Automatic Architecture Synthesis and Compiler Retargeting for VLIW and EPIC Processors

Shail Aditya, B. Ramakrishna Rau
HP Laboratories Palo Alto
HPL-1999-93
January, 2000

E-mail:{aditya,rau}@hpl.hp.com

architecture synthesis, micro-architecture synthesis, VLIW processors, EPIC processors, automatic processor design, abstract architecture specification, datapath design, resource allocation, mdes extraction, compiler retargeting, controlpath design, instruction pipeline design, RTL generation

This paper describes a mechanism for automatic design and synthesis of very long instruction word (VLIW), and its generalization, explicitly parallel instruction computing (EPIC) processor architectures starting from an abstract specification of their desired functionality. The process of architecture design makes concrete decisions regarding the number and types of functional units, number of read/write ports on register files, the datapath interconnect, the instruction format, its decoding hardware, and the instruction unit datapath. The processor design is then automatically synthesized into a detailed RTL-level structural model in VHDL along with an estimate of its area. The system also generates the corresponding detailed machine description and instruction format description that can be used to retarget a compiler and an assembler respectively. This process is part of an overall design system, called Program-In-Chip-Out (PICO), which has the ability to perform automatic exploration of the architectural design space while customizing the architecture to a given application and making intelligent, quantitative, cost-performance tradeoffs.

An overview of the system described in this document was presented at the *International Symposium for System Synthesis, 1999*, San Jose, November. 1999

1 Introduction

VLIW (Very Long Instruction Word) processors have begun to establish themselves as the processor of choice in high performance embedded computer systems, especially in situations where it is important to have a compiler that can generate efficient code from a high level language. The cost of the embedded VLIW processor, which is one of the primary concerns in embedded systems, can be greatly reduced if the processor is customized to the anticipated workload by making a number of application-specific design choices. However, the non-recurring expense (NRE) of designing a custom VLIW processor, along with a compiler for it, can be prohibitive. The key enabling technology, therefore, for realizing the full potential of application-specific VLIW architectures lies in architectural design automation.

Although there has been a fair amount of work done on providing the capability to automatically design the architecture of a sequential, application-specific instruction-set processor (ASIP) – primarily a matter of designing the opcode repertoire – there has been relatively little work in the area of architecture synthesis of VLIW processors or, for that matter, processors of any kind that provide significant levels of instruction-level parallelism (ILP). The work which has been done tends to focus largely upon the synthesis of a VLIW processor's datapath [5, 7, 11]. The automatic design of a non-trivial instruction format, and the synthesis of the corresponding instruction fetch and decode micro-architecture have not been addressed for VLIW processors. And yet, it is these issues that consume the major portion of a human designer's efforts during the architecture and micro-architecture phases of a VLIW design project.

One reason for this might be the lack of retargetable VLIW compiler technology. A retargetable compiler is an essential component of the design space exploration process, enabling the evaluation of each candidate design. Another reason is that designing a VLIW processor, even manually, is a complex task. The designer of an application-specific VLIW processor must make a large number of architectural and micro-architectural decisions in the context of the given application, including the following:

1. select the minimal opcode repertoire for the specified workload,

2. choose to include or exclude architectural features such as predication, control or data speculation, rotating registers, etc. which, consequently, impact the design of the instruction format and the micro-architecture of the processor,

3. decide upon the least expensive mix of functional units that can provide the desired level of instruction-level parallelism,

4. decide upon the number of register files, how many registers each should contain, and what data types each should accommodate and, hence, the width of the registers in each register file,

5. minimize the number of read and write ports for each register file by maximizing the extent to which the functional units share the ports, while ensuring that this does not compromise the desired level of ILP,

6. design an instruction format that supports the requisite level of ILP and which reduces the wasting of code space due to the presence of no-ops in the VLIW instruction without excessively complicating the instruction pipeline, and

7. design the instruction pipeline and the instruction decoders, as well as the instruction prefetch unit which keeps the instruction pipeline full.

The design space is enormous and the cost of exploring various points in detail is so great that most architects approach the problem with the mind-set of only making incremental modifications to an initial design. The initial design is generally based on the architect's experience and intuition as to what constitutes a good design. The modifications, although often based on measurements and simulation, are limited to relatively small perturbations of the initial design. Such an approach has the serious disadvantage of being susceptible to getting stuck with a locally optimal design while ignoring superior designs that may be radically different, but which would require an impractically large amount of manual exploration and evaluation.

1.1 Our approach

In this paper, we present a fully automated system for designing the architecture and micro-architecture of VLIW processors and their generalization, EPIC (Explicitly Parallel Instruction Computing) processors[1]. We refer to this process as *architecture synthesis* to distinguish it from behavioral or logic synthesis which are at a lower level. In addition to the well understood features of the VLIW style of architecture, the space of processors that we are interested in exploring is characterized by the HPL-PD architecture family [13] which includes features such as predication, control and data speculation, rotating registers, and explicit source and destination specifiers for load and store operations at various levels of the memory hierarchy. Processors with these features have the ability to exploit high degrees of compiler-specified ILP both in numerically-intensive applications as well as in applications that are intensive in branches and pointer-based memory references.

The architecture synthesis system that we describe in this paper is part of PICO (Program-In-Chip-Out), a broader system synthesis and design exploration tool which performs hardware-software co-synthesis. In addition to the custom VLIW processor, PICO may design one or more non-programmable, systolic-array co-processors (ASICs) and a two-level cache hierarchy to support these processors. It partitions the given application between hardware (the systolic arrays) and software, compiles the software to the custom VLIW, and synthesizes the interface between the processors. We refer to PICO's VLIW design capability as PICO-VLIW which is the subject of this paper.

In PICO-VLIW, we decompose the process of automatically designing an application-specific VLIW processor into three closely inter-related sub-systems as shown in Figure 1. The first sub-system is our design space explorer, the **Spacewalker**, whose responsibility is to search for the Pareto-optimal architectures, *i.e.*, those architectures whose implementations are either cheaper or faster (or both) than any other architecture. In order to do this efficiently, the Spacewalker uses sophisticated search strategies and heuristics that are, however, beyond the scope of this paper.

[1]For the sake of brevity, we use the term VLIW to include EPIC as well in the rest of this paper.

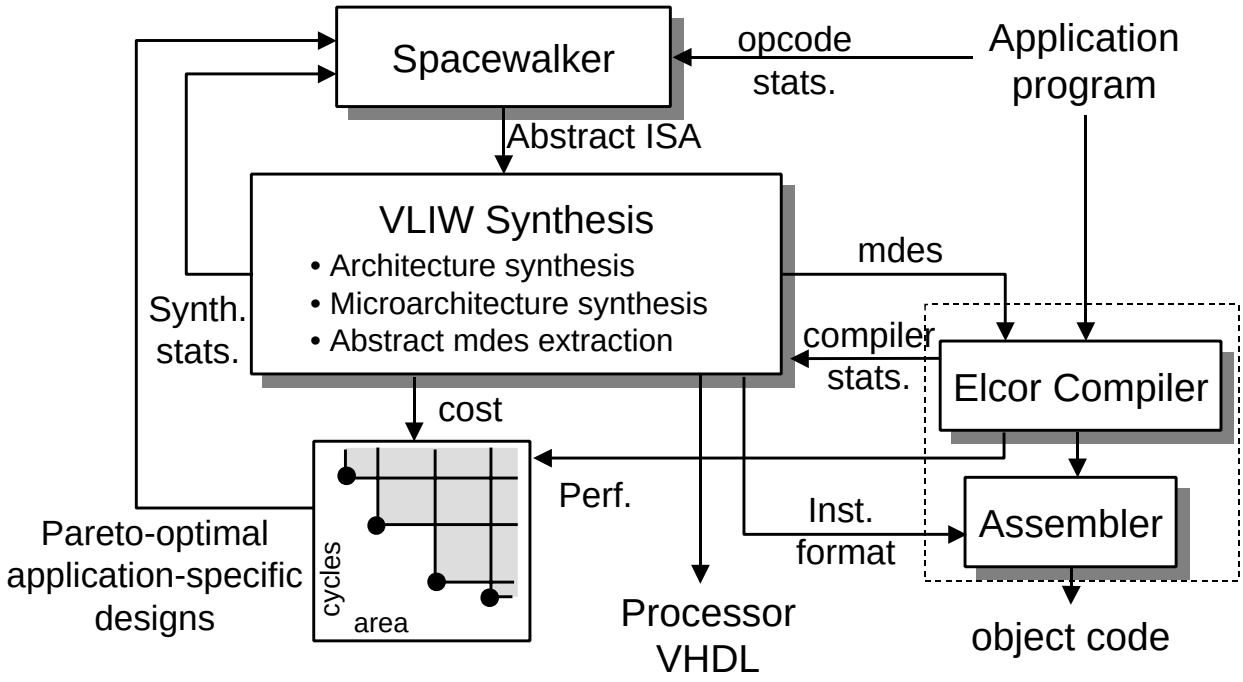

Figure 1: The PICO-VLIW design system.

The second sub-system is the **VLIW architecture synthesis** sub-system whose responsibility is to take the abstract architecture specification generated by the Spacewalker and to create the best possible concrete architecture and micro-architecture, as well as a machine-description database used to retarget the compiler. The system outputs a RTL-level, structural VHDL description of the processor and estimates the chip area consumed by it.

The third sub-system consists of **Elcor**, our retargetable compiler for VLIW processors, and a retargetable assembler. Both are automatically retargeted by supplying the machine-description database. Elcor's responsibility is to generate the best possible code for the application on the processor designed by the VLIW architecture synthesis sub-system, and to evaluate its performance by counting the number of cycles taken to execute the program. The area and execution time estimates are then used by the Spacewalker to guide the next step of its search.

The PICO-VLIW system explores a number of design variables within this framework including the number and kind of functional units in the processor, the number and size of register files,

the number of read/write ports on each file, various interconnect topologies between functional units and register files, cache and memory hierarchy, high performance architectural mechanisms such as predication and speculation, the number and structure of the various instruction templates, and the corresponding instruction fetch and decode hardware. The system makes intelligent cost and performance tradeoffs involving the above variables, some internally within a single design process, and some externally by walking the architectural design space defined by these variables. Additional feedback statistics related to the design, such as register port usage, instruction template usage, and functional unit utilization, are also generated that can be used to make automatic adjustments and improvements in the high-level specification.

