear predicate registers before entering a software pipelined loop. HPL-PD provides control register aliases to predicate registers so that we can operate on 32 predicate registers at a time. We can achieve the effect of these operations by clearing 32 registers at a time. But, simulating any one of them requires multiple operations. Since *en masse* clear is not that difficult to implement in hardware, we included these operations rather than simulating them.

Table 7: Operations to clear the predicate registers *en masse*

Opcode	Operation description	I/O description	Sp	Opcode semantics
PRED_CLEAR_ALL	Clear all predicate registers, static and rotating	P? :	Y	P0, ..., Pn = 0 P[0], ..., P[m] = 0
PRED_CLEAR_ALL_STATIC	Clear all static predicate registers	P? :	Y	P0, ..., Pn = 0
PRED_CLEAR_ALL_ROTATING	Clear all rotating predicate registers	P? :	Y	P[0], ..., P[m] = 0

9 Compare operations

These operations are used to determine a relation such as =, < etc. between two integers or floating-point values. The result of the operation is a boolean value. The architecture provides two different types of compare operations. The first type of operations, called *compare-to-register*, write the result into general purpose registers. The second type of compare operations, called *compare-to-predicate*, write the result into predicate registers. Section 9.1 describes the compare conditions that can be specified with these operations. Sections 9.2 and 9.3 describe compare-to-register and compare-to-predicate operations, respectively. Section 9.4 contains some comments about the usage of these operations.

9.1 Compare conditions

Table 8 lists the compare conditions that can be specified with integer compare operations. These conditions, including their mnemonics, are identical to the compare/subtract conditions in the HP PA-RISC architecture.

Table 8: Compare conditions for integer compare operations

Condition name	Description	Result of comparison
	Always false	0 (false)
=	Equal	opd1 = opd2
<	Signed less than	opd1 < opd2
<=	Signed less than or equal to	opd1 <= opd2
>	Signed greater than	opd1 > opd2
>=	Signed greater than or equal to	opd1 >= opd2
SV	Opd1 minus opd2 overflows	Overflow (opd1 - opd2)
OD	Opd1 minus opd2 is odd	Odd (opd1 - opd2)

Condition name	Description	Result of comparison
TR	Always true	1 (true)
<>	Not equal	opd1 != opd2
<<	Logical less than	opd1 < opd2
<<=	Logical less than or equal to	opd1 <= opd2
>>	Logical greater than	opd1 > opd2
>>=	Logical greater than or equal to	opd1 >= opd2
NSV	Opd1 minus opd2 doesn't overflow	! Overflow (opd1 - opd2)
EV	Opd1 minus opd2 is even	Even (opd1 - opd2)

In cases where signed and unsigned (called logical) comparisons have different semantics, both forms are provided. In the description of compare operations (see Tables 10 and 11), we will use <I-cond> as an abbreviation for one of the integer compare conditions.

Table 9 lists the floating-point compare conditions. They are a subset of the ones provided in the PA-RISC architecture, and their mnemonics are taken directly from the PA-RISC manual. Note that the floating-point compare conditions in PA-RISC conform to the IEEE standard. Floating-point comparisons are exact and neither overflow or underflow. Between any two operands, one of four mutually exclusive relations is possible: *greater than* (>), *less than* (<), *equal* (=) and *unordered*. The last case arises when at least one operand is a NaN. Every NaN compares unordered with every operand, including another NaN. Also, comparisons ignore the sign of zero, so +0 is equal to -0. The table specifies the result for each combination of the compare condition and the relation that holds between the operands; T means the result is 1, F means the result is 0. Consider, for example, the ?= compare condition. If the operands are either equal or unordered, then the result is 1; otherwise the result is 0. In the description of compare operations, we will use <F-cond> as an abbreviation for one of the floating-point compare conditions.

Table 9: Compare conditions for floating-point compare operations

Condition name	Relation between the operands			
	>	<	=	unordered
false?	F	F	F	F
?	F	F	F	T
=	F	F	T	F
?=	F	F	T	T
!?>=	F	T	F	F
?<	F	T	F	T
!?>	F	T	T	F
?<=	F	T	T	T

Condition name	Relation between the operands			
	>	<	=	unordered
!?<=	T	F	F	F
?>	T	F	F	T
!?<	T	F	T	F
?>=	T	F	T	T
!?=	T	T	F	F
!=	T	T	F	T
!?	T	T	T	F
true?	T	T	T	T

PA-RISC also includes a set of floating-point compare conditions, which are similar to the ones described above except that they cause an invalid operation exception if their operands are unordered. We may incorporate these compare conditions in a later version of the architecture.

9.2 Compare-to-register operations

Table 10 lists the compare-to-register operations. As mentioned earlier, these operations write their results into general-purpose registers. Integer compare operations operate on word (32-bit) quantities; there are no operations that explicitly operate on byte and half-word data types. On the other hand, floating point operations provide both single and double precision forms. Note that all these operations can be issued speculatively, and they have a predicate input that guards their execution.

Table 10: Integer and floating-point compare-to-register operations

Opcode	Operation description	I/O description	Sp	Opcode semantics
CMPR W <I-cond>	Compare to register word compare condition	P? IL, IL : I	Y	dest1 = src1 <I-cond> src2
FCMPR S | D <F-cond>	Compare to register single or double precision compare condition	P? F, F : I	Y	dest1 = src1 <F-cond> src2

9.3 Compare-to-predicate operations

Compare-to-predicate operations are the primary operations that write into predicate registers, and the architecture provides a rich set of these operations. This section describes the semantics of these operations; the next section contains comments about their utility and usage.

Table 11 describes the various forms of compare-to-predicate operations provided by the architecture. Each operation has three sources; the first is a predicate register and the other two specify the values to be compared. Unlike most other operations, a compare-to-predicate operation specifies two destinations, both of which are predicate registers. In addition to the compare condition, each operation also specifies the type of action that is performed on each of the two destinations. These action specifiers are called <D-action> in the table and are discussed later in this section. See Section 9.4.3 for examples of fully expanded compare-to-predicate opcodes.

Table 11: Integer and floating-point compare-to-predicate operations

Opcode	Operation description	I/O description	Sp	Opcode semantics
CMPP W <I-cond> <D-action> <D-action>	Compare to predicate word compare condition action to take for the first destination action to take for the second destination	P, IL, IL : P, P	Y	r = src2 <I-cond> src3 dest1 = See text and Table 12 dest2 = See text and Table 12
FCMPP S | D <F-cond> <D-action> <D-action>	Conditional compare to predicates single or double precision compare condition action to take for the first destination action to take for the second destination	P, F, F : P, P	Y	r = src2 <F-cond> src3 dest1 = See text and Table 12 dest2 = See text and Table 12

The semantics of one of these operations is as follows. The input values are compared according to the condition specified in the operation. The result of the comparison along with the predicate input and the <D-action> specifier for a destination determine the action performed on the corresponding destination.

To understand the action specifiers, consider one of the destination. For each combination of the predicate value and the result of the comparison, there are three possible choices as to what can be done with the destinations. The choices are as follows:

1. Write 0 into the destination register.
2. Write 1 into the destination register.
3. Leave the destination unchanged.

That is, there are three possible actions for each of the four combinations of the predicate input and compare result. Thus, there are a total of $3^4 = 81$ possible actions that can be performed on a destination. Out of these, the architecture supports the ones described in Table 12. That is,

<D-action> = UN | CN | ON | AN | UC | CC | OC | AC.

We believe that these are the ones that are important and that they are sufficient to cover most of the requirements imposed by the various uses of predicates.

Table 12: Destination action specifiers for compare-to-predicate operations and their semantics. An entry with -- means leave the target unchanged.

Predicate input	Result of comparison	On result				On the complement of result			
		UN	CN	ON	AN	UC	CC	OC	AC
0	0	0	--	--	--	0	--	--	--
0	1	0	--	--	--	0	--	--	--
1	0	0	0	--	0	1	1	1	--
1	1	1	1	1	--	0	0	--	0

A bit of terminology that we use internally and is reflected in the names of these modifiers as well as in this report. *Unconditional* class refers to operations with UN and UC modifiers, and *conditional* class refers to the ones with CN and CC modifiers. *OR* class refers to operations with ON and OC modifiers as they are used in OR reductions, and *AND* class refers to the ones with AN and AC modifiers as they are useful in AND reductions (see the next section). N in the name stands for "normal result" and C stands for "complement of result".

First, we discuss the four actions grouped under the heading "on result". Unconditional operations (UN) always write into the destination register. If the predicate input is false, they clear the destination register; otherwise, they copy the result of the comparison into the destination register. In other words, these operation effectively compute the boolean conjunction of the input predicate and the result of the comparison. Conditional operations (CN) behave like predicated compares. That is, if the predicate input is false, they leave the destination unchanged; otherwise they copy the result of the comparison into the destination. Note that both conditional and unconditional operations display identical functionality if the predicate input is true and differ only in the case when the predicate input is false. The other two classes (OR and AND) are useful in efficient evaluation of boolean reductions (see the next section). Operations in the OR class (ON) write a 1 into the destination register only if both the predicate input and the result of the comparison are true. Otherwise, they leave the destination unchanged. Operations in the AND class (AN) write a 0 into the destination register when the predicate input is true and the result of the comparison is false. Otherwise, they leave the destination unchanged.

The other four actions are similar to the ones described above except that they implicitly complement the result of the comparison. For example, consider UN and UC modifiers. Operations with UN modifier write the result of the comparison into the destination register if the predicate is true, whereas operations with UC modifier write the complement of the result.

A point to note is that the semantics of compare-to-predicate operations is somewhat unique with respect to the conditions under which the writes to the destination register are nullified. As discussed in Section 3.3, a predicated operation doesn't write into its destinations if the predicate input is false. Conditional compares are the only ones that follow this semantics. Unconditional compare operations always write into their destinations and use the predicate input like a regular data input. Operations in the OR class and the AND class use not only the predicate input but also the result of the comparison to decide whether to modify their destinations.

9.4 Usage and compiling issues

This section contains a brief discussion about the usage of compare operations. A detailed discussion is beyond the scope of this report.

9.4.1 Compare-to-register vs. compare-to-predicate operations

Since the architecture provides two different types of compare operations, a compiler must decide when to generate compare-to-register operations and when to generate compare-to-predicate operations. A simple view is as follows. Predicates are used mainly to model the control-flow of a program, either using branches or using predicated execution of operations. Thus, if the result of a comparison is used to model the control-flow, then the compiler uses a compare-to-predicate operation. On the other hand, if the result is used for other purposes, *e.g.*, as a boolean datum in an expression or as a value to be stored in the memory, then the compiler uses a compare-to-register operation. Efficient use of the two types of compare operations, however, may require more sophisticated compiler-heuristic than the one outlined above for the following reasons. First, there are cases where the result is used in both ways. These cases can be handled either by moving the result from one register file to another or by computing the result twice. Second, the architecture doesn't support a full suite of boolean operations on predicate registers. Thus, sometimes it is more efficient to compute a predicate by using compare-to-register operations and then moving the result to a predicate register.