The VLIW synthesis module and Elcor are both designed with an understanding that they will be invoked hundreds, possibly thousands, of times by the Spacewalker in the course of finding a Pareto-optimal set of designs. Consequently, quick, heuristic algorithms are used in both modules as a rule. Optimal, expensive algorithms could be employed once the Pareto-optimal abstract architectures have been selected, in order to obtain higher levels of quality for the hardware and the code that is actually to be used.

Such a design system is not only useful as a push button tool for fully automated architecture synthesis and exploration but also as a manual design assistant, whereby some aspects of the design are done manually and the rest are filled in by the system, or existing designs are customized to an application or an application domain and their cost and performance is evaluated automatically. In the latter case, a high-level specification may be automatically extracted from the existing architecture.

The major contribution of the work reported here is in establishing a framework which formalizes and makes algorithmic what has thus far been an ad hoc, manual process. We do not believe that the specific heuristics, currently in place, are necessarily the best possible; the development of robust, near-optimal heuristics is a topic of further research.

1.2 Focus of this paper

In this paper, we focus our attention on the VLIW architecture synthesis sub-system of PICO-VLIW. Starting from an abstract specification of an architectural design point, the system automatically generates the following:

1. the instruction set architecture (ISA) for the processor, including the choice of opcodes and a multi-template instruction format for the machine, both possibly customized to a given application or workload,

2. the detailed micro-architecture of the machine, *i.e.*, its datapath and the instruction unit, which can be output in the form of structural VHDL, along with a specification of its control tables for subsequent synthesis into combinational logic,

3. an abstract (non-structural) machine description, *mdes* for short, that is used by our mdes-driven, retargetable compiler and assembler to generate code for the newly designed VLIW processor, and by our mdes-driven simulator to simulate it,

4. detailed synthesis feedback for the Spacewalker describing hardware resource costs and utilization, and

5. an architecture manual for the VLIW processor documenting the above information along with a cost report.

The remaining sections describe the VLIW synthesis process in more detail. Section 2 describes the overall design flow. In Section 3, we describe the abstract architecture specification that is used by the Spacewalker (or a manual designer) to specify each architecture design point. Section 4 describes the micro-architecture (datapath) design process and the related algorithms. The design of the concrete instruction-set architecture (instruction format) is the topic of a separate technical report [1]. Section 5 described the process of automatically extracting a machine description (mdes) from the datapath which is used to retarget the Elcor compiler to the architecture being synthesized. In Section 6, we complete the processor micro-architecture by designing the instruction

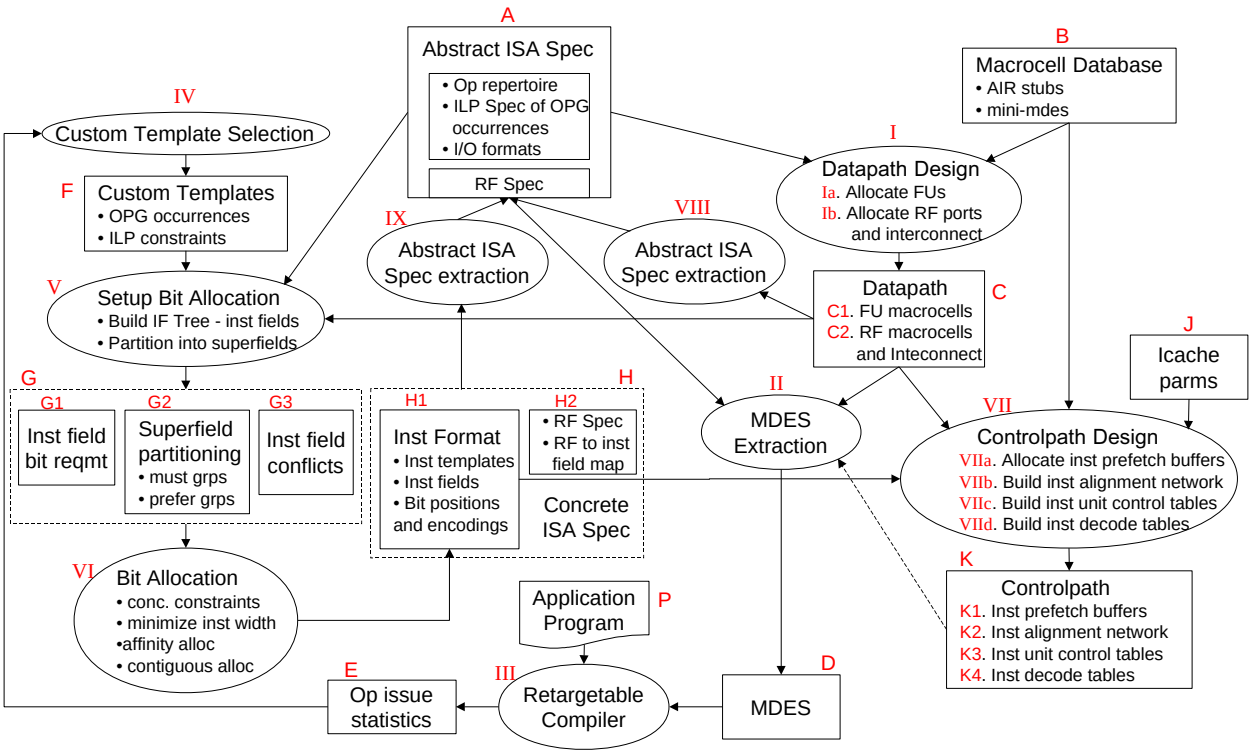

Figure 2: VLIW synthesis design flow in PICO-VLIW.

fetch pipeline and the fetch and decode control logic. Section 7 describes the various output from synthesis process including the processor VHDL and the structural feedback to the Spacewalker. Finally, Section 8 discusses related work and Section 9 concludes the paper.

2 VLIW Processor Synthesis Design Flow

The overall design flow for the VLIW/EPIC processor synthesis is shown in Figure 2 and is outlined below. The boxes in the figure represent internal and external data-structures, while the ovals represent methods and design processes that generate and use those data-structures.

In manual VLIW design as well as related work on VLIW synthesis [7, 10], the starting point is the concrete ISA which consists of a specification of the register file structure and an instruction format. We take a different approach, since we view the concrete ISA as an overly-constrained

input specification. Instead, we start with an abstract architecture specification (A), which specifies the desired levels of concurrency and the opportunities for resource sharing, but which leaves the detailed decisions as to how best to share register ports and instruction bits to the datapath and the concrete ISA design steps, respectively. This allows PICO-VLIW to go about the design in an unconventional order: first, to design a datapath that is consistent with the requirements of the abstract architecture specification (Step I) and to extract an abstract machine description from it (Step II); next, to design a concrete ISA in the light of the control ports of the datapath (Steps IV, V, VI), and to then design the controlpath (Step VII), *i.e.*, the instruction prefetch, alignment and decode hardware. By designing the concrete ISA after the datapath, we are able to achieve better trade-offs between code size and the complexity of the controlpath. We briefly describe the details of each of these steps below.

Abstract ISA specification (A). The input to the design process is an abstract architecture specification (*archspec* for short) of the machine to be designed. This specification consists of the following components that are discussed in more detail in Section 3.

- A list of operations that can be executed on the machine.
- ILP constraints among operation specifying which operations can execute in parallel and which ones are mutually exclusive.
- A specification of the input/output operand locations for each operation.
- A list of all the register files in the machine and their storage properties.

Datapath design (I). The datapath design process uses the archspec (A) to generate the datapath (C), while drawing its components from a database of macrocells (B). The database contains macrocell descriptions in our architecture intermediate representation (AIR) that summarize various hardware and programming properties of the cells. These AIR stubs point to actual HDL descriptions of these macrocells.

The datapath design process consists of the following steps that are discussed in more detail in Section 4.

Functional unit allocation – A set of functional unit macrocells are instantiated from the database that can together implement the all operations specified in the archspec subject to the specified ILP constraints.

Register file port allocation – For each of the register files specified in the archspec, we first determine the number of input and output ports needed by the various functional units based on their input and output operand connectivity and the ILP constraints.

Register file and interconnect generation – Using the results of the port allocation, the register files specified in the archspec are instantiated from the macrocell database. Then, the files are connected to the various functional units via buses, multiplexors and tristate elements according to their input and output requirements.

Mdes extraction (II). Given the datapath (C), this step extracts an abstract machine description (D) which can be used to retarget a compiler (III) to the machine being designed. The machine description hides unnecessary details of the datapath and presents a programming model of the target machine to the compiler. This extraction process is discussed in Section 5.

Instruction format design (IV, V, VI). The archspec is also used to design an instruction format (H1) to control the various hardware control ports in the datapath (C). This process is described in detail in the companion technical report [1]. We outline the major steps below.

Custom template design (IV) – The ILP constraints within the archspec give rise to a set of instruction templates consisting of operations that may be issued concurrently. Another source of such constraints (and the corresponding instruction templates) is the operation scheduling statistics (E) obtained by compiling a given application program (P) to the target machine. Frequently occurring combinations of co-scheduled operations are used to generate cus-

tomized instruction templates (F) that are smaller in size thereby reducing the overall size of the program.

Bit allocation problem setup (V) – The instruction format design process uses an auxiliary data-structure called the IF-tree which is built using the instruction templates and the operation format information present in the archspec. The leaves of the tree are the instruction fields that need to be assigned bit positions within an instruction. The IF-tree data-structure helps to identify the following information that is subsequently used for instruction bit allocation:

- a list of all the instruction fields in various instruction templates and their bit-width requirement (G1),
- a partitioning of the instruction fields into sets (super-fields) that control the same datapath control port so that they may be assigned the same bit positions in the instruction template (G2), and,
- a conflict graph of the instruction fields that are used concurrently (G3).

Bit allocation (VI) – The various instruction fields within each template are assigned bit positions subject to their bit-requirement and concurrency conflicts using a graph coloring approach. Heuristics are used to reduce the overall template width and the decode complexity by packing the instruction fields to the left (leftmost allocation), assigning contiguous bit positions to multi-bit fields (contiguous allocation), and aligning instruction fields within the same super-field to the same bit position (affinity allocation).

Controlpath design (VII). Once the instruction format (H1) and the datapath (C) have been designed, we can proceed to complete the design of the processor by generating the controlpath (K). This consists of the following components that are discussed in more detail in Section 6.

Instruction prefetch buffer – An instruction packet of a certain size is fetched from the instruction cache and brought into a FIFO queue. The number of buffers in this queue and its width depend on the instruction cache parameters (J), the cache access time and its packet width.

Instruction alignment network – Since, the instructions may be of variable length, they may not be properly aligned as they are fetched from the memory into the instruction cache and from the cache into the prefetch buffer. An instruction alignment network aligns the left boundary of the instruction to the first bit position of the instruction register at each cycle.

Instruction unit control tables – The alignment network, the prefetch buffer, and the instruction fetch from the cache is controlled by logic whose specification as a control table is generated automatically. This logic is responsible for the following tasks at each cycle:

- keeping track of the width of the instruction and the unused bits in the instruction register and at the head of the prefetch buffer,
- issuing instruction cache fetches, prefetch buffer fills and instruction register fills at the appropriate times, and
- generating the appropriate shift signal for the alignment network to align the next instruction into the instruction register.

Instruction decode tables – At each cycle, the left aligned instruction in the instruction register is decoded to yield the appropriate control signals for the various datapath control ports. A control table specification for this decode logic is also generated automatically.

Instead of a general-purpose instruction unit that fetches and decodes instructions based on compact instruction templates as described above, it is also possible to control the datapath with a a finite-state machine (FSM) based or a ROM-based controller that is specialized to a given scheduled program. The FSM specification (or the ROM bits) can be automatically generated from the scheduled operations. This strategy may prove useful when the programmability of a general-purpose instruction unit is not needed (e.g. in an ASIC design), and the cost of the instruction unit outweighs the code-size reduction achieved by encoding the specification of control signals needed at each cycle into program instructions.

VHDL and synthesis report generation. The final step of the design process (not shown in Figure 2) is to assemble the various AIR components and their interconnect wires and produce

a structural description of the hardware at the RTL-level in a standard hardware description language such as VHDL. This description can be linked with the respective HDL component libraries pointed to by the macrocell database and processed further for hardware synthesis and simulation.

The PICO system also produces a synthesis report consisting of a breakdown of the area estimate for the processor by components and feedback information for the architectural Spacewalker with regard to the utilization of various hardware resources and instruction format bits by the various components of the input archspec. In addition, the system produces a reference manual for the concrete instruction set architecture (H) of the target machine complete with its operation repertoire, instruction formats, and register file specifications.

3 Input architecture specification

Architecting a VLIW processor is considerably more complex than a sequential one. In addition to picking an operation repertoire, one must specify the extent and nature of the processor's ILP concurrency. A VLIW processor, when designed by a competent architect, exhibits certain features as listed below which we wish PICO-VLIW to emulate.

- Functional units are heterogeneous; although one might include the ability to issue two adds every cycle, which requires two integer units, only one unit may be capable of shifting and the other unit able to do multiplication.

- Register file ports are shared; a multiply-add operation, which requires three register read ports, may be accommodated by ”borrowing” one of the ports of another functional unit which cannot, now, be used in parallel with the multiply-accumulate.

- Likewise, instruction bits are shared; a load or store operation, which requires a long displacement field, might use the instruction bits that would otherwise have been used to specify an operation on some functional unit.

In order for PICO-VLIW to yield competently designed processors, we need the Spacewalker to be able to specify such architectures to the VLIW synthesis sub-system.

Our choice of the interface between the Spacewalker and the VLIW synthesis sub-system (refer Figure 1) involves a delicate balance between giving the Spacewalker adequate control over the architecture, without bogging it down by requiring it to specify a detailed instruction format. Our compromise is that the Spacewalker specifies the operation repertoire, the requisite level of ILP concurrency, and the opportunities for sharing register ports and instruction bits. Thereafter, it relies upon the concrete ISA design module, the datapath design module and the controlpath design module to avail of these opportunities while supplying the requisite level of concurrency.

The Spacewalker tasks the VLIW synthesis sub-system via the *Abstract ISA Specification* (*arch-spec* for short). In this specification, the operation repertoire of the target processor is specified in an abstract manner together with constraints upon its concurrency. These concurrency constraints can then be exploited by the VLIW synthesis sub-system to yield less expensive architectures, with heterogeneous functional units and resource sharing, at the requisite level of concurrency. We discuss the various components of the archspec below.

3.1 Operation Groups and Exclusions

At an abstract level, an architecture specification need only specify the functionality of the hardware implementation in terms of its opcode repertoire and the desired performance level. Then the exact structure of the implemented machine in terms of its datapath and control structure may be synthesized from this specification. In PICO, we specify an architecture by simply enumerating the set of opcodes that it implements and the level of parallelism that exists amongst them.

For convenience, the various instances of HPL-PD opcodes for a given machine are grouped into *Operation Groups* (*opgroups* for short) each of which is a set of opcode instances that are similar in nature in terms of their latency and connectivity to physical register files and are expected to be mutually exclusive with respect to operation issue, e.g. add and subtract operations on the same ALU. By definition, all opcode instances within an operation group are mutually exclusive while,

by default,those across operation groups are allowed to execute in parallel. The parallelism of the machine may be further constrained by placing two or more operation groups into *Exclusion Groups* which makes all their opcode instances mutually exclusive and allows them to share resources, e.g. a multiply operation that executes on a separate multiply unit but shares the result bus with the add operation executing on an ALU.

As an example, a simple 2-issue machine is specified below[2]. The specification language we use is the database language HMDES Version 2 [9] that organizes the information into a set of interrelated tables called *sections* containing rows of records called *entries* each of which contain zero or more columns of property values called *fields*.

```
SECTION Operation_Group {
  OG_alu_0(ops(ADD SUB) format(OF_intarith2l));
  OG_alu_1(ops(ADD SUB) format(OF_intarith2l OF_intarith2r));
  OG_move_0(ops(MOVE) format(OF_intarith1));
  OG_move_1(ops(MOVE) format(OF_intarith1));
  OG_mult_0(ops(MPY) format(OF_intarith2l));
  OG_cmp_0(ops(CMP) format(OF_intarith2lp));
  OG_shift_0(ops(SHL SHR) format(OF_intarith2l OF_intarith2r));
}
SECTION Exclusion_Group {
  EG_0(opgroups(OG_alu_0 OG_move_0 OG_mult_0 OG_cmp_0));
  EG_1(opgroups(OG_alu_1 OG_move_1 OG_shift_0));
  EG_2(opgroups(OG_mult_0 OG_shift_0));
}
```

This example specifies two alu operation groups, two move operation groups, and one each of multiply, compare and shift groups. These operation groups are classified into several independent

[2]A complete specification for a 2-integer, 1-float, 1-memory, 1-branch unit VLIW processor customized to the "jpeg" application is shown in Appendix A.

exclusion groups denoting sharing relationships among the operation groups. Each operation group also specifies one or more operation formats (defined shortly) shared by all the opcodes within the group. Additional operation properties such as latency and resource usage may also be specified with the operation group but are not shown here since they are not relevant to this discussion.

3.2 Register Files and Operation Formats

The archspec specifies additional information to describe the physical register files of the machine and the desired connectivity of the operations to those files. A *Register File* entry defines a physical register file of the machine and identifies its width in bits, the registers it contains, and a virtual file specifier that specifies the types of data it can hold [13]. It may also specify additional properties such as whether or not the file supports speculative execution, whether or not the file supports rotating registers, and if so, how many rotating registers it contains. The immediate literal field within the instruction format of an operation is also considered to be a (pseudo) register file consisting of a number of "literal registers" that have a fixed value.

The *Operation Format* entries each specify the set of choices for source/sink locations for the operations in an operation group. Each operation format consists of a list of *Field Types* that determine the set of physical register file choices for a particular operand. For predicated operations, a separate predicate input field type is also specified.

```
SECTION Register_File {
  gpr(width(32) regs(r0 r1 ... r31) virtual(I));
  pr(width(1) regs(p0 p1 ... p15) virtual(P));
  s(width(16) intrange(-32768 32767) virtual(L));
}
SECTION Field_Type {
  FT_I(regfile(gpr));
  FT_P(regfile(pr));
  FT_L(regfile(s));
```

```
  FT_IL(compatible_with(FT_I FT_L));
}
SECTION Operation_Format {
  OF_intarith1(pred(FT_P) src(FT_IL) dest(FT_I));
  OF_intarith2l(pred(FT_P) src(FT_IL FT_I) dest(FT_I));
  OF_intarith2r(pred(FT_P) src(FT_I FT_IL) dest(FT_I));
  OF_intarith2lp(pred(FT_P) src(FT_IL FT_I) dest(FT_P));
}
```

The example shows that the above machine has a 32-bit general purpose register file "gpr", a 1-bit predicate register file "pr" and a 16-bit literal (pseudo) register file "s". Each register file can be used alone or in conjunction with other files in a field type specification as a source or sink of an operand. Field types for the predicate, source and destination operands are combined to form the valid operation formats for each operation group. For example, the 2-input alu operation group "OG_alu_0" has an operation format "OF_intarith2l" which specifies that its predicate comes from the register file pr, its left input could be a literal or come from the register file gpr, and its right input and output come from and go to the register file gpr, respectively. For notational convenience, we may write this operation format as a string such as "pr ? gpr s, gpr : gpr", where the colon separates the inputs from the outputs and the comma separates the various field types [13].

4 Datapath Design

A VLIW/EPIC datapath consists of multiple functional units and register files connected together via a bus interconnect. The register files are specified in the archspec directly while the design of the functional units and their connectivity to the register files is the topic of discussion in this section.