9.4.2 OR and AND classes of operations (ON, OC, AN and AC modifiers)

These operations require some explanation because of their somewhat unusual semantics and because their effective usage relies upon the cooperation of several operations. To simplify the exposition, we phrase the discussion in terms of operations with only one destination.

These operations are used in the efficient evaluation of boolean reductions. Using these operations, any OR or AND reduction can be evaluated effectively in a time equal to the latency of a single compare operation provided there is enough parallelism in the machine. As an example, we illustrate the evaluation of the following OR-reduction. In the expression, p is a predicate register and a, b, c, d are general purpose registers.

$$p = (a < b) \vee \neg(c > d) \vee (a < c) \vee (b > d)$$

The code for evaluating the reduction is given below. The first statement in the code is a way to set the register p to 0. In the statement, 0 refers to the integer literal 0. Note that the negation in the second term can be computed either by complementing the compare condition or by using the OC modifier. The code uses the first approach.

Code for evaluating the reduction

```
/ *  The code assumes single destination compares  * /
p = CMPP.W. .UN(0,0); / * Set  p to 0 * /
            / * Compare condition is "always false"  * /
p = CMPP.W.<.ON(a,b);
p = CMPP.W.<=.ON(c,d);
p = CMPP.W.<.ON(a,c);
p = CMPP.W.>.ON(b,d);
```

The code works as follows. The predicate register p is initialized to 0. Each compare operation, then, overwrites the register by 1 if the result of the comparison is true, and leaves the register

unchanged if the result is false. In other words, if one or more comparisons evaluates to true, the predicate register is set to 1; otherwise it contains the initial value 0. This correctly evaluates the reduction.

The most important point to note is that there are no restrictions in scheduling the compare operations other than those imposed by the amount of parallelism in a machine. They can be scheduled in any order including in the same instruction. Putting it another way, there are no output dependences to honor even though they all write to the same register. As an example, the code can be scheduled as shown below on a machine with at least four units.

Schedule for the code	
Cycle	Instruction
1	p = CMPP.W. .UN(0,0);
$\ell+1$	p = CMPP.W.<.ON(a,b); p = CMPP.W.<=.ON(c,d); p = CMPP.W.<.ON(a,c); p = CMPP.W.>.ON(b,d);
$2\ell+1$	Value in the register p available for use

Thus, the evaluation of the reduction takes 2ℓ cycles where ℓ is the latency of an operation. Typically, the initialization of p can be overlapped with other code in the program. Thus, the evaluation effectively takes only ℓ cycles.

Two features of the architecture cooperate to make the above possible. First is the semantics of simultaneous write to a register. As mentioned in Section 3.2, the architecture permits multiple operations to simultaneously write a value into a register provided all such operations write the *same value*. In this case, the result stored in the register is well-defined and is the value being written into the register. Second is the semantics of the OR class. Consider the four compare operations issued simultaneously in the schedule given above. Within these operations, the ones that write into p write the value 1; it is never the case that one operation attempts to write 1 and another 0. Thus, the value stored into p is well-defined even though multiple operations write into p.

As the name implies, operations in the AND class are used for boolean AND reduction. To evaluate such a reduction, the result predicate is initialized to 1. Then, each of the compare operation participating in the reduction over-writes the register by 0 if the result of the comparison is false.

Note that compare-to-predicate operations also have a predicate input, and each operation effectively computes the boolean AND of the predicate input and the result of the compare. In other words, OR and AND classes of operations can be used to efficiently evaluate reductions in which each term is the boolean AND of the result of a compare and the value stored in a predicate register.

9.4.3 Use of compare-to-predicate operations

Compare-to-predicate operations were designed primarily to address the compiler requirements in three important areas (see below), all of which are related to the broad issue of how to efficiently model the control-flow of a program. To simplify the exposition, we phrase the early part of the discussion in terms of operations with only one destination. Thus, we only consider operations with the following modifiers: UN, CN, ON and AN. Note that the others, *i.e.*, the ones with C as the second letter, can be simulated by simply complementing the compare condition.

1. Predicated execution: In this area, there are two techniques of interest: if-conversion (see [24, 23, 22]) and predicated code motion. If-conversion uses compare operations in UN, CN, and ON classes. Operations in the UN class are suitable only for the if-conversion of "structured" code. They are actually the best choice in that case, since they don't require extra operations to initialize predicates. ON and CN classes can handle "unstructured" code. The use of CN class requires fewer operations to initialize predicates than the use of ON class. However, the use of ON class gives much more freedom in scheduling; there are no output dependences to honor even if operations target the same register (see the last section). Table 13 summarizes the above discussion. The compiler requirements for predicated code motion are similar to those for if-conversion.

Table 13: Pros and cons of using UN, CN and ON classes in if-conversion

Operation class	Handles "Unstructured" code	Number of clears	Unnecessary output dependences
UN	No	None	No
CN	Yes	Less	Yes
ON	Yes	More	No

2. Height reduction of control-dependences: Techniques in this area typically require efficient computation of boolean AND of several branch conditions, and the AN class provides the needed functionality.
3. Efficient computation of complex branch conditions: Many cases of complex branch conditions can be computed efficiently by reducing them to a combination of boolean OR and AND reductions. ON and AN class of operations provide a means for fast evaluation of these reductions.

Compare-to-predicate operations with two destinations provide a means to optimize certain common uses of these operations. Compilation technique that make use of predicated execution (*e.g.*, if-conversion) typically associate two predicates with each branch, one for the then clause and one for the else clause. In these cases, the use of dual-destination operations reduces the number of predicate-setting operations by half. For operations with two destinations, it becomes important to provide complementary actions (*e.g.*, UN and UC) so that the action on one destination can be controlled by the result of the comparison and the action on the other by the complement of the result. For example, the UN-UC combination is very useful in the if-conversion of structured code.

There are 64 (8×8) distinct types of two-destination operations. We have decided to include all 64 types in the architecture to support the collection of data on the usefulness of various classes of operations in compiling real programs.

Compare-to-predicate operations can also used to move a bit from a GPR to a predicate register. As an example, the following operation moves the LSB of the general-purpose register GPR2 to the predicate register PR2. The operation uses the compare condition OD to detect if LSB is 1 or not. Note that the second output of the operation is ignored, since PR0 is a bit-bucket.

$$PR2,\ PR0 = CMPP.W.OD.UN.UN(GPR2, 0);$$

As mentioned in Section 8, compare-to-predicate operations are also used to move datum from a predicate register to another predicate register. As an example, the following code sets PR3 to the same value as PR2.

PR3, PR0 = CMPP.W.=.UN.UN(0,0) if PR2;

Since the result of the comparison is always true (*i.e.*, 1), the semantics of unconditional compare operations ensures that PR3 is set to the same value as PR2.

10 Memory system and load/store operations

10.1 Memory system

The memory hierarchy in the HPL-PD architecture consists of the following:

1. main memory,
2. second-level cache,
3. conventional first-level cache, and
4. a data prefetch or streaming cache at the first level.

The exact structure of each of the three caches depends upon the implementation and is not architecturally visible. The first-level and second-level caches may be structured either as a combined instruction and data cache or as separate instruction and data caches.

The data prefetch cache is used to prefetch large amounts of data having little or no temporal locality without disturbing the conventional first level data cache. In other words, the emphasis in the case of data prefetch cache is more on masking load latencies than on reuse. Accesses to the data prefetch cache don't touch the first-level cache. Many applications, *e.g.*, scientific computation, iterate over some or all elements of a large arrays. Furthermore, the computation has little or no reuse of accessed elements. In such cases, bringing the data through first-level cache may lead to thrashing. The accessed elements may replace other data that is reused often such as scalar variables in a loop, in which case these other data items will be loaded again and again. The data prefetch buffer is used to avoid such cache thrashing. Array elements are prefetched to data prefetch cache and then loaded from this cache, and these elements don't pollute the first-level cache. Typically, the data prefetch cache will be a fully associative cache much smaller in size than the first-level cache. Its size is determined by the total number of loads that can be in flight at the same time.

10.2 Latency specification and cache control directives

The architecture provides explicit control over the memory hierarchy. In this section, we describe the modifiers associated with load/store operations which are used to control the placement of data in the memory hierarchy as well as to specify load latencies. A discussion of how to use these modifiers in compiling programs and the expected benefits can be found in [25].

A load operation (except an LDV operation) has two modifiers, which are described below.

1. Latency and source cache specifier: This modifier is used by the compiler to indicate its view of where the data is likely to be found. Consequently, it also specifies the load latency assumed by the compiler. In the EPIC version of the architecture, the latency specification for a load operation is an integral part of the operation. Like other operations, load operations have architecturally visible latencies. That is, the result of a load operation

must be available at the specified time (in the program's virtual time) for a program to execute correctly. If the result is not available at the specified time, then the processor must be stalled for the execution to be correct. Unlike other operations, there are more than one class of load operations to accommodate different latencies needed to communicate with different levels in the memory hierarchy. In the superscalar model of the architecture, the latency specification has no bearing on the correct execution of a program.

2. Target cache specifier: This modifier is used by the compiler to indicate its view of the highest level in the memory hierarchy where the loaded data should be left for use by subsequent memory operations. This is a hint to the hardware and has no bearing on the correct execution of the program.

Store operations have only one modifier, namely, target cache specifier. As with the load operations, it is used by the compiler to specify the highest level in the hierarchy where the stored data should be left for use by the subsequent memory operations.

Each of these modifiers can be one of the following:

- V1 for data prefetch cache,
- C1 for first level data cache,
- C2 for second level data cache, and
- C3 for main memory.

10.3 Standard load and store operations

Table 14 describes the set of standard load and store operations. All these operations take a fully-resolved virtual memory address as an argument. For fixed-point (or integer) values, the operation repertoire include byte, half-word and word operations at all levels of the memory hierarchy. Note that fixed-point load/store operations can also be used to load control registers from the memory and to store control registers in the memory. For floating-point values, the repertoire includes both single-precision and double-precision operations, again at all levels of the memory hierarchy. All these operations have a predicate input that guards their execution. Load operations can be issued speculatively, but there are no speculative stores.

Table 14: Standard load and store operations

Opcode	Operation description	I/O description	Sp	Opcode semantics
L B | H | W V1 | C1 | C2 | C3 V1 | C1 | C2 | C3	Load GPR or CR byte, half-word, or word latency and source cache specifier target cache specifier	P? I : IC	Y	dest1 = Mem[src1](B|H|W)
FL S | D V1 | C1 | C2 | C3 V1 | C1 | C2 | C3	Load FPR single or double precision latency and source cache specifier target cache specifier	P? I: F	Y	dest1 = Mem[src1](S|D)
S B | H | W V1 | C1 | C1 | C3	Store GPR or CR byte, half-word, or word target cache specifier	P? I, ICL :	N	Mem[src1](B|H|W) = src2
FS S | D V1 | C1 | C1 | C3	Store FPR single or double precision target cache specifier	P? I, F:	N	Mem[src1](S|D) = src2

An important point to note is that byte and half-word load operations are "unsigned" loads in the sense that they zero-extend the loaded value to convert it into a 32-bit quantity. Explicit sign-extension using the appropriate EXTS operation (see Section 8) must be performed to use the loaded value as a signed value.