Until now, we have used the term "functional unit" informally. Now, we define it more clearly. A *functional unit* is an abstraction of a group of hardware resources that behaves as a unit for

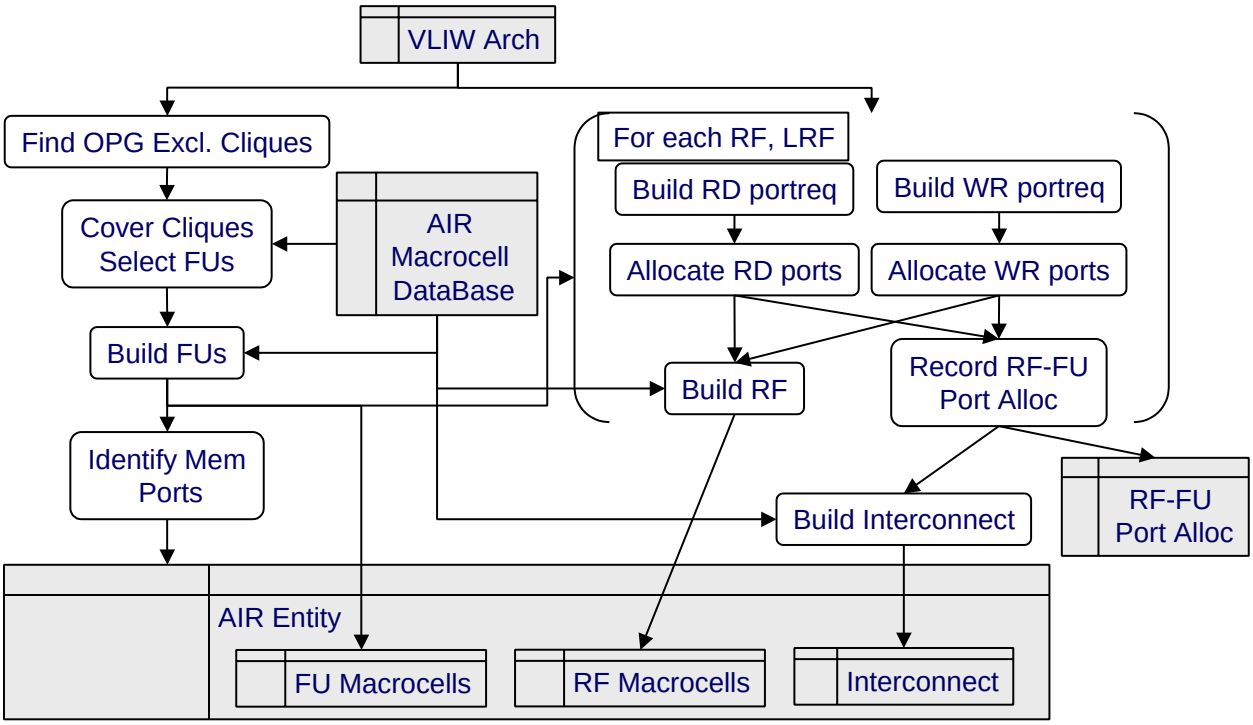

Figure 3: The datapath design flow.

the purpose of executing a sequential stream of operations issued to that unit. For our purpose, it consists of one or more hardware macrocells that are capable of executing the sequential stream of operations together with the register file read/write ports needed to source/sink operands to/from the macrocells. In this section, we describe how the various functional units of the machine can be synthesized automatically starting from the archspec.

The overall design flow process is shown in Figure 3. We need to make the following major design decisions:

1. *Selecting macrocells* – The archspec identifies operations that the machine should be capable of executing and the amount of instruction-level parallelism desired. This information is used to select a set of macrocells that cover the desired functionality at the minimum cost. Then, each of the operation groups specified in the archspec is mapped to a specific macrocell.

2. *Determining number of read/write ports of register files* – In a VLIW/EPIC architecture,

many functional units may connect to the same register file requiring multiple read/write ports to that file. Since highly ported files are very expensive, it is very important to minimize the number of ports on each register file. The mutual exclusion among the operation groups mapped to the various macrocells guides the sharing of register ports.

Subsequently, the selected macrocells and the register files are instantiated into our architecture intermediate representation (AIR) and are interconnected via buses, multiplexors, and tristate buffers.

4.1 Macrocell Selection

The various macrocells are drawn from a macrocell database. Each macrocell keeps track of the operations its can execute as well as a pointer to its actual hardware description that implements it. A standard component library such as Synopsys Designware may be used for implementing the macrocells directly or these implementations may be custom built, for example, to provide additional VLIW/EPIC functionality such as predication and/or speculation support.

The first step in selecting the appropriate macrocells from the macrocell database is to determine which operation groups in the archspec are mutually exclusive so that they may be mapped to the same macrocell. Furthermore, it is desirable to make such mutually exclusive sets of operation groups as large as possible so that a minimum number of independent macrocells (and other hardware resources) may be used. This can be achieved by computing all *cliques*[3] within a graph formed by the operation groups and their mutual exclusion relationships.

4.1.1 Finding exclusion cliques

The pseudo-code for the algorithm to find maximal cliques appears in Figure 4. The algorithm recursively finds all cliques of the graph starting from an initially empty current clique by adding one node at a time to it. The nodes are drawn from a pool of candidate nodes which initially

[3] A *clique* of nodes within a graph is a subgraph in which every node is a neighbor of every other node and no other node from the graph may be added without violating this property.

```
procedure FindCliques(NodeSet currentClique, NodeSet candidateNodes)
1:    // Check if any candidate remains
2:    if (candidateNodes is empty)  then
3:       // Check if the current clique is maximal
4:       if (currentClique is maximal)  then
5:          Record(currentClique) ;
6:       endif
7:    else
8:       NodeSet tryNodes = candidateNodes ;
9:       while (tryNodes is not empty)  do
10:         H1: if ((currentClique ∪ candidateNodes) ⊆ some previous clique) break ;
11:         Node node = Pop(tryNodes) ;
12:         candidateNodes = candidateNodes - {node} ;
13:         if (currentClique ∪ {node} is not complete) continue ;
14:         H2: NodeSet prunedNodes = candidateNodes ∩ Nhbrs(node) ;
15:         FindCliques(currentClique ∪ {node}, prunedNodes) ;
16:         H3: if (candidateNodes ⊆ Nhbrs(node)) break ;
17:         H4: if (this is first iteration) tryNodes = tryNodes - Nhbrs(node) ;
18:      endwhile
19:   endif
```

Figure 4: Pseudo-Code for finding maximal cliques.

contains all nodes of the graph. The terminating condition of the recursion (Line 2) checks to see if the candidate set is empty. If so, the current clique is recorded if it is maximal (Line 4), i.e. there is no other node in the graph that can be added to the current clique while still remaining complete.

If the candidate set is not empty, then we need to grow the current clique. Each incoming candidate node is a potential starting point for growing the current clique (Line 8). This is the place where we are doing an exponential search. Various heuristics are found in the literature [12] to grow the maximal cliques quickly and to avoid examining sub-maximal and previously examined cliques repeatedly. The heuristics used by our algorithm are described below.

The first heuristic we use (H1) is to check whether the current clique and the candidate set is a subset of some previously generated clique. If so, the current procedure call can not produce any new cliques and is pruned. Otherwise, a candidate is selected for growing the current clique. If the selected candidate forms a complete graph with the current clique (Line 13), we add it to the current clique and call the procedure recursively with the remaining candidates. The second

heuristic we use (H2) is to restrict the set of remaining candidates in the recursive call to just the neighbors of the current node since any other node will always fail the completeness test within the recursive call.

After the recursive call returns, we apply two more heuristics that attempt to avoid re-examining the cliques that were just found. If the remaining candidates are all found to be neighbors of the current node (H3), then we can prune the remaining iterations within the current call since a maximal extension of the current clique involving any of those neighbors must include the current node and all such cliques were already considered in the recursive call. On the other hand, if non-neighboring candidates are also present, we drop the neighbors of the current node from being considered as start points for growing the current clique (H4). This is because a maximal extension of the current clique involving one of the neighboring nodes and not involving the current node must involve one of the non-neighboring nodes and therefore can be detected by starting from the non-neighboring nodes directly. This pruning of the trial nodes may be performed only during the first iteration of the while loop, otherwise we may miss the cliques formed among the neighbors being dropped in each iteration.

The above algorithm performs reasonably well in practice for moderate sized graphs (less than 30 sec for 100-150 operation groups on an average workstation). Of course, the time varies greatly with the kind of exclusions specified in the archspec as it governs the number of distinct maximal cliques required to cover all node-to-node exclusion relations.

4.1.2 Finding minimal set-cover of the cliques

Once the exclusion cliques are determined, we draw macrocells from the database such that all operation groups covered by a macrocell fall within a single clique. This guarantees that a macrocell is assigned operation groups that are mutually exclusive. Furthermore, we need to cover as many operation groups with as few and as cheap (in terms of area) macrocells as possible. This is achieved by finding a minimal set-cover of the cliques with the macrocells in the database. This is a well known NP-complete problem [8] and various heuristics have been suggested in the literature

```
procedure McellCover(NodeSetList cliques, McellTable database)
1:    // repeatedly find the best macrocell to cover remaining op groups
2:    while (cliques is not empty) do
3:       Mcell bestCell = nullCell ;
4:       NodeSet bestCover = nullSet ;
5:       for (mcell ∈ database) do
6:          NodeSet mcellCover = nullSet ;
7:          for (clique ∈ cliques) do
8:             NodeSet tryCover = { opg | opg ∈ clique, mcell implements opg } ;
9:             H1: if (|tryCover| > |mcellCover|) then
10:               mcellCover = tryCover ;
11:            endif
12:         endfor
13:         H2: if (|mcellCover| > |bestCover| or
14:         H3: (|mcellCover| == |bestCover| and Cost(mcell) < Cost(bestCell))) then
15:            bestCover = mcellCover ;
16:            bestCell = mcell ;
17:         endif
18:      endfor
19:      Instantiate(bestCell, bestCover) ;
20:      for (clique ∈ cliques) do
21:         clique = clique - bestCover ;
22:      endfor
23:   endwhile
```

Figure 5: Pseudo-Code for finding minimal set-cover of the cliques.

to solve this [12]. We describe our heuristics below.

The pseudo-code for the minimal set-cover algorithm appears in Figure 5. The outer loop at Line 2 repeatedly reduces the given list of cliques by covering a subset of operation groups occurring within some clique by the best macrocell that implements the operations within that subset (Line 8). This best macrocell is instantiated within the datapath (Line 19) and the operation groups in the selected subset are recorded as having been covered by this macrocell by subtracting them from every clique (Line 21).

The heuristics for selecting the best macrocell fall under two categories: those that maximize the size of the set being covered and hence minimize the number of macrocells selected (H1, H2), and those that minimize the cost of the macrocell selected (H3). For the same overall functionality,

minimizing the total number of macrocells reduces the external interconnect cost while picking the least cost macrocells reduces overall chip area. There may be additional considerations such as timing, power, routability, geometry (for hard macros) etc. that are not shown here.

4.2 Register File Port Allocation

For each operation group prescribed in the archspec, its operation format specifies the desired connectivity to the appropriate register files. Once an operation group has been mapped to a macrocell as above, this specification can be translated to a connectivity specification between the macrocells and the various register files by mapping the input/output operands of each operation to the corresponding port of the macrocell using the information described in Section 5.2. The next major design decision is to determine the number of read and write ports required on each register file so that it can be instantiated in hardware and the appropriate interconnect may be setup.

As shown in Figure 3, we formulate the problem of determining the number of read and write ports for each register file as a separate resource allocation problem. In each formulation, macrocell ports are thought of as the resource requesters each requesting one register file port, which is the resource. The resource allocation problem is to assign resources to requesters using a minimum number of resources, while guaranteeing that concurrent requesters are assigned different resources. It is also possible to generalize this situation by allowing the same macrocell port to access different register file ports for different operation groups mapped to it, or for that matter, the same operation group under different instruction templates. Currently, we do not allow this.

The resource allocation is done subject to a resource conflict graph among the requesting macrocell ports. The graph has an edge between two macrocell ports if and only if the two ports are active simultaneously, either because they are used for multiple operands of the same operation group mapped to that macrocell, or because the operation groups mapped to the two ports are concurrent in the archspec. In this manner, mutually exclusive operation groups mapped to the same or different macrocells would be able to share the register file ports for accessing their operands.

An important observation regarding concurrent use of register file ports is that the exact time

they are used (and hence cause the resource conflict) also depends on the individual input/output operand latency specified for the operation. For example, there may not be a resource conflict between the addend and either multiplicand of a three input multiple-add operation because the addend may be fetched one cycle later and hence can use the same register file port as one of the multiplicands. Similarly, the outputs of concurrently issued multiply operation with latency 3 and an add operation with latency 1 are not produced at the same time and hence may share the same write port. This effect is taken into account by including only those port request conflicts that correspond to the respective operands being accessed at the same latency.

The pseudo-code for the resource allocation algorithm appears in Figure 6. Our allocation heuristic is a variant of Chaitin's graph coloring register allocation heuristic [3]. Chaitin made the following observation. Suppose G is a conflict graph to be colored using k colors. Let n be any node in G having fewer than k neighbors, and let G' be the graph formed from G by removing node n. Now suppose there is a valid k-coloring of G'. We can extend this coloring to form a valid k-coloring of G by simply assigning to n one of the k colors not used by any neighbor of n; an unused color is guaranteed to exist since n has fewer than k neighbors. Stated another way, a node and its w neighbors can be colored with $w + 1$ or fewer colors.

Our formulation differs from Chaitin's in two important ways: first, we are trying to minimize the number of required colors, rather than trying to find a coloring within a hard limit; and second, our graph nodes have varying integer resource requirements. We generalize the reduction rule to non-unit resource requests by simply summing the resource requests of a node and its neighbors. In Figure 6, the total resource request for a node and its neighbors is computed by the first loop.

Our heuristic repeatedly reduces the graph by eliminating the node with the current lowest total resource request (node plus remaining neighbors). At each reduction step, we keep track of the worst-case resource limit needed to extend the coloring. If the minimum total resources required exceeds the current value of k, we increase k so that the reduction process can continue. The graph reduction is performed by the second loop in Figure 6.

Nodes are pushed onto a stack as they are removed from the graph. Once the graph is reduced to

```
procedure ResourceAlloc(IntVector request, Graph conflicts)
1:    // compute resource request for each node + neighbors
2:    for (n ∈ conflicts)  do
3:       mark[n] = false ;
4:       allocRes[n] = emptySet ;
5:       totalRequest[n] = request[n] + request[Nhbrs(n)] ;
6:    endfor
7:
8:    // sort nodes by increasing remaining total resource request,
9:    // compute upper-bound on resources needed by allocation
10:   int resNeeded = 0 ;
11:   NodeStack stack = emptyStack ;
12:   repeat NumNodes(conflicts)  times
13:      find unmarked node m such that totalRequest[m] is minimum ;
14:      mark[m] = true ;
15:      stack.push(m) ;
16:      resNeeded = max(resNeeded, totalRequest[m]) ;
17:      for (nhbr ∈ Nhbrs(m))  do
18:         totalRequest[nhbr] -= request[m] ;
19:      endfor
20:   endrepeat
21:
22:   // process nodes in reverse order (ie. decreasing total request)
23:   while (stack not empty)  do
24:      Node n = stack.pop() ;
25:      IntSet totalRes = { 0 .. resNeeded-1 } ;
26:      // available bits are those not already allocated to any neighbor
27:      IntSet availRes[n] = totalRes - allocRes[Nhbrs(n)] ;
28:
29:      // select requested number of bits from available positions
30:      // according of one of several heuristics
31:      AllocRes[n] = select request[n] bits from availRes[n] ;
32:            H1: contiguous allocation
33:            H2: affinity allocation
34:   endwhile
```

Figure 6: Pseudo-Code for resource allocation.

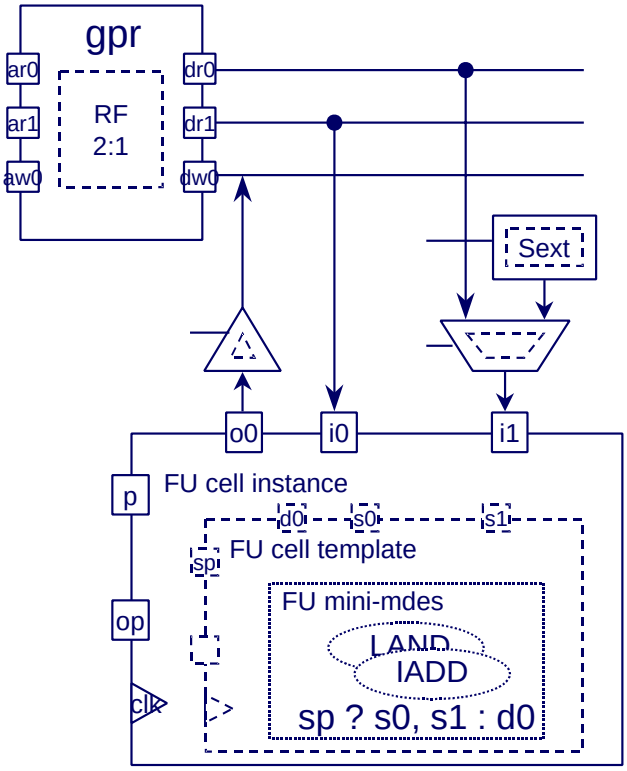

Figure 7: Datapath schema.

a single node, we begin allocating bit positions (resources) to nodes. Nodes are processed in stack order, i.e. reverse reduction order. At each step, a node is popped from the stack and added to the current conflict graph so that it conflicts with any neighbor from the original graph which is present in the current conflict graph. The existing allocation is extended by assigning bit positions to satisfy the current node's request, using bits disjoint from bits assigned to the current node's neighbors. This process is shown in the third loop in Figure 6.

4.3 Building the register files and the interconnect

Once the number of read/write ports of the various register files are determined we can allocate these files in hardware. The hardware component corresponding to the register files is also drawn from the macrocell database. Usually, such components are created using hardware generators

rather than enumerating all register files with different read/write porting structure. Integer literal files that consist of dense ranges of integers are not allocated in this manner since the instruction field representing the literal would be directly routed to the macrocell port. Literal files containing specially encoded literals (e.g., the value of π), however, may be allocated just like the regular register file in order to share the literal decoding hardware.

The register port allocation process also generates a tentative port assignment which identifies the exact register file port number to connect to each macrocell port. Since all register port of a given file and of a give type (read/write) are equivalent, it is possible to optimize this assignment based on layout or topological considerations for the interconnect. For example, if read-port 1 of a register file connects to two mutually exclusive macrocells, it may connect to the left or the right port of the macrocells independent of each other. However, constraining it to connect to the macrocells in a similar way (either to the left port or to the right port for both macrocells) so that only one instruction format field needs to drive the corresponding register address port.

The next step is to generate the interconnect by connecting up the macrocell ports to their assigned register ports as shown in Figure 7. A given macrocell port may be connected to several register ports (from different register files) which reflects a choice within the field type of a single operation as well as multiple operations with different field types mapped to the same functional unit. Similarly, a given register file port may be connected to several macrocell ports reflecting the mutual exclusion among them.

A multiplexor is used at the input port of a macrocell to accept input data from various register file read ports or literal instruction fields. On the other hand, a tristate buffer is used at the output port of a macrocell to drive each register file write port that it may output its data to. This is macrocells are allowed to write their data back to the register file only if their output is enabled, otherwise there should be no transaction on the register file write port.

All the wires left unconnected in Figure 7 are the datapath control ports that will later be connected to the instruction decode logic as shown in Section 6. These are the register file address ports, the multiplexor select ports, the tristate enable ports, and the macrocell opcode ports.

5 Mdes Extraction

The structural description of the machine as shown in Figure 7 is usually too detailed for a retargetable compiler to target directly, although such an approach has been tried in the past [14, 15]. Most retargetable compiler work with an abstraction of the micro-architecture, namely, the programming model of the machine. This model specifies the machine operations and their properties related to code selection and scheduling, as well as program-visible register state and their properties related to register allocation. Given a datapath design, such a programming model is usually directly specified as an auxiliary input specification [9, 10, 6]. In the PICO system, on the other hand, we automatically extract this programmatic view from the datapath in the form of a *machine-description* (*mdes* for short). In this section, we will describe how this is done.

5.1 Structure of the mdes

An in-depth description of mdes is presented in [16]. We outline its major components below. An mdes abstracts the following information from the datapath:

Opcode, Register, and Operation binding hierarchy – The operations visible to the compiler are organized within the mdes in a hierarchy starting from the generic operations (implementing the semantic operations of the program), down to the compiler operations (implemented by the architectural operation of the target machine). This hierarchy is also reflected into the opcodes and registers visible to the compiler ranging from the generic opcode and register sets to the compiler opcodes and registers respectively.