As described in Section 3.2, memory operations within an instruction are executed in an order that is consistent with their sequential left-to-right order of execution.

10.4 Post-increment load and store operations

Post-increment operations are similar to the standard load/store operations as far as their memory access behavior is concerned. However, they have the additional capability to compute new addresses for subsequent load/store operations. The new address is the sum of the accessed memory address and a displacement, which can be either a literal or a value stored in a GPR. The new address is deposited in the specified destination register, which may or may not be identical to the source address register. Table 15 describes post-increment load and store operations.

Table 15: Post-increment load and store operations

Opcode	Operation description	I/O description	Sp	Opcode semantics
LI B | H | W V1 | C1 | C2 | C3 V1 | C1 | C2 | C3	Load GPR or CR and increment byte, half-word, or word latency and source cache specifier target cache specifier	P? I, ICL : IC, I	Y	dest1 = Mem[src1](B|H|W) dest2 = src1 + src2
FLI S | D V1 | C1 | C2 | C3 V1 | C1 | C2 | C3	Load FPR and increment single or double precision latency and source cache specifier target cache specifier	P? I, ICL : F, I	Y	dest1 = Mem[src1](S|D) dest2 = src1 + src2
SI B | H | W V1 | C1 | C1 | C3	Store GPR or CR and increment byte, half-word, or word target cache specifier	P? I, ICL, ICL : I	N	Mem[src1](B|H|W) = src2 dest1 = src1 + src3
FSI S | D V1 | C1 | C1 | C3	Store FPR and increment single or double precision target cache specifier	P? I, F, ICL : I	N	Mem[src1](S|D) = src2 dest1 = src1 + src3

10.5 Prefetch via loading to register 0

Prefetch or non-binding load of a memory address to any cache level in the hierarchy can be done by simply loading the value to static register 0 in either a GPR file or a FPR file. Note that static register 0 is a bit-bucket in both GPR and FPR files.

10.6 Support for run-time memory disambiguation

Potential dependences between memory operations are often a limiting factor in exploiting the parallelism in EPIC or superscalar architectures. Specifically, a load operation is potentially dependent upon all stores that precede the load in a program and thus, cannot be scheduled before these stores. This makes it difficult to mask the latency of a load operation. Compile-time memory disambiguation may alleviate the problem. However, compile-time memory disambiguation is a difficult problem and often gives inconclusive results, especially in languages like C which make heavy use of pointers. The HPL-PD architecture provides a run-time memory disambiguation capability, which permits compilers to schedule a load before

potentially aliasing stores even if conclusive aliasing information is not available at compile-time. The run-time disambiguation mechanism is similar to the *memory conflict buffer* proposed by Chen in his thesis [13], which describes several possible implementations and contains some results on the utility of such a mechanism. Silberman *et al.* [15] also describe a similar mechanism.

At the architecture level, the support for run-time disambiguation consists of three related families of operations, called data speculative load (LDS), data verify load (LDV) and data verify branch (BRDV). As described in more detail in Section 10.6.2, an LDS operation is used in conjunction with either an LDV or a BRDV operation. An LDS-LDV pair is used to schedule a load before potentially aliasing stores. Both operations in a pair specify the same memory address and the same destination register. Moreover, they have the same width modifier (*i.e.*, byte, half-word, etc.). An LDS-BRDV pair permits not only a load but also operations that depend upon the loaded value to be scheduled before potentially aliasing stores. In this case, the BRDV operation is used to branch to a piece of compiler-generated compensation code.

10.6.1 LDS, LDV and BRDV operations

Table 16 lists the various forms of LDS and LDV operations. Table 17 describes BRDV operations and is an excerpt from Table 20, which describes all branch operations.

Table 16: Load operations related to run-time memory disambiguation

Opcode	Operation description	I/O description	Sp	Opcode semantics
LDS B | H | W V1 | C1 | C2 | C3 V1 | C1 | C2 | C3	Data speculative load to GPR byte, half-word, or word latency and source cache specifier target cache specifier	P? I : I	Y	dest1 = Mem[src1](B|H|W) Also see the text
FLDS S | D V1 | C1 | C2 | C3 V1 | C1 | C2 | C3	Data speculative load to FPR single or double precision latency and source cache specifier target cache specifier	P? I: F	Y	dest1 = Mem[src1](S|D) Also see the text
LDSI B | H | W V1 | C1 | C2 | C3 S1 | C1 | C2 | C3	Data speculative load to GPR and increment byte, half-word, or word latency and source cache specifier target cache specifier	P? I, ICL : I, I	Y	dest1 = Mem[src1](B|H|W) dest2 = src1 + src2 Also see the text
FLDSI S | D V1 | C1 | C2 | C3 V1 | C1 | C2 | C3	Data speculative load to FPR and increment single or double precision latency and source cache specifier target cache specifier	P? I, ICL : F, I	Y	dest1 = Mem[src1](S|D) dest2 = src1 + src2 Also see the text
LDV B | H | W	Data verify load for GPRs byte, half-word, or word	P? I : I	Y	See the text
FLDV S | D	Data verify load for FPRs single or double precision	P? I : F	Y	See the text

Table 17: Branch operations related to run-time memory disambiguation (excerpt from Table 20)

Opcode	Operation description	I/O description	Sp	Opcode semantics
BRDVI	Data verify branch for use with data speculative loads to GPRs	P? B, I :	N	See the text.
BRDVF	Data verify branch for use with data speculative loads to FPRs	P? B, F :	N	See the text.

To describe the ("micro") semantics of these operations, it is necessary to introduce the notion of *LDS log*. In this report, we describe the LDS log in an abstract way and only in as much detail as is necessary to explain the semantics of operations introduced in this section. There are several ways to implement the LDS log; see, for example, [13].

The LDS log records information about a subset of LDS operations that has been already issued. The size of the log, *i.e.*, the number of entries in the log, depends on the implementation and is not architecturally visible. Each entry in the log contains (at least) the following two fields to store the information about an LDS operation:

1. Target register: This field contains the register loaded by the operation.
2. Address: This field contains either the memory address referenced by the operation or a "syndrome" derived from the address. (See the discussion related to stores.)

In addition, there is a way to mark an entry as valid or invalid. Operations that either access or modify the LDS log include all the LDS, LDV and BRDV operations as well as the store operations.

Now, we describe the semantics of the three classes of operations introduced in this section as well as the action of store operations on the LDS log.

Data speculative load operations:

As is evident from the first two rows of Table 16, there is a data speculative load operation corresponding to each load operation described in Section 10.3. The source and destination specification for an LDS operation is identical to that for the corresponding load operation. Similarly, all the modifiers associated with these operations have the same meaning as in the case of load operations. The semantics of an LDS operation is as follows.

1. Like a load operation, the operation loads the datum from the specified memory address into its destination register.
2. In addition, the processor performs the following two actions on the LDS log. First, it invalidates any entry whose target register field is the same as the destination register of the LDS operation. Second, it *may* add the LDS operation to the log by storing the destination register and the memory address (or its syndrome) referenced by the LDS operation into an entry in the log. Whether the operation is added to the log or not depends upon how the log entries are managed, what happens when the log is full, etc. All these aspects of the LDS log are completely implementation dependent.

Like post-increment load operations, there are also post-increment LDS operations. These operations perform all the actions described above and, in addition, compute new addresses for subsequent memory operations. Note that post-increment LDS operations have two destinations.

In the case of a post-increment LDS operation, the destination register in the above description refers to the first destination, since it is the one that is being loaded from the memory.

Also note that all LDS operations listed in the table have a (control) speculative version and have a predicate input that guards their execution.

Action of store operations on the LDS log:

When a store operation executes, the processor performs the following actions on the LDS log. For each valid entry in the log, it checks the memory address (or the syndrome) stored in the entry and the memory address (or its syndrome) referenced by the store operation to see if the store operation potentially writes into a *physical* memory location that was accessed by the LDS operation corresponding to the entry. If the answer is yes, then the processor invalidates the entry. How addresses are compared depends upon the implementation. The only requirement is that the address comparison be safe in the sense that it must detect all cases in which the store operation and the LDS operation access a common physical memory location. A couple of important cases to note are as follows. First, if the system permits two different virtual addresses to map to the same physical memory location, then the address comparison must detect these cases. Second, it is possible that an LDS operation and a store operation access overlapping but not exactly the same locations. For example, the LDS operation reads a word from the memory and the store operation writes a byte within that word. Again, the address comparison must detect such aliasing cases. For these reasons, it may be necessary to use an inexact but conservative approach based on syndromes derived from virtual addresses.

Data verify load operations:

The last two rows of Table 16 lists the various forms of data verify load operations supported by the architecture. Broadly speaking, these operations behave like conditional load operations. Each of these operations has one source, which specifies a memory address, and one destination. Also, all these operations have a (control) speculative version and have a predicate input to guard their execution.

The execution of an LDV operation proceeds as follows:

1. The processor checks the LDS log to see if there is a valid entry whose target register field is identical to the destination register specified in the LDV operation. Note that there can be at most one such entry. If so, the processor *may* treat the operation as a NOP. Specifically, the processor need not update the destination register.
2. If there is no such entry, then the processor must update the destination register. Furthermore, it must ensure that the value deposited in the register is identical to the one returned by an appropriate type of load (byte, half-word, etc.) from the memory address specified in the operation. There are a number of ways to do this; the simplest being to issue a load to the memory. Note that an LDV operation has all the information, *i.e.*, the memory address, the width of the datum to fetch and the destination register, that is necessary to issue a load. It is the compiler's responsibility to ensure that these arguments to an LDV operations are identical to the ones given to the corresponding LDS operation.
3. Finally, the processor invalidates any valid entry whose target register field is identical to the destination register specified in the LDV operation.

We assume that the architectural latency of an LDV operation is less than the latency of a load from the first level cache and is comparable to the latency of, say, an integer add operation. The rationale is this. In the expected use of these operations, the first case described above will be the

predominant case and the second case is expected to occur very rarely. In the first case, the processor simply has to check if the LDS log contains an entry whose target register field matches the destination register of the LDV operation. This can be done in much less time than the time taken to execute a load operation. Note that if the second case applies, then the processor may need to stall to ensure that the destination register contains a valid result.

LDV operations don't have source and target cache specifiers. Also, there are no post-increment versions of these operations. Since LDV operations are expected to be used in conjunction with LDS operations, it suffices to provide these capabilities only with LDS operations.

Data verify branch operations:

The natural place to describe these operations is the section on the branch architecture (Section 11). However, these operations are closely tied to the run-time disambiguation mechanism, and thus, we describe them in this section. Unfortunately, that creates a forward reference that is hard to avoid. To fully understand these operations, it is necessary to understand how branches are performed in the HPL-PD architecture, for which we refer the reader to Section 11.

Table 17 lists the two forms of data verify branch operations supported by the architecture. These operations behave like conditional branches and are used to branch to a piece of compiler-generated compensation code to ensure the correct execution of a program. Each of these operations have two sources. The first source is a branch target register (BTR) containing the target address. The second source is a general-purpose or a floating-point register and is used as a key to find a matching entry in the LDS log.

The execution of a data verify branch proceeds as follows:

1. The processor checks the LDS log to see if there is a valid entry whose target register field is identical to the second source register specified in the BRDV operation. Note that there can be at most one such entry. If so, the processor *need not* branch to the target address.
2. If there is no such entry, then the processor does branch to the target address.
3. Finally, the processor invalidates any valid entry whose target register field is identical to the second source register specified in the BRDV operation.

Note that these operations cannot be issued speculatively, but they do have a predicate input to guard their execution.

Both LDV and BRDV operations invalidate matching entries and thus, to a limited extent, provide a programmatic way to manage entries. However, that is not sufficient because the number of entries in the LDS log is not architecturally visible. A point to note is that the expected usage of these operations doesn't require that an LDS operation has a matching LDV operation on all execution paths. This may happen, for example, if the LDS operation has been scheduled (control) speculatively. Thus, the LDS log may contain entries that are no longer useful on an execution path, and the implementation should provide a way (*e.g.*, LRU tags) to expunge such entries.

10.6.2 Expected usage and compiling issues

We explain the expected usage of the three classes of operations introduced in the last section using a simple example. The left column below shows the unscheduled code for the example. To simplify the exposition, we omit modifiers associated with operations and assume that all

registers are general-purpose registers. The right column shows a naive schedule assuming the following latencies for operations: 2 cycles for loads, 1 cycle for adds and 1 cycle for stores.

Original code for the example	**Schedule**	
	Cycle	Instruction
r1 = L(a1); / * First load * /	1	r1 = L(a1);
r2 = ADD(r1, r5);	2	– – – –
S(a2, r2); / * Store * /	3	r2 = ADD(r1, r5);
r3 = L(a3); / * Second load * /	4	S(a2, r2);
r4 = ADD(r3, r5);	5	r3 = L(a3);
S(a4, r4);	6	– – – –
	7	r4 = ADD(r3, r5);
	8	S(a4, r4);

The naive schedule can be improved by issuing the second load in an earlier cycle, *i.e.*, before the store that preceded the load in the original program. Reordering a store-load pair, however, requires a compile-time proof they don't reference the same memory address. As mentioned earlier, compile-time memory disambiguation is a difficult problem and often gives inconclusive results. If the compile-time analysis fails, then the schedule given above is the best a compiler can do.

The run-time disambiguation mechanism permits efficient code generation in cases where the compile-time analysis fails. Consider the above example. To schedule the second load before the store, we replace the load by two operations, an LDS operation and an LDV operation. The resulting code is given below. Note that both operations have the same source and destination as the original load operation.

Code with the load replaced by an LDS-LDV pair
r1 = L(a1);
r2 = ADD(r1, r5);
S(a2, r2);
r3 = LDS(a3);
r3 = LDV(a3);
r4 = ADD(r3, r5);
S(a4, r4);

In general, it is the compiler's responsibility to ensure that the parameters specified in an LDV operation, *i.e.*, the destination register, the memory address and the width of the datum to fetch, are identical to those specified in the corresponding LDS operation. Note that the address register specified in an LDV operation need not be the same as the one specified in the corresponding LDS operation; the only requirement is that the memory address supplied to the two operations be the same.

The LDS operation is not constrained by the preceding store even if they alias and can be scheduled before the store. On the other hand, the LDV operation must obey the following scheduling constraints in order to ensure that the subsequent operations dependent upon the destination register get the correct value. The first constraint is that the LDV operation must be scheduled so that it is effectively issued after the store operation. Note that, since memory operations in an instruction are prioritized from left to right (see Section 3.2), the LDV operation can be issued in the same cycle as the store operation. The second constraint relates to when the LDV operation can be scheduled relative to the LDS operation. For each machine, its mdes specifies the minimum number of cycles that must be used to separate an LDV operation from the corresponding LDS operation. If an LDV operation is scheduled closer than the specified minimum distance, the destination register will contain a non-deterministic value.

The following is a schedule for the code given above assuming that the LDV operation has a latency of 1 cycle.

Schedule for code with LDS and LDV operations	
Cycle	Instruction
1	r1 = L(a1);
2	r3 = LDS(a3);
3	r2 = ADD(r1, r5);
4	S(a2, r2); r3 = LDV(a3);
5	r4 = ADD(r3, r5);
6	S(a4, r4);

It is instructive to go through the execution of the above program in detail. There are following two cases to consider.

1. Assume that, in the original program, the second load doesn't alias with the preceding store. In this case, the LDS operation loads the correct value in register r3; and if things go right, the LDV operation behaves like a NOP. The detailed execution, in this case, proceeds as follows. The LDS operation loads the register. In addition the processor adds the LDS operation to the log. When the store executes, the processor doesn't invalidate the entry, since the store and the LDS operation don't alias. When the LDV operation executes, it finds the matching entry in the log, which guarantees that there were no aliasing stores. The LDV operation, then, performs no action on the register r3.

   The other case to consider is when either the processor doesn't add the LDS operation to the log or the entry is replaced before the LDV operation executes. In this case, the LDV operation doesn't find the matching entry and effectively re-issues the load, since it cannot ensure that there were no potentially aliasing stores.

2. Now assume that the load and the store do potentially alias. When the store executes, the processor invalidates all entries that potentially alias with the store. This guarantees that the LDV operation will not find a matching entry, and the LDV operation will load the correct value in the register r3.

Thus, the correct value is available in the register r3 in all cases after the execution of the LDV operation.

Next, we discuss a slightly different issue, namely, register spill. In the usage described above, the final value in the destination register may come from either the LDS operation or the LDV operation. Thus, the lifetime of the value extends all the way from the LDS operation to its uses. However, the register contains an indeterminate (and a potentially incorrect) value from the time the LDS is executed to the time the corresponding LDV is executed. So the question is whether the register can be spilled in the usual way or not during this time period. The semantics of these operations is such that the register can be spilled in the usual way and reused for other computation including as a target of another LDS-LDV pair. As an example, consider the code shown in the left column below. The right column shows the code after register allocation. Register r1 has been assigned to both vr1 and vr2. Furthermore, the register r1 has been spilled after the LDS operation; spill-addr is the memory address to which r1 has been spilled.

Code before register allocation	**Code after register allocation with spill code inserted**
vr1 = LDS(a1) / * First LDS * /	r1 = LDS(r2) / * First LDS * /
vr2 = LDS(a2) / * Second LDS * /	S(spill - addr, r1);
⋮	r1 = LDS(r3) / * Second LDS * /
vr2 = LDV(a2)	⋮
⋮	r1 = LDV(r3)
vr1 = LDV(a1)	⋮
	r1 = L(spill - addr)
	r1 = LDV(a1)

The code in the right column produces the correct result. The important point to note is that the entry corresponding to the second LDS will be removed by the corresponding LDV (*i.e.*, the first LDV) and will not be visible to the last LDV. Thus, the last LDV effectively re-issues a load to r1. The above scenario also works for the case when r1 is used for other types of computation. In that case, the load restores the contents of r1 and the LDV operation ensures that the effect of potentially aliasing stores is captured. Although the usual spill strategy works, it is not the most efficient one. There are more efficient strategies; for example, the compiler can simply delay the first LDS operation. This is not specific to LDS and LDV operations and applies equally well in the case of ordinary load operations.

Although the above discussion assumes straight-line code, branches and program merges don't present additional problems. An LDS operation can be scheduled before a branch, in which case it becomes a (control) speculative operation. Similarly, it can be scheduled before a merge point, in which case it is replicated on all execution paths merging at that point. In general, if each execution path leading to an LDV operation satisfies the following constraints, then the value stored in the destination register will be correct. First, there is a corresponding LDS operation that targets the same register. Second, there are no other writes to the register between the LDS and LDV operation unless the register is spilled in the way described above.

Several points to note about the use of these operations:

1. The code sequence given below produces a non-deterministic result in r1. The value in r1 is either the result of the add or the value deposited by the LDV operation; it all depends

upon the state of the LDS log at the time the LDV operation is executed. The expected usage described above precludes such uses.

```
r1 = LDS(a1);
     ⋮
r1 = ADD(r2, r3);
     ⋮
r1 = LDV(a1);
```

2. In the following code sequence, the second LDS effectively masks the first LDS. That is, a subsequent LDV operation never sees the log entry corresponding to the first LDS.

   ```
   r1 = LDS(a1);
   r1 = LDS(a2);
   ```

   Now, consider the following code. In this case, the second LDV always re-loads the register, since the first LDV invalidates any matching entry.

   ```
   r1 = LDV(a1);
   r1 = LDV(a2);
   ```

3. As described earlier, nested pairs of LDS-LDV operations targeting the same register produce correct result, though in an inefficient way. On the other hand, non-nested uses may produce unexpected and potentially incorrect result.

The run-time disambiguation mechanism also supports a more general form of code motion. Not only a load operation but also operations that depend upon the loaded value can be scheduled before preceding stores that may alias with the load. The general form of code motion uses BRDV operations instead of LDV operations to ensure the correct execution of the program. For an LDS operation, the corresponding BRDV operation detects if a store does alias with the LDS operation. If so, then it transfers the control to a piece of compiler-generated compensation code.

Consider the example given at the beginning of this section. We can transform the original code into the code shown in Figure 1. The figure shows the code as a control-flow graph in order to clearly illustrate the relation between various parts of the code. The transformation replaces the original load with LDS and BRDVI operations and splits the original code into three basic blocks. The basic block labeled "before" contains the code before the original load operation as well as the LDS and BRDVI operations, and the basic block labeled "after" contains the code after the original load operation. The basic block labeled "comp" contains the compensation code. The compensation code re-issues the load and then branches back to the code after the original load. The BRU operation in the compensation code performs an unconditional branch (see Section 11).

The transformed code can be scheduled in the usual way as long as the scheduler recognizes the following. First, the LDS operation is not constrained by the preceding store. Second, the BRDVI operation is constrained by both the preceding store and the LDS operation. Third, the compensation code is rarely executed and should be treated as such.

Operations that are flow-dependent upon the LDS operation can be scheduled before the preceding store as long as all other scheduling constraints are met. Consider, for example, the add operation in the basic block labeled "after". The operation is flow-dependent on the load, but

doesn't depend on the store. Thus, this operation can be scheduled before the store. To do so, the operation must be moved before the BRDVI operation. Code motion across a BRDV operation follows the standard rules of code motion across a branch. For example, if the add operation is moved before the BRDVI operation, then it must be replicated in the compensation code. Moreover, the values of its operands must be preserved (not necessarily in the same registers) in order to execute the corresponding operation in the compensation code.