The various levels of the operation hierarchy include generic operations, access-equivalent operations, opcode-qualified operations, register-qualified operations, and fully-qualified operations that are organized in a partial lattice called the Operation Binding Lattice [16]. This hierarchy abstracts the structural aspects of the machine and allows the compiler to successively refine the binding of operations within the program from the semantic level to the

architectural level making choices at each level that are legal with respect to the target machine.

Operation descriptors – Operations at each level of the hierarchy are characterized by several properties that are also recorded within the mdes. These consist of the following.

Operation formats – Along with each operation, the mdes records the sets of registers and literals[4] that can source or sink its various input or output operands respectively. A tuple of such sets, one for each operand, is called the operation format. The size of these sets becomes larger as we climb the operation hierarchy, ranging from the exact set of registers accessible from each macrocell port implementing an operation at the architectural level to a set of virtual registers containing all architectural registers at the semantic level.

Latency descriptors – Each input and output operand of an operation specifies a set of latencies associated with its sample and production times respectively relative to the issue time of the operation. In addition, a few other latencies may be recorded based on the semantics of the operation (e.g. branch latency, or memory latency). These latencies are used during operation scheduling to avoid various kinds of timing hazards.

Resources and reservation tables – The various macrocells present in the datapath, the register file ports, and the interconnect between them are hardware resources that various operations share during execution. Other shared resources may include operation issue slots within the instruction register, pipeline stages or output ports within the macrocells. Each operation within the mdes carries a table of resource usages, called a reservation table, that contains all the resources used by the operation and the corresponding cycle times at which they are used relative to the start of the operation. This table is used during operation scheduling to avoid hazards due to sharing of resources.

[4] As far as the mdes is concerned, literal operands are treated just like any other register operands albeit with a fixed value. Therefore, from now on, we will use the term "registers" to include literal operands as well.

Opcode descriptors – The structural and semantic properties of opcodes at each level of the hierarchy are also kept within the mdes. These properties include its semantic opcode name, the number of input and output operands, whether or not the opcode can be speculated and/or predicated, whether or not it is associative, commutative etc.

Register descriptors – Similarly, several properties of registers and register files are recorded at each level of the operation hierarchy including the bit-width, whether or not speculative execution is supported, whether the register (or register file) is static, rotating, has literals etc.

5.2 Mini-mdes components

In order to facilitate mdes extraction directly from the datapath components, each macrocell in the macrocell database carries a *mini-mdes* which records the mdes-related properties shown above for the architectural opcodes that it implements. The mini-mdes is organized just as described above except that it contains only one level of the operation hierarchy, the architectural level, and that there are no registers and register descriptors. Instead, the operation format of an architectural operation is described in terms of the input/output ports of the macrocell used by each of its operands. This "OperandToMcellPort" mapping is also used during register file port allocation (Section 4.2) in order to determine which ports of the macrocell need to connect to which register files.

For each operand of a given operation, the mini-mdes also records the various internal latencies through the macrocell in its latency descriptor. If the macrocell is a hard macro, the latencies may be accurately modeled as absolute time delay (nanoseconds), or in case of soft macros, approximately as the number of clock cycles relative to the start of the execution of the operation.

For each operation, the mini-mdes records any shared internal resources (e.g. output ports, internal buses) and their time of use relative to the start of the execution in an internal reservation table. This table records internal resource conflicts and timing hazards between operations. For example, if a macrocell supports multiple operations with different output latencies that are channeled through the same output port, there may be a output port conflict between such operations issued

successively to this macrocell. Recording the usage of the output port at the appropriate time for each operation allows the compiler to separate such operations sufficiently in time so as to avoid the port conflict.

Finally, the mini-mdes of a macrocell also reflects whether the macrocell implements speculative and/or predicated execution capability by incorporating such opcodes within itself. The macrocell selection process shown in Section 4.1 may choose macrocells based on the presence or absence of such capabilities. Note that a macrocell supporting speculative execution and/or predicated execution may be used in place of one that does not, but its cost may be somewhat higher.

5.3 Extracting global mdes from the datapath

The process of extracting a compiler-centric machine description from the datapath of the machine is described by the **ExtractMdes** pseudo-code shown in Figure 8. The key idea here is to collect the information contained in the mini-mdeses of the various functional unit macrocells and the mdes-related properties of the register files present in the datapath into a single global mdes, and augment it with the topological constraints of the datapath such as connectivity to shared buses and register file ports.

The extraction process starts by initializing the global mdes of the machine to an empty mdes (Line 1). Then, for each component of the datapath that is a functional unit macrocell, the extractor installs its mini-mdes architectural opcodes as compiler opcodes within the global mdes to form the lowest level of the opcode hierarchy (Line 9). Various semantic and structural properties of the opcode including semantic opcode name, commutativity, associativity, number of input and output operands, bit encoding etc. are also copied into the corresponding opcode descriptor.

Likewise, for register file components of the datapath, the extractor installs the various architectural registers as compiler registers into the global mdes to form the lowest level of the register hierarchy along with a register descriptor (Line 30) that records the structural properties of the register file. Most of these properties are determined either from the type of the hardware component used (*e.g.*, whether or not speculative execution and/or rotating registers are supported), or from its

```
procedure ExtractMdes(Datapath dpath)
1:    Mdes globalMdes = nullMdes;
2:    for (component ∈ dpath)  do
3:       if (component is a FU macrocell)  then
4:          PortAltMap altMap = nullMap ;
5:          Mdes miniMdes = component.MiniMdes() ;
6:          // accumulate the mini-mdes operations into the global mdes
7:          for (operation ∈ miniMdes)  do
8:             CompilerOpcode opcode = a copy of operation.opcode() ;
9:             globalMdes.InstallOpcode(opcode) ;
10:            for (each input/output operand of operation)  do
11:               OperandAlts opdAlts = nullList ;
12:               ReservationTable opdResv = nullTable ;
13:               OperandLatency lat = a copy of operation.OpdLatency(operand) ;
14:               McellPort port = operation.OperandToMcellPort(operand) ;
15:               // accumulate mdes properties by traversing the datapath from this port
16:               if (this port has not been traversed before)  then
17:                  TraversePort(port, lat, opdResv, opdAlts) ;
18:                  // save operand alternatives for this port
19:                  altMap.bind(port, opdAlts) ;
20:               else
21:                  opdAlts = altMap.value(port) ;
22:               endif
23:               opcode.RecordOperandAlternatives(operand, opdAlts) ;
24:            endfor
25:            // build operation alternatives as a cross product of operand alternatives
26:            opcode.BuildOperationAlternatives(operation) ;
27:         endfor
28:      else if (component is a register file)  then
29:         // accumulate register file properties into the global mdes
30:         globalMdes.InstallRegisterFile(component) ;
31:      endif
32:   endfor
33:   // build a hierarchy of operation alternatives for each semantic operation
34:   BuildOperationHierarchy(globalMdes) ;
35:   return globalMdes ;
```

Figure 8: Pseudo-Code for extracting mdes from the datapath.

structural instance parameters (*e.g.*, the number and bit-width of static and rotating registers). A few remaining properties are carried forward from the archspec (*e.g.*, the virtual file type).

The mdes-related details of the operations implemented by a functional unit macrocell are collected as follows. For each input or output operand of a machine operation, the extractor collects a set of "operand alternatives". This set is obtained by first mapping the operand to its corresponding macrocell port at which it is received or produced (method call **OperandToMcellPort** at Line 14), and then traversing the datapath components connected to that port (procedure call **TraversePort** at Line 17). Operands mapped to the same port share the same alternatives and hence datapath traversal needs to be performed only once per port. The details of this traversal and the generated operand alternatives are provided later.

Next, the sets of operand alternatives obtained as above are combined into "operation alternatives" (method call **BuildOperationAlternatives** at Line 26). This is done by taking each tuple in the Cartesian product of the sets of operand alternatives for the given operation and combining its operand properties to form operation properties. The operand field types are concatenated to form an operation format, individual operand latencies are collected to form the complete operation latency descriptor, and the operand reservation tables are combined together with the internal reservation table of the operation into an overall reservation table for that operation alternative. As described below, the field types of the various operand alternatives partition the compiler registers of the machine into access-equivalent register sets. Therefore, the operation alternatives formed above correspond to an opcode-qualified compiler operation consisting of a compiler opcode and a set of access-equivalent register-set tuples. All such distinct operation alternatives are installed into the global mdes as alternatives for the given compiler opcode.

5.3.1 Datapath traversal

The heart of the above mdes extraction scheme is the datapath traversal routine **TraversePort** shown in Figure 9 which extracts the operand alternatives associated with a given functional unit macrocell port. We only show the input port traversal since it is symmetric for output ports. For

```
procedure TraversePort(McellPort thisport, OperandLatency lat, ReservationTable resv, OperandAlts opdAlts)
1:    // Assume one-to-one connections among ports
2:    if (thisport is INPUT port)  then
3:       case (predecessor component connected to thisport)  of
4:       multiplexor: // accumulate all field type choices
5:          for (each inputport of the multiplexor)  do
6:             TraversePort(inputport, lat, resv, opdAlts) ;
7:          endfor
8:       de-multiplexor: // add a resource column to reservation table
9:          Resource res = Resource(inputport of the de-multiplexor) ;
10:         ReservationTable resv' = resv.AddColumn(res, lat) ;
11:         TraversePort(inputport, lat, resv', opdAlts) ;
12:      pipeline latch: // add one to latency
13:         Identify inputport of the latch ;
14:         ReservationTable resv' = resv.AddRow(lat) ;
15:         OperandLatency lat' = lat.AddLatency(1) ;
16:         TraversePort(inputport, lat', resv', opdAlts) ;
17:      register/literal file: // base case
18:         FieldType ftype = FieldType(file.Registers()) ;
19:         Resource res = Resource(outputport of the register file) ;
20:         ReservationTable resv' = resv.AddColumn(res, lat) ;
21:         opdAlts.addAlt(ftype, lat, resv') ;
22:      endcase
23:   else // thisport is OUTPUT port (symmetric cases)
24:      ...
25:   endif
```

Figure 9: Pseudo-Code for accumulating mdes properties while traversing the datapath.

simplicity, we also assume that only one-to-one connections exist between the input and output ports of various datapath components, i.e., multiple sources to an input port are connected via a multiplexor, and multiple sinks from an output port are connected via a de-multiplexor. It is easy to extend this to many-to-many connections by treating such connections as multiple sources multiplexed onto a bus that are de-multiplexed to the various sinks.