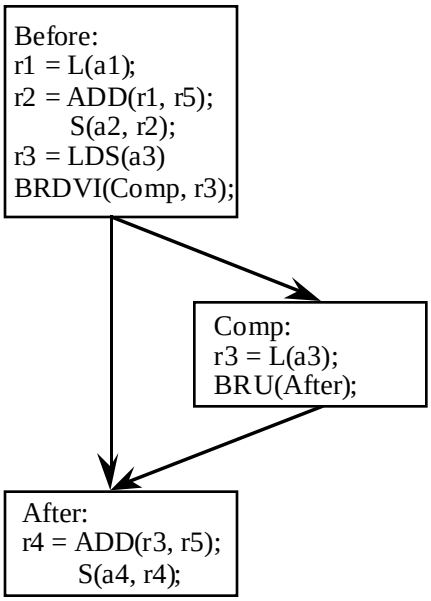

Figure 1: Transformed code illustrating the use of LDS and BRDV operations

The following is a schedule for the transformed code assuming that branches have a latency of 1 cycle. Note that both the LDS operation and the add operation in the basic block labeled "after" are scheduled before the preceding store.

General form of code motion

Main code		Compensation code	
Cycle	Instruction	Cycle	Instruction
	Before:		Comp:
1	r1 = L(a1); r3 = LDS(a3);	1	r3 = L(a3);
2	– – – –	2	– – – –
3	r2 = ADD(r1, r5); r4 = ADD(r3, r5);	3	r4 = ADD(r3, r5);
	/ * The second add should be marked	4	BRU(After);
	speculative to defer exceptions. * /		/ * Branch back * /
4	S(a2, r2); BRDVI(Comp, r3);		
	/ * Branch to compensation code * /		
	After:		
5	S(a4, r4);		

An important point to note relates to exceptions. Since an LDS operation is executed before potentially aliasing stores, it may return a value that is programatically incorrect. Thus, operations that use the loaded value may produce spurious exceptions, that is, exceptions that would not be generated in the original program. Such spurious exceptions should not be signaled. It is the compiler's responsibility to ensure that the signaling of any exception due to the use of the value returned by an LDS operation is deferred until the corresponding BRDV operation executes and doesn't take. The speculative execution mechanism described in Section 3.4 provides the necessary architectural support.

We conclude this section with several comments. The first comment relates to the action taken on the LDS log at context-switches and procedure calls. At a context-switch, the implementation has two options. Either it can save-restore the log or it can simply invalidate all entries, in which case the appropriate registers will be re-loaded by the corresponding LDV operations. In the case of a procedure call, we assume that no action is taken on the LDS log, and it is the compiler's responsibility to ensure that a program executes correctly in all cases. A simple way to do this is to never split an LDS-LDV pair (or an LDS-BRDV pair) across a procedure call.

Second, it may appear that an LDS-LDV pair serves the same purpose as a prefetch to first level cache followed by a load from the first level cache. The difference is this. The prefetch-load scheme is unable to mask the latency of the first level cache, since the second load is a load from the first level cache. The latency of an LDV operation, on the other hand, is less than the latency of a load from the first level cache and is comparable to the latency of, say, an integer add operation.

Finally, the run-time disambiguation mechanism is useful only in cases where there is a very high probability that the original load operation doesn't alias with preceding stores. If that is not the case, then its use may actually degrade the performance. Thus, its effective use requires a way, *e.g.*, analysis, profiling, user-directives, to get "probabilistic" aliasing information.

10.7 Memory operations to save/restore and spill registers

This section describes load/store operations that are used for the following two purposes. First, they are used to save/restore registers as part of the procedure calling convention. Second, they are used to spill registers to memory and then to reload them back. The standard load/store operations are not suitable for this purpose because each register in HPL-PD has an associated speculative tag bit. Like the datum part of a register, the speculative tag bit also must be saved/restored. More importantly, the semantics of the standard load/store operations are not what is needed to save/restore registers. As described in Section 3.4, speculative tag bits play an important role in the semantics of these operations, since the speculative tag bit of a register determines whether the register holds a valid datum or not. For example, the standard store operation signals an exception if the speculative tag bit of the register being stored is set. This is certainly not the semantics we would like in order to save a register to memory, since we simply want to save the register regardless of the validity of the datum stored in the register. In other words, the memory operation used to save a register to memory shouldn't examine the speculative tag bit of the register being stored. Similarly, the operation used to restore registers from memory shouldn't set the speculative tag bit of the register being loaded. Having separate opcodes to load the value part of a register makes it possible to load the value part and the tag bit in parallel or in any order.

Table 18 lists the memory operations used to save and restore registers. All these operations have a predicate input that guards their execution. Restore operations can be issued speculatively, but,

like standard store operations, save operations don't have a speculative version. The operations described in Table 18 are used to save the datum part of a general-purpose, floating-point, or branch target register. As mentioned earlier, the speculative tag bit associated with a register is saved/restored separately.

There are no operations that can be used directly to save/restore predicate registers and speculative tag bits. The next section describes ways to save/restore them efficiently.

Table 18: Memory operations to save/restore (or spill /unspill) registers

Opcode	Operation description	I/O description	Sp	Opcode semantics
SAVE	Save a GPR or CR to memory.	P? I, IC :	N	Mem [src1] = src2 Don't look at the speculative tag bit of src2, i.e., the register being stored. The tag bit for src1 is treated as in the case of a standard store operation.
FSAVE	Save a FPR to memory.	P? I, F :	N	Mem [src1] = src2 Don't look at the speculative tag bit of src2, i.e., the register being stored. The tag bit for src1 is treated as in the case of a standard store operation.
BSAVE	Save a BTR to memory.	P? I, B :	N	Mem [src1] = src2 Don't look at the speculative tag bit of src2, i.e., the register being stored. The tag bit for src1 is treated as in the case of a standard store operation.
RESTORE	Restore a GPR or CR from memory.	P? I : IC	Y	dest1 = Mem [src1] Don't touch (i.e., set or clear) the speculative tag bit of the destination register. The tag bit for src1 is treated as in the case of a standard load.
FRESTORE	Restore a FPR from memory.	P? I : F	Y	dest1 = Mem [src1] Don't touch (i.e., set or clear) the speculative tag bit of the destination register. The tag bit for src1 is treated as in the case of a standard load.
BRESTORE	Restore a BTR from memory.	P? I : B	Y	dest1 = Mem [src1] Don't touch (i.e., set or clear) the speculative tag bit of the destination register. The tag bit for src1 is treated as in the case of a standard load.

10.7.1 Code schemas to save/restore and spill predicate registers and speculative tag bits

In this section, we describe code schemas to save/restore as well as to spill predicate registers and speculative tag bits. The discussion is phrased in terms of predicate registers, but the schemas described in this section, with some of the opcodes changed appropriately, also apply to speculative tag bits. Please refer to Tables 5 and 6 in Section 8 for the semantics of various move operations used in the code schemas in this Section.

Save/restore schema

First, we discuss how to save/restore predicate registers, for example, as part of the procedure calling convention. A simple but inefficient approach is to save/restore predicate registers one at a time using the schema described below. In the code, px is the predicate register being saved/restored, addr is the address of the memory location assigned to hold px, and gpry is a gpr register used as a temporary. In the code used to restore a predicate register, one can use either a MOVEGBP or a compare-to-predicate operation to move the least significant bit (LSB) of the GPR to the predicate register.

Code to save a predicate register	**Code to restore a predicate register**

addr = Address where to save;
gpry = MOVEPG(px);
= SAVE(addr, gpry);
	addr = Address from where to restore
	gpry = RESTORE(addr);
	px = MOVEGBP(gpry, 0);
	(or px = CMPP.EQ.CN.CC(gpry, 1);)

Although this schema produces the intended results, it is quite inefficient because it requires three operations for each 1-bit register. Therefore, HPL-PD provides architectural support for a more efficient approach--it allows access to predicate registers not only one register at a time but also in groups of 32 using control registers aliases. It is, therefore, possible to save/restore 32 registers at a time giving a 32-fold reduction in the number of operations as compared to the above approach.

Suppose that we want to save registers numbered from x to y. Since x and y may not be aligned on 32-bit boundary, we must save/restore only the required registers. While saving the registers to memory, it is not essential that we save only the required ones, since saving some extra registers will not lead to incorrect result. On the other hand, we must restore only the required registers and ensure that no other registers are modified, since that may lead to incorrect results.

The code to save the predicate registers numbered from x to y is given below. In the code, s = floor(x/32) and n = ceiling(y/32). The code uses register aliases of the form PV(i, j) to refer to the j^{th} group of 32 register in the i^{th} predicate register file, which were introduced in Section 4.7. Note that if x and y are not aligned at 32-bit boundary, the code will save some extra registers before x and after y.

Code to save predicate registers in groups of 32
$addr_s$ = Address where to save the first 32 registers in the range; SAVE($addr_s$, PV(1, s)); $addr_{s+1}$ = Address where to save the next 32 registers; SAVE($addr_{s+1}$, PV(1, s+1)); … $addr_n$ = Address where to save the last 32 registers; SAVE($addr_n$, PV(1, n));

To illustrate the code to restore predicate registers, first suppose that x and y are aligned at 32-bit boundary. Then the code is similar to the code for saving registers and is as follows:

Code to restore predicate registers in groups of 32 assuming alignment at 32-bit boundary

```
addr_s = Address where the first 32 registers in the range are saved;
PV(1, s) = RESTORE(addr_s );
addr_s+1 = Address where the next 32 registers are saved;
PV(1, s+1) = RESTORE(addr_s+1);
…
addr_n = Address where the last 32 registers are saved;
PV(1, n) = RESTORE(addr_n);
```

On the other hand, if x and/or y are not aligned at 32-bit boundary, then the first and/or the last restore operation in the code given above is replaced by a two operation sequence in which the first operation loads 32 bits in a GPR and the second operation writes into the appropriate predicate registers. For example, suppose x is not aligned on a 32-bit boundary. Then the first two operations above are replaced by the following code. To restore registers selectively, the MOVEGCM operation takes a bit-mask that specifies the registers that should be restored.

Code to restore predicate registers that are not aligned at 32-bit boundary

```
addr_s = Address where the first 32 registers in the range are saved;
gpry = RESTORE(addr_s );
PV(1, s) = MOVEGCM(gpry, mask, PV(1, s));
```

Spill schema

Code schemas to spill/unspill predicate registers are similar to the ones used to save/restore predicate registers. As was the case with save/restore schemas, there are two ways to spill predicate registers. First, we can spill/unspill one register at a time. Second, we can optimize the spill code and try to spill/unspill multiple registers in one go. Since indices of registers that need to be spilled are rarely contiguous, it may not be that easy to optimize spill code by spilling multiple registers simultaneously. In both schemes, the code to spill is identical. We simply store the corresponding control register alias(s) to memory. Suppose we want to spill the predicate register px, *i.e.*, the register with index x. Then, assuming n = floor (x/32), the spill code is given below.

Code to spill one or more predicate registers

```
addr = Address where to save the register;
SAVE (addr, PV(1, n));
```

The code spills not only the register px, but also other registers "around" px. As pointed out earlier, spilling some extra registers will not lead to incorrect result. On the other hand, we must unspill only the required register and ensure that no other registers are modified because that may lead to incorrect results. Note that the same code can also be used to spill multiple registers in one step; for example, the above code spills all predicate registers between floor(x/32) × 32 to ceiling(x/32) × 32 – 1.

Two different code schemas to unspill predicate registers are given in the table below. The three operation sequence in the left column unspills a single predicate register, p34 in this case. Note that p34 is the same as the bit at offset 2 in the control register PV(1, 2). MOVEGBP extracts the bit at offset 2 in gpr23 and stores it in p34. Like the restore schema discussed in the last section, the three operation sequence in the right column can be used to unspill multiple predicates in one step by specifying an appropriate mask in the MOVEGCM operation.

Code to unspill a predicate register	**Code to unspill multiple predicate registers**

addr = Address where p34 (*i.e.*, control register PV(1, 2)) is spilled
gpr23 = RESTORE(addr);
p34 = MOVEGBP (gpr23, 2);
	addr = Spill address;
	gpr23 = RESTORE(addr);
	PV(1, i) = MOVEGCM (gpr23, mask);

11 Branch architecture

Branch operations are of major concern in designing ILP architectures because branches tend to interrupt the smooth flow of instructions and thus degrade the performance. The branch mechanism described in this report is a preliminary attempt to address the efficient implementation of branches. It permits different pieces of the information related to a branch to be specified as soon as they become available in the hope that the information can be used to reduce the adverse effect of the branch, *e.g.*, by prefetching instructions from the potential branch target. In these respects, it is similar to the one first proposed by IBM's Stretch project [9] and subsequently by Young and Goodman [10]. However, we believe that the efficient implementation of branch operations in ILP architectures is an area wide open for research. Issues to be investigated include policies for instruction prefetch and instruction cache management, efficient handling of multiple branches in a cycle, *etc*. The branch mechanism described in this report is simply a starting point to enable the research in this area. We expect it to evolve as more experience is gained.

In the HPL-PD architecture, a branch is performed in multiple steps--three steps for a conditional branch and two steps for unconditional branches. The steps involved in a conditional branch are as follows:

1. Specification of the target address: Prepare-to-branch (PBR) operations described in Section 11.1 are used for this purpose. This step specifies the branch target address in advance of the branch point allowing a prefetch of instructions from the target address. Hints to control instruction prefetch can also be specified at this step. The architecture

provides separate 64-bits wide branch-target registers to store the information provided at this step.

2. Computation of branch condition: The branch condition is stored in a predicate register and is computed by an appropriate type of compare-to-predicate operation (see Section 9).

3. Transfer of control: Branch operations perform the actual transfer of control if the branch is taken. The architecture provides several types of branch operations including unconditional and conditional branches, branch and link for subroutine calls, and special branch operations to support software pipelining of loops. Branch operations are discussed in Section 11.2.

An unconditional branch (or a jump) doesn't involve the computation of the condition, and thus, consists of only the first and last steps.

As discribed in Section 3.2, the architecture permits multiple branch operations in an instruction. Moreover, the latency of a branch operation is an architectural parameter that is specified for each machine in its mdes. A significant point to note is that a branch takes effect after exactly n cycles where n is the latency of the branch. That is, branch operations always have the "equals" semantics, even in "less-than-or-equals" machines. Branch operations are executed in a parallel pipelined manner with the following implications. First, consider the execution of an instruction containing multiple branch operations. In this case, the result is well-defined only when at most one operation takes the corresponding branch. If more than one branch operations specify that the corresponding branches be taken, then the result of the execution is undefined. It is the compiler's responsibility to ensure that, in an instruction, at most one branch takes. Second, branch operations in the delay slots of a branch are executed in a normal way, *i.e.*, they are not squashed.

11.1 Prepare-to-branch operations (PBRR, PBRA)

There are two prepare-to-branch operations: PBRR and PBRA. Their semantics are identical except for the way they interpret target address. PBRR is used to prepare branches to PC-relative addresses, whereas PBRA is used to prepare branches to absolute addresses. Both these operations specify a target BTR, which is used to communicate the information from a prepare operation to the corresponding branch operation. The arguments to these operations are as follows:

1. Target address: This can be either a literal or a GPR containing the address. The target address must be the starting address of an instruction. There is no provision to branch to an operation in the middle of an instruction.

2. Static branch prediction: This is a 1-bit literal with the following interpretation: a value of 1 means the branch is predicted taken, 0 means it is predicted not taken. It is a hint to control the instruction prefetch and its use is machine-dependent.

These operations compute the effective address, and store the computed address and the static prediction into the specified BTR.

Table 19 lists the prepare-to-branch operations. Both these operations have a predicate input that guards their execution, and they can be issued speculatively.

Table 19: Prepare to branch operations

Opcode	Operation description	I/O description	Sp	Opcode semantics
PBRR	Prepare a branch to a PC-relative address	P? IL, L : B	Y	dest1 = {PC + src1, src2}
PBRA	Prepare a branch to an absolute address	P? IL, L : B	Y	dest1 = {src1, src2}

11.2 Branch operations

Branch operations perform the actual transfer of control. All the branch operations are listed in Table 20 and are described in some detail in the subsequent subsections.

Each branch operation specifies a source BTR, which contains the branch target address. Most branch operations are predicated, that is, they have a predicate input that guards their execution. Exceptions are the branch operations used in the software-pipelining of loops; their execution cannot be nullified. Predicated branch operations simplify the code generation in many cases. Note, however, that predicated branches are not a necessity as the same effect can be obtained using un-predicated branches. Branch operations cannot be issued speculatively.

Table 20: Branch operations

Opcode	Operation description	I/O description	Sp	Opcode semantics
BRU	Unconditional Branch (jump)	P? B :	N	PC = src1.address
BRCT	Conditional branch on predicate = 1 (true)	P? B, P :	N	if src2 then PC = src1.address
BRCF	Conditional branch on predicate = 0 (false)	P? B, P :	N	if !src2 then PC = src1.address
BRL	Branch and link	P? B : B	N	dest1 = {return address, default value} PC = src1.address
BRLC	Branch on zero loop count and decrement loop count	P? B :	N	if LC != 0 then LC = LC -1 PC = src1.address
BRF B | F B | F B | F	To support software-pipelining of counted loops Continue_dir modifier Ramp_dir modifier Stop_dir modifier B means branch; F means fall-through	B : P	N	See the text.
BRW B | F B | F B | F	To support software-pipelining of loops with exit Continue_dir modifier Ramp_dir modifier Stop_dir modifier B means branch; F means fall-through	B, P, P : P	N	See the text.
BRDVI	Data verify branch for use with data speculative loads to GPRs	P? B, I :	N	See the text.
BRDVF	Data verify branch for use with data speculative loads to FPRs	P? B, F :	N	See the text.

11.2.1 Unconditional branch operation (BRU)

This operation jumps to the target address stored in the specified BTR.

11.2.2 Conditional branch operations (BRCT, BRCF)

There are two forms of conditional branch operations: BRCT and BRCF. The first branches if the branch condition is true; the second if the branch condition is false. The branch condition is available in the specified predicate register. Having both operations seems unnecessary as one can simulate the other. However, having both forms makes it easy to perform certain optimizations, *e.g.*, re-ordering of basic blocks so that the most likely path is the fall-through path.

11.2.3 Branch and link (BRL)

This operation is used for subroutine call. In addition to transferring the control to the callee, it also prepares the branch that returns to the caller. The operation specifies a source BTR and a target BTR. First, the operation computes the return address and stores it in the address field of the target BTR. The branch prediction field of the target BTR is set to some default value. Then, it transfers control to the address specified in the source BTR.

To return to the caller, the callee simply executes an unconditional branch using the BTR prepared by the corresponding branch and link operation. By convention, compilers may use a specific register as the return register.

11.2.4 Branch on loop count (BRLC)

It is used to close a counted loop that has not been software-pipelined. If the control register LC contains a non-zero value, then it decrements LC and transfers the control to the specified address.

11.2.5 Branch operations to support software-pipelining of loops (BRF and BRW)

This section provides a brief description of the branch operations provided to support software pipelining of both counted loops and loops with exits. The architecture provides two classes of operations, called BRF and BRW. Operations in the BRF class are used in the case of counted loops (*e.g.*, DO, For), whereas operations in the BRW class are used for loops with exits (*e.g.* While, Repeat-until). The code-schema paper by Rau *et. al.* [26, 27] contains a detailed description of the semantics of these operations and describes how they are used in various code schemas for software-pipelining of loops. See also [28] for software-pipelining of loops with exits.

BRF class of operations:

Three modifiers, called *continue_dir*, *ramp_dir* and *stop_dir*, are used to specify operations in this class. Each of these modifiers can be one of the following: B for branch and F for fall-through. These modifiers determine the sense of the branch, i.e., whether to take a branch or fall-through, at three distinct phases in the execution of a software-pipelined loop. This parameterization is used to accommodate distinct branch requirements for forward and backward branches and to accommodate differing strategies with respect to the flow of control within the prolog, kernel and epilog portions of a software-pipelined loop.

Each of these operations specify one source and one destination. The source is a BTR containing the branch address. The destination is a predicate register that is used to enable or disable the next (source) iteration. Note that these operations execute unconditionally, *i.e.*, they don't have a predicate input to guard their execution. Figure 2 describes the detailed semantics of these operations.