Each operand alternative is a triple consisting of the following information that characterize the macrocell port and the hardware structures surrounding it:

1. The field type of the operand, which describes a set of compiler registers that are the potential

sources of the operand and that are equally accessible from the input port.

2. The operand latency descriptor, which contains the earliest and latest sampling latencies of the operand with respect to the issue time of the operation. This may be different for different sources reaching this port or even for the same sources reachable via different paths.

3. The operand reservation table, which identifies any shared resources used for accessing this operand (*e.g.*, buses and register file ports) and their time of use relative to the issue time of the operation.

The strategy for collecting the operand alternatives for a given macrocell port is as follows. The operand latency of the various alternatives is initialized using the macrocell mini-mdes and their reservation table is set to empty. Starting from the macrocell port, the extractor then traverses the various datapath components connected to it in a depth-first traversal until an operand source such as a register file or literal instruction field is reached. As hardware components such as multiplexors, de-multipelxors, pipeline latches and registers files are encountered during the traversal, their effect is accumulated into the operand latency and the reservation table as described below.

A multiplexor (Line 4) at the input port serves to bring various sources of this operand to this port and therefore represents alternate field types and latency paths leading to different operation alternatives. We perform a recursive traversal for each of the inputs of the multiplexor.

The effect of a de-multiplexor (Line 8) at the input is to distribute data from a shared point (such as a shared input bus) to various macrocell ports. This is modeled by introducing a new resource column in the reservation table corresponding to this shared data source. A check is placed at the current latency to show that this new resource is used at that latency. The input of the de-multiplexor is followed recursively.

A pipeline latch (Line 12) encountered during the traversal adds to the sampling latency of the operand as well as affects the operation reservation table by adding a new row at the beginning. The input of the latch is recursively traversed to identify the source of the operand.

Finally, a register file port or a literal instruction field (Line 17) is the point where the recursion

terminates. All the registers (literals) accessible via the register file port (literal field) become part of the field type of the operand. The register file port (literal field) itself is recorded as a shared resource being accessed at the current latency by adding a resource column to the current reservation table. The triple consisting of the field type, the operand latency, and the reservation table is accumulated into the list of operand alternatives for this macrocell port.

5.3.2 Building operation hierarchy

The final step in the mdes extraction process is to complete the higher levels of the opcode, register and operation hierarchy within the global mdes (procedure call **BuildOperationHierarchy** at Line 34 of Figure 8). This process is discussed below.

The process of constructing operand alternatives shown above already identifies the compiler registers, and the access-equivalent register sets. In order to complete the register hierarchy, all distinct access-equivalent register sets are collected to form a generic register set which implements the semantic notion of a virtual register in the program.

Next, the corresponding levels in the opcode hierarchy are constructed using the register hierarchy. First, all compiler opcodes implementing the same semantic opcode (as identified by its opcode property) are collected into a generic opcode set which forms the top layer of the opcode hierarchy. Any operation alternative pointed to by a compiler opcode within this generic opcode set is a valid implementation of the corresponding semantic operation. However, not all such alternatives are equivalent in terms of their operand accessibility. Therefore, the set of operation alternatives pointed to by a generic opcode set is then further partitioned into sets of access-equivalent alternatives that use the same access-equivalent register-set tuples. The compiler opcodes present in each such partition form a distinct access-equivalent opcode set which constitutes the middle layer of the opcode hierarchy.

Finally, the missing layers of the operation hierarchy, *i.e.*, generic operation sets, access-equivalent operation sets, and register-qualified operation sets may be built using the corresponding layers of the opcode and the register hierarchies. In the current implementation, these layers are not directly

represented, instead they are implicitly referenced via the opcode hierarchy.

5.4 Example

Figure 7 may be used as an example to show how the traversal routine works. The datapath shows one instance of an ALU macrocell. The operation group mapped to this instance contains two operations, LAND (logical and) and IADD (integer add). The operation format for these operations stored within the mini-mdes shows the macrocell ports used for their various operands. The mini-mdes also records the sampling and production times of the various input and output operands that is intrinsic to the macrocell. Let us suppose that it is 0 for each data input s0 and s1, 1 for the predicate input sp, and 2 for the data output d0 (assuming that the macrocell is pipelined). Finally, the mini-mdes records that these operations execute on the same macrocell and share its computation resources.

The datapath traversal starts from the actual input and output ports of the macrocell instance. Following input port i0, we find that it is directly connected to the gpr register file port dr1, introducing a shared resource column for that register port to be used at cycle 0, which is the sampling latency of this input operand. The field type accessible via this port is denoted by “gpr” which stands for all the registers contained in the register file gpr. This operand alternative is recorded temporarily.

The input port i1 of the macrocell instance is connected via a multiplexor to the gpr register file port dr0 as well as a sign-extender for the short literal instruction field. This gives rise to two distinct operand alternatives, one with field type “gpr” at latency 0 using the gpr file port dr0, and the other with field type “s” at latency 0 using the literal instruction field connected to the sign-extender. Similarly, the predicate input gives rise to the operand alternative with field type “pr” at latency 1 using the pr file port (not shown), and the destination port o0 gives rise to the operand alternative with field type “gpr” at latency 2 using the gpr file port dw0. The various operand alternatives are combined to form two distinct operation format and reservation table combinations as shown below.

"pr ? gpr, gpr : gpr"

Cycle	Resource Usages					
	ALU	pr0	dr0	dr1	dw0	lit
0	X		X	X		
1		X				
2					X	

"pr ? gpr, s : gpr"

Cycle	Resource Usages					
	ALU	pr0	dr0	dr1	dw0	lit
0	X			X		X
1		X				
2					X	

Note that the overall latencies of the operands are the same as the intrinsic macrocell port usage latencies since there are no external pipeline latches. Also, the ALU resource is marked as being used only at cycle 0 since the macrocell is pipelined and the usage of subsequent stages of the ALU pipeline at subsequent cycles is implicit. The above combinations of operation formats, latencies, and reservation tables apply to both IADD and LAND opcodes, thereby forming two distinct operation alternatives each. These alternatives would be combined with other alternatives from other macrocells to give rise to the complete operation hierarchy for these opcodes.

Operation issue conflicts stemming from the instruction format may also be added to the above reservation tables in the following way. We repeat the above exercise after the instruction format for the target machine has been designed and the corresponding controlpath and instruction decode logic has been inserted (described in the next section). Now, the datapath traversal would need to be carried through the register files back up to the instruction register treating the register files like pipeline latches. The latency of the register files may cause one or more rows to be added at the beginning of the reservation table automatically corresponding to instruction decode and operand fetch cycles. The traversal paths leading towards the same bit positions in the instruction register would end up recording an operation issue conflict.

Alternatively, one may directly represent the operation group exclusions prescribed in the archspec (Section 3.1) as shared abstract resources that are used at cycle 0 and, therefore, model operation issue conflict for the mutually exclusive operation groups. The PICO system currently uses this mechanism since it is much simpler and it de-couples the extraction of the mdes and its use in scheduling application programs from instruction format design and controlpath insertion pro-

cesses.

6 Controlpath Design

The controlpath of a processor is responsible for controlling the various control ports in the datapath at each cycle by providing a mechanism to interpret and sequence the various operations of an object program in a meaningful and efficient manner. The controlpath design poses two major challenges. One is that the multi-template instruction format, which is in response to our emphasis on minimizing code size, complicates the design of the instruction unit. Since the templates are of various lengths, the instruction fetch path must include an instruction alignment network so that, on each cycle, the next instruction is left-aligned in the instruction register. Although a generic architecture for the alignment network can be defined, the details of the design are, necessarily, intimately tied to the specific instruction format that has been generated. The second problem is that an instruction cannot, in general, be fetched, decoded and distributed within a single cycle. The instruction fetch must be pipelined. This is exacerbated by the presence of the alignment network. If the processor is to be capable of issuing an instruction on each cycle, PICO-VLIW must provide it with a sophisticated instruction prefetch unit. In this section, we describe the controlpath design scheme used in PICO-VLIW that takes care of the above issues.

We divide the controlpath design task into two major subtasks:

Instruction pipeline design – This involves providing a mechanism to fetch program instructions from system memory (typically, from an instruction cache, or *I-cache* for short) at a consistent rate, to decode the various operations within each instruction, and to distribute the decoded control signals to the various control ports in the datapath. The branch functional unit may interact with the controlpath to alter the sequence of operations periodically.

Instruction sequencer design – Additional sequencing mechanisms may be required to increase the functionality of the processor or to improve its performance. These include interrupt and exception handling, error recovery, branch prediction, data speculation etc.

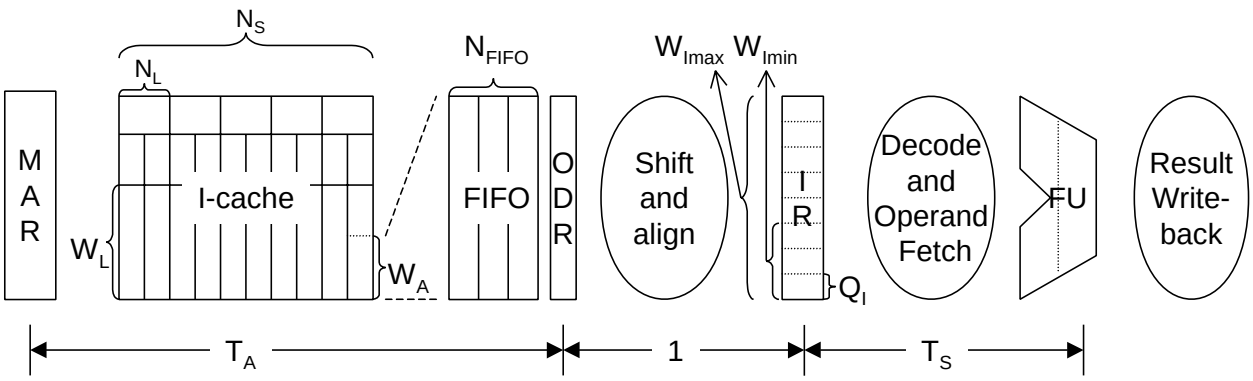

Figure 10: The instruction fetch/execute pipeline.