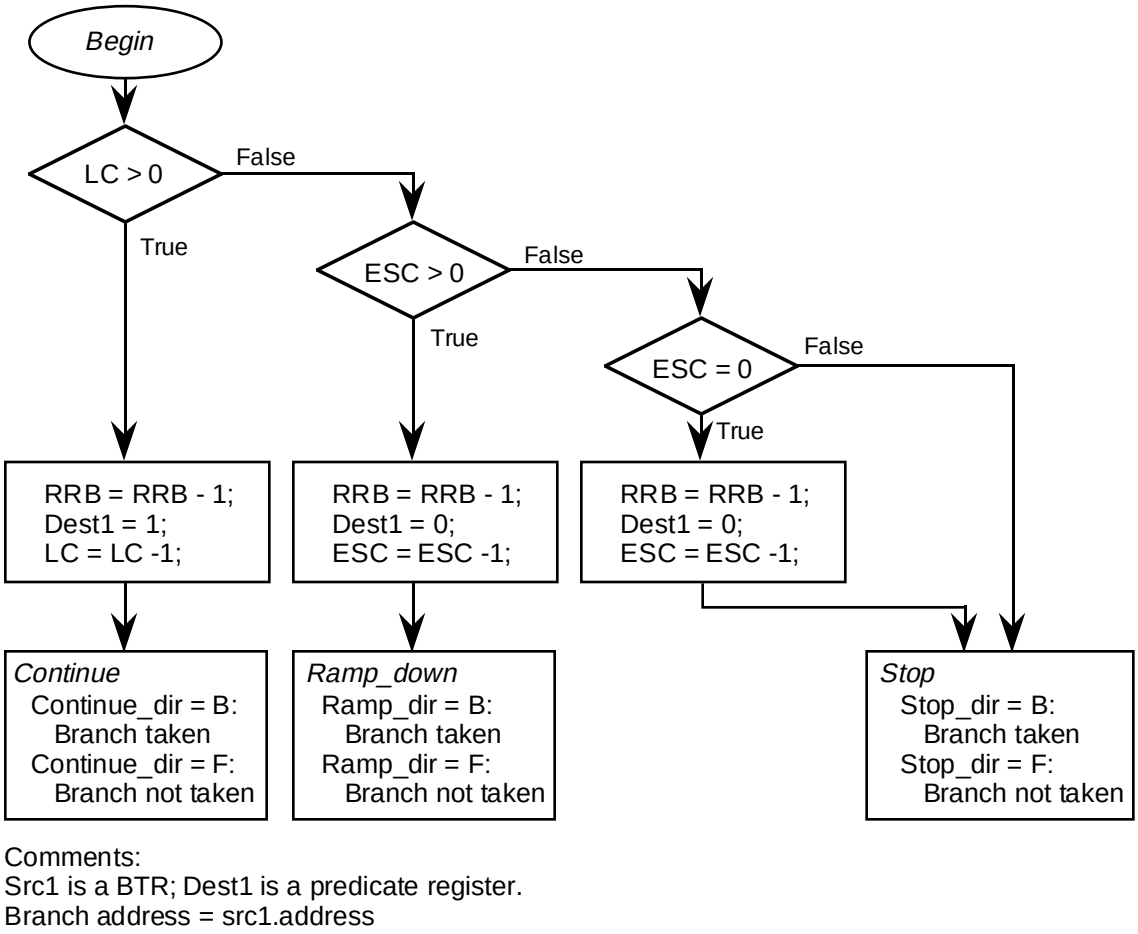

Figure 2: Semantics of BRF class of operations

If the loop count (LC) is not zero, then a new loop iteration is initiated. In this case, a BRF operation follows the path from the *begin* node to the node marked *continue***.** That is, it performs the following actions:

1. Decrements the rotating register base (RRB) to implement the dynamic single assignment (see [29]).
2. Sets the destination predicate register to 1 in order to enable the first stage of the new iteration.
3. Decrements the loop count.
4. Starts the new iteration either by branching to the specified address or by falling through. The modifier continue_dir determines the sense of the branch (*i.e.*, branch or fall through).

If LC is equal to or less than zero, the loop is executed an additional ESC number of times to drain the pipeline. This is achieved in a way similar to the normal execution of the loop except that the destination predicate is set to 0 in order to suppress the issuing of a new loop iteration. See the path from the *begin* node to the node marked *ramp_down*. Note that, in this case, ramp_dir determines the sense of the branch.

If ESC is equal to or less than zero, the execution of the loop is complete. In this case, a BRF operation follows one of the paths from the *begin* node to the node marked *stop*, and the modifier stop_dir determines the sense of the branch.

Although the architecture provides all combinations of the three modifiers, the combinations listed in Table 21 are the ones used in various code schemas described in [26]. The full generality is provided to facilitate research in this area.

Table 21: Useful BRF operations

Continue_dir	Ramp_dir	Stop_dir
B	B	F
B	F	F
F	B	B
F	F	B
F	F	F

Historically, the first operation in the table has been referred to as Brtop, and the last as Nexti. Strictly speaking, Nexti is not a branch operation as it always falls-through to the next instruction. However, its semantics is so similar to other BRF operations that it is treated like a branch operation.

BRW class of operations:

These operations are similar to the operations in the BRF class, and the last section contains more details about the modifiers used to specify these operations. Each of these operations specify three sources and one destination. The first source is a BTR containing the branch address. The second source is a predicate register, and it is used to determine if the previous iteration corresponds to a source iteration or to the epilog part of the loop. The third source, again a predicate register, contains the result of the compare operation that determines whether to exit the loop or not. The destination is a predicate register that is used to enable or disable the next (source) iteration. The second source and the destination are related in the sense that the destination of the branch operation in one iteration is the source of the branch operation in the next iteration. Note that these operations execute unconditionally, *i.e.*, they don't have a predicate input to guard their execution.

The detailed semantics of these operations is described in Figure 3 and is similar to the semantics of BRF operations. The main difference is that the LC > 0 condition is replaced by conditions that check the source predicates in order to determine whether to issue a new iteration or not.

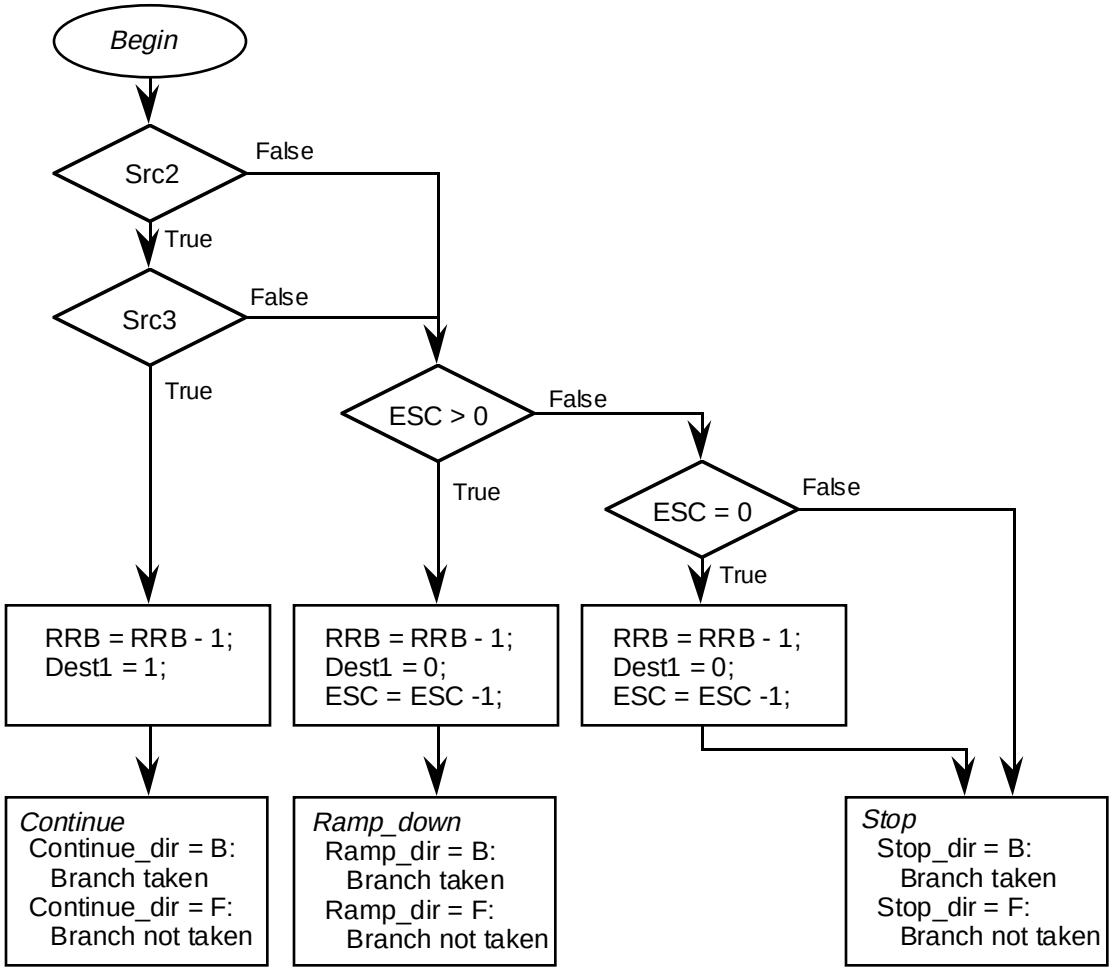

Figure 3: Semantics of BRW class of operations

11.2.6 Branch operations used in run-time disambiguation mechanism (BRDVI, BRDVF)

These operations are part of the run-time disambiguation mechanism described in Section 10.6, which contains a detailed description of these operations. BRDVI is used in conjunction with data speculative loads to GPRs, and BRDVF is used with data speculative loads to FPRs.

11.3 Usage and compiling issues

Prepare-to-branch operations are amenable to some of the traditional code optimizations, especially partial redundancy elimination and loop invariant removal. Several prepare-to-branch operations that correspond to branches to the same address can be combined into one operation. Similarly, the prepare-to-branch operation for a branch inside a loop can be moved out of the loop.

A prepare-to-branch operation, whenever possible, should be scheduled so that there is just enough time, between the prepare operation and the corresponding branch, to prefetch instructions from the target address. A similar comment applies to compare operations that compute branch conditions. Note that there are no ordering constraints between the prepare-to-branch operation for a branch and compare operations that compute the corresponding branch condition; they can be scheduled in any order.

Like GPRs or FPRs, branch target registers may need to be saved and restored depending upon the procedure calling convention. As described in Section 10.7, the architecture provides two memory operations, called BSAVE and BRESTORE, for this purpose. Note that, for each BTR, the architecture also defines a pair of control registers that are synonymous with the lower and upper half of the BTR. These control register aliases also can be used to save and restore BTRs.

As mentioned in Section 3.2, it is the compiler's responsibility to ensure that, in an instruction, at most one branch operation takes the branch. As an example, consider the sequence of branches in the left column. In the code, B1, B2, B3 are branch target registers and p1, p2, p3 are predicate register containing the branch conditions.

Control dependent branches	**Transformed code with fully resolved branches**
BRCT(B1, p1);	p4 = ~ p1;
BRCT(B2, p2);	p5 = ~ p1 ∧ ~ p2;
BRCT(B3, p3);	BRCT(B1, p1);
	BRCT(B2, p2) if p4;
	BRCT(B3, p3) if p5;

The second branch is control-dependent upon the first branch; that is, it executes only if the first branch falls through. Similarly, the third branch is control-dependent upon both first and second branches. These operations can be scheduled in an instruction only if branch conditions are mutually exclusive, *i.e.*, any two of them are not true at the same time. There are some special cases, such as branches used to simulate an arithmetic-if in Fortran, that automatically satisfy the requirement. In most cases, however, it is necessary to transform the code to ensure that branch conditions are mutually exclusive. The second column shows one way to do so. Consider the second branch in the transformed code. The guarding predicate p4 ensures that the second branch executes only if the first branch didn't take. In other words, the second branch performs a branch only if its own branch condition is true and the first branch didn't take. We call such branches *fully resolved branches*. Note that compare-to-predicate operations with OR and AND reduction capabilities (see Section 9.3) provide an efficient means to compute predicates p4 and p5.

In the above example, the effect of conditional execution can also be obtained by modifying the branch conditions. Consider, for example, the second branch in the transformed code. To get the same effect, we can modify its branch condition to $p2 \wedge p4$. This is a general transformation that can be used to fold the guarding predicate for a branch into the branch condition. However, this transformation applies only to branch operations that have explicit branch conditions such as BRCT. It doesn't apply to operations like BRL, which don't have explicit branch conditions but do have a predicate to guard their execution.

The effect of multiple branch operations in an instruction can also be obtained by using multiple PBR operations that target the same branch target register. However, this technique may not

provide any benefits over the use of multiple operations. The compiler must ensure that the address visible at the branch point corresponds to the taken branch. One way to do this is to compute fully resolved branch conditions as described above and use them to guard PBR operations. Another is to order PBR operations in an appropriate way.

Branch operations in the delay slots of a branch execute in a pipelined fashion, which may lead to "visits". Consider a branch operation that branches to address A and a branch operation in its delay slot that branches to address B. Suppose the branch latency is 2, and assume that both branches take. Then, the execution proceeds as follows.

Cycle	Instruction
1	First branch;
2	Second branch;
3	Instruction at address A;
4	Instruction at address B;
5	Instruction at address B + 1;

That is, the code at address A executes for some time. Then, the execution starts at the address B, though there is no explicit branch to the address B in the code starting at A. A compiler is, of course, free to take advantage of this fact, but it usually results in convoluted code. It may be better to simply avoid such cases by ensuring that branch conditions for the two branches (one in the delay slot of another) are mutually exclusive.

The effective use of multiple branches in an instruction and branches in the delay slots of another branch introduces additional complexities. It is hard to separate code generation from scheduling, since the decision whether to generate fully resolved conditions or not requires scheduling information. Two scheduling passes, one before and one after code generation, may help alleviate the problem.

We conclude this section with some comments about branch operations used in the software pipelining of loops. We expect that, in most code schemas, a BRF operation will be the only branch operation in an instruction. Moreover, there will be no branch operations in the delay slots of a BRF operation. In some cases, it may be necessary to schedule other BRF operations in the delay slots of a BRF operation. An example is the kernel unrolling to fill the delay slots when the II (initiation interval) of the loop is smaller than the latency of a BRF operation. A simple way to handle such cases is to use Nexti (*i.e.*, BRF.F.F.F) in the delay slots. Similar comments apply to the use of BRW operations. Note that, unlike BRF operations, BRW operations do have an explicit branch condition, which can be transformed in the way described earlier.

The use of these operations is not limited to code schemas that use predicated execution to control the execution of pipeline stages. The programmability of source and destination predicates allows them to be used with or without predicated execution. Table 22 lists the typical settings of sources and destinations for various schemas.

Similarly, these operations can also be used to experiment with code schema that don't make use of rotating registers. Note, however, that these operation do modify RRB (rotating register base). Thus, it may not be possible to use rotating registers in a machine as static registers. Experiments that simply want to use the rotating registers as static registers can still be performed by creating a pseudo-machine that has the appropriate number of static registers and no rotating registers.

Table 22: Typical settings of source and destination predicates of BRF and BRW operations for various code schemas

Code schema	Destination predicate of BRF or BRW	First source of BRW
With predicated execution and rotating registers	PR[0] (i.e., 0^{th} rotating predicate register relative to RRB)	PR[0]
With predicated execution but no rotating registers	A static predicate register	Same as the destination of the BRW operation in the previous iteration
No predicated execution and with or without rotating registers	Bit bucket, *e.g.*, static PR0	Static PR1, which is permanently 1 when used as an input

12 Concluding remarks

We have described an instruction set architecture which has been designed to support research in instruction level parallelism. The architecture is intended to support research experiments in EPIC and superscalar styles of parallelism by providing a baseline for comparison among independent experiments in instruction level parallelism. The architecture is parametric in the degree of parallelism to promote a better understanding of the incremental utility of additional parallelism. A variety of more specialized features are incorporated into the architecture which may provide substantial benefit when they are used in conjunction with advanced compiler technology and a suitable implementation of the architecture. The measurement of the utility of instruction level parallelism and the benefits of these more specialized features is a primary goal of our research. We will continue to explore the merits of compiler techniques and advanced architectural features within this framework.

We hope that this architecture supports the needs of a broader research community which may take active interest in these issues. We invite other researchers who have similar interests to adopt this architecture in order to provide more commonality in comparing results and in order to alleviate the replicated work required if we all use an independent approach. It is inevitable that this architecture will evolve. Its evolution will be engineered to support the investigation of important research questions in instruction level parallelism as they are identified by our collaborators and ourselves. We will ensure that the evolution takes place in a controlled manner and that it is coordinated with our collaborators.

Acknowledgments

We gratefully acknowledge a number of contributions to the body of knowledge presented within this report.

First, we wish to acknowledge our colleagues at Hewlett-Packard Laboratories, who participated in the SWS program which led to the development of the PA-WW architecture, the precursor to the Intel IA-64 architecture. They provided a fertile atmosphere for discussion which helped shape the HPL-PD architecture. In this climate, it was inevitable that we incorporated some ideas from our colleagues without providing specific credit.

Second, we gratefully acknowledge the work of those who participated in the design and definition of the closest historical ancestors to HPL-PD. The HPL-PD architecture inherits substantial insight from the Cydrome Cydra 5, the HP PA-RISC, and the Multiflow Trace architectures. We have incorporated these ideas into a cohesive package to define a state-of-the-

art research vehicle. We thank the members of the teams which defined these machines for a body of knowledge which has greatly assisted our work.

Finally, the HPL-PD architecture incorporates ideas from some of the architectures that have been proposed in the literature, and we gratefully acknowledge those who participated in the definition of these architectures.

References

1. V. Kathail, M. Schlansker and B. R. Rau. HPL PlayDoh architecture specification: Version 1.0 HPL-93-80. Hewlett-Packard Laboratories, February 1994.

2. M. Schlansker, B. R. Rau, S. Mahlke, V. Kathail, R. Johnson, S. Anik and S. G. Abraham. Achieving High Levels of Instruction-Level Parallelism with Reduced Hardware Complexity. HPL Technical Report HPL-96-120. Hewlett-Packard Laboratories, February 1997.

3. M. S. Schlansker and B. R. Rau. EPIC: Explicitly Parallel Instruction Computing. Computer 33, 2 (February 2000), 37-45.

4. M. S. Schlansker and B. R. Rau. EPIC: An Architecture for Instruction-Level Parallel Processors. HPL Technical Report HPL-1999-111. Hewlett-Packard Laboratories, February 2000.

5. IA-64 Application Developer's Architecture Guide. (Intel Corporation, 1999).

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

7. 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 (1988), 967-979.

8. PA-RISC 1.1 Architecture and Instruction Set Reference Manual. (Hewlett-Packard Company, 1992).

9. H. Schorr. Design principles for a high-performance system. Proc. Symposium on Computers and Automata (New York, New York, April 1971), 165-192.

10. H. C. Young and J. R. Goodman. A simulation study of architectural data queues and prepare-to-branch instruction. Proc. IEEE International Conference on Computer Design: VLSI in Computers ICCD '84 (Port Chester, NY, 1984), 544-549.

11. K. Ebcioglu. Some design ideas for sequential natured software, in Parallel Processing (Proceedings of IFIP WG 10.3 Working Conference on Parallel Processing), M. Cosnard (Editor). (North Holland, 1988), 3-21.

12. K. Ebcioglu and R. Groves. Some global compiler optimization and architectural features for improving performance of superscalars RC16145. IBM T.J. Watson Research Center, Yorktown Heights, NY, 1990.

13. W. Y. Chen. Data Preload for Superscalar and VLIW Processors. Ph.D. Thesis. University of Illinois, Urbana, IL, 1993.

14. S. A. Mahlke, W. Y. Chen, R. A. Bringman, R. E. Hank, W. W. Hwu, B. R. Rau and M. S. Schlansker. Sentinel scheduling: A model for compiler-controlled speculative execution. ACM Transactions on Computer Systems 11, 4 (1993), 376-408.

15. G. M. Silberman and K. Ebcioglu. An architectural framework for supporting heterogenous instruction-set architectures. IEEE Computer 26, 6 (1993), 39-56.

16. B. R. Rau, V. Kathail and S. Aditya. Machine-description driven compilers for EPIC and VLIW processors. Design Automation for Embedded System 4 (1999), 71-118.

17. J. C. Gyllenhaal, W.-m. W. Hwu and B. R. Rau. HMDES Version 2.0 Specification. Technical Report IMPACT-96-3. University of Illinois at Urbana-Champaign, 1996.

18. S. Aditya, V. Kathail and B. R. Rau. Elcor's machine description system: Version 3.0 HPL-98-128. Hewlett-Packard Laboratories, October 1998.

19. S. Aditya, B. R. Rau and R. C. Johnson. Automatic Design of VLIW and EPIC Instruction Formats. HPL Technical Report HPL-1999-94. Hewlett-Packard Laboratories, March 2000.

20. S. Aditya, S. A. Mahlke and B. R. Rau. Retargetable assembly and code minimization techniques for custom EPIC / VLIW instruction formats. ACM Transactions on Design Automation of Electronic Systems (to appear 2000).

21. B. R. Rau. Dynamically scheduled VLIW processors. Proc. 26th Annual International Symposium on Microarchitecture (Austin, Texas, December 1993), 80-92.

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

23. S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank and R. A. Bringman. Effective compiler support for predicated execution using the hyperblock. Proc. The 25th Annual International Symposium on Microarchitecture (Portland, OR, 1992), 45-54.

24. J. C. H. Park and M. S. Schlansker. On predicated execution. Technical Report HPL-91-58. Hewlett Packard Laboratories, May 1991.

25. S. G. Abraham, R. A. Sugumar, D. Windheiser, B. R. Rau and R. Gupta. Predictability of load/store instruction latencies. Proc. 26th Annual International Symposium on Microarchitecture (1993).

26. B. R. Rau, M. S. Schlansker and P. P. Tirumalai. Code generation schemas for modulo scheduled DO-loops and WHILE-loops HPL-92-47. Hewlett-Packard Laboratories, April 1992.

27. B. R. Rau, M. S. Schlansker and P. P. Tirumalai. Code generation schemas for modulo scheduled loops. Proc. 25th Annual International Symposium on Microarchitecture (Portland, Oregon, December 1992), 158-169.

28. P. Tirumalai, M. Lee and M. S. Schlansker. Parallelization of loops with exits on pipelined architectures. Proc. Supercomputing '90 (November 1990), 200-212.

29. B. R. Rau. Data flow and dependence analysis for instruction level parallelism, in Fourth International Workshop on Languages and Compilers for Parallel Computing, U. Banerjee, D. Gelernter, A. Nicolau and D. Padua (Editor). (Springer-Verlag, 1992), 236-250.