In the PICO design system, we concentrate on the instruction pipeline design rather than the sequencer design. This is because the design of the instruction pipeline is crucially dependent upon the datapath and the instruction format design and, therefore, must be automated as part of the overall design of the processor. The sequencer mechanisms, on the other hand, are well understood controlpath enhancements that may be added to the design independently. Each of these mechanisms either leave the instruction pipeline design unchanged or transform it in a pre-determined way that can be shown to be equivalent in semantics to the original pipeline [2]. Therefore, we assume that we have the desired sequencer macrocell (or a set of macrocells) and controlpath extensions available in the macrocell database which can be chosen and applied to the controlpath design after the instruction pipeline has been designed in an archspec-driven manner. We will, however, describe the interface between the instruction pipeline and the sequencer in order to show how the sequencer fits into our overall controlpath schema.

6.1 Instruction pipeline scheme

The abstract structure of the instruction pipeline is shown in Figure 10. This schema is parameterized across various aspects of the design that are chosen according to the archspec, the instruction format, and the datapath. The details of the instruction format design are provided in the companion technical report [1]. However, we will define the parameters relevant to the controlpath design

as we go along.

During instruction format design, all instruction templates are adjusted to be of length which is some integral multiple of a minimum size called a *quantum*, Q_I, measured in bytes. This is also the address boundary to which an instruction is aligned in memory. Usually the quantum size is a power of 2, otherwise there is no benefit in aligning instructions to that boundary. For example, if the quantum size is 4 bytes, then all instructions consist of one or more 32-bit words and must be word-aligned in program memory. The *minimum width* of an instruction template is denoted by W_{Imin} in units of quanta. Likewise, the *maximum width* of an instruction template is denoted by W_{Imax} in units of quanta. Branch targets may have an additional constraint of being aligned to a different address boundary Q_T which must be an integral multiple of Q_I.

The instruction cache is parameterized by its cache line size W_L (measured in quanta), the degree of associativity (the number of lines per set) N_L, and the number of sets it contains N_S. The total size of the cache in bytes is, therefore, $N_S \times N_L \times W_L \times Q_I$. The cache line size is further restricted to be a power of 2 in order to simplify the path to/from main memory (or second-level cache). On the processor side, instructions are fetched in units of *instruction packets* of size W_A (measured in quanta). Currently, we have the restriction that a cache line must contain an integral number of instruction packets, avoiding packet wraparound into multiple cache lines. The instruction packet is also required to be larger than the maximum width instruction ($W_A \geq W_{Imax}$). This ensures that no more than one packet may be consumed at any cycle. Both these restrictions may be relaxed at the expense of more complicated instruction fetch mechanism that may cause stalling during sequential instruction fetch. Another parameter of the instruction cache is its access time, T_A, which is the number of cycles it takes from the point an address is provided to the point when the instruction packet is ready to be latched at the output of the instruction cache.

Aside from instruction pipeline stalls, at each cycle we want the next instruction in sequence to be latched left-aligned into the instruction register (*IR* for short), whose length, W_{IR}, must be large enough to hold the maximum size instruction ($W_{IR} \geq W_{Imax}$).[5] Since instructions may have various widths, a complete instruction packet may not be consumed every cycle. Therefore,

[5] In fact, in order to reduce the overall cost of our designs we assume that $W_{IR} = W_{Imax}$.

an instruction prefetch buffer (FIFO) is used to buffer the packets as they are fetched from the instruction cache as shown in Figure 10. The length of this buffer, N_{FIFO}, is computed during the pipeline design process as described below. We also need a shift-and-align network in order to correctly align each instruction to the leftmost position in the instruction register based on the width of the previous instruction. Design of this network is also discussed below. Finally, an additional latch, called the *on-deck register* (*ODR* for short), is inserted between the prefetch buffer and the alignment network to provide additional slack to cover the latency of the alignment network.

6.2 Using the instruction pipeline

Before we describe the concrete design of the various components of the pipeline, it is useful to understand its behavior which illustrates several design tradeoffs.

6.2.1 Determining the branch latency

Figure 10 shows how the branch latency, T_B, is computed. The branch latency of the machine is the number of cycles elapsed from the time at which the branch instruction is issued to the time at which the branch target instruction is issued (assuming no speculation). This latency is used during instruction scheduling and is added to the set of mdes parameters [16]. Looking at Figure 10, it is clear that the branch latency is a combination of the execution latency of the branch operation, T_S, which is the time at which the branch address is resolved within the branch functional unit relative to the issue time of the branch operation, and the instruction cache access time, T_A. When the target packet is ready, it is latched directly into the ODR, skipping over the prefetch buffer. The packet is latched into the IR one cycle later. Therefore, the hardware branch latency in PICO designs is computed as follows.

$$T_B = T_S + T_A + 1$$

We choose to expose this latency in the mdes directly. It is also possible to mandate a smaller program visible latency by inserting a execution pipeline stall mechanism together with a branch prediction mechanism. These mechanisms help to hide the branch latency in the most frequent

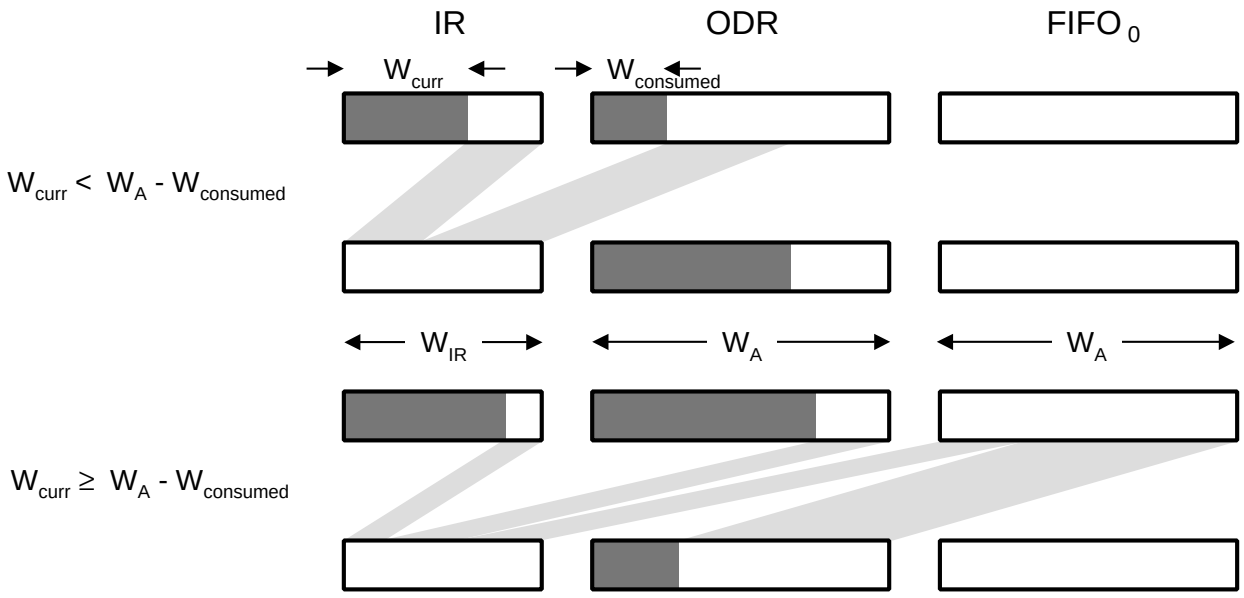

Figure 11: Sequential instruction fetching.

cases by correctly predicting the branch target and prefetching it so that there is no bubble in the pipeline.

6.2.2 Sequential instruction fetching

Figure 11 shows how the instruction register is filled with useful bits during sequential instruction fetch. The width of the current instruction in the IR is denoted by W_{curr} measured in units of quanta. These are the instruction register bits that have been consumed in the current cycle. In general, some initial prefix in the ODR, $W_{consumed}$, may also have been consumed by previous instructions. In order to prepare the IR for the next cycle, we need to shift the unused bits in the IR to the left and fill the remaining bits from the unused bits of the ODR ($W_{curr} < W_A - W_{consumed}$) and, possibly, from the prefetch buffer ($W_{curr} \geq W_A - W_{consumed}$). If the bits in the ODR are completely consumed in this process, the prefetch buffer is also advanced by one packet.

A potential stall situation in the above scheme arises when the IR and the ODR are both almost fully consumed and there are no instruction packet requests outstanding in the instruction cache.

If we assume that an instruction packet is just as big as the maximum sized instruction, i.e., $W_A = W_{Imax}$, and a series of maximum sized instructions are encountered, the prefetch buffer should have enough packets prefetched to cover up the cache access latency T_A, or else stall cycles may have to inserted. A buffer length of T_A would ensures that by the time the prefetched packets are all consumed at full rate ($W_{Imax} = W_A$ per cycle), the next packet from the cache would be ready to be dispatched. On the other hand, typically the instruction packet size is rounded up to the next power of 2 in order to fit a cache line properly so this situation may not arise in practice.

If an instruction consists entirely of no-ops, it may be eliminated entirely from the instruction stream and the number of no-op cycles is encoded as a count into the previous instruction in a *multi-noop* field. This effectively amounts to introducing a controlled number of bubbles in the instruction issue pipeline which is handled by the pipeline control logic.

6.2.3 Branch target fetching

Processing a branch instruction introduces some more design issues. As discussed earlier, on encountering a branch that is resolved to be taken, all the unused bits in the IR, the ODR and the prefetch buffer must be thrown away. Instead, fetching is initiated from the branch target address. After the branch latency, the instruction packet containing the branch target is loaded directly into the ODR from the I-cache as shown in Figure 12. However, the actual target instruction may be displaced by some amount P_{target} from the packet boundary. Depending on the amount of displacement, either enough bits may be available to load the IR ($W_{IR} \leq W_A - P_{target}$), or the next packet may also be required to be fetched in order to bring enough bits into the IR ($W_{IR} > W_A - P_{target}$), further adding to the branch latency and making it target dependent.

It is possible to eliminate this problem entirely by ensuring during program assembly that branch target instructions do not span an instruction packet boundary, or better still, that branch targets are aligned to the instruction packet boundary, i.e., $Q_T = W_A$. The latter constraint eliminates the initial target displacement P_{target} entirely. In either case, this constraint may introduce *holes* in the code space between the branch target instruction and the immediately preceding instruction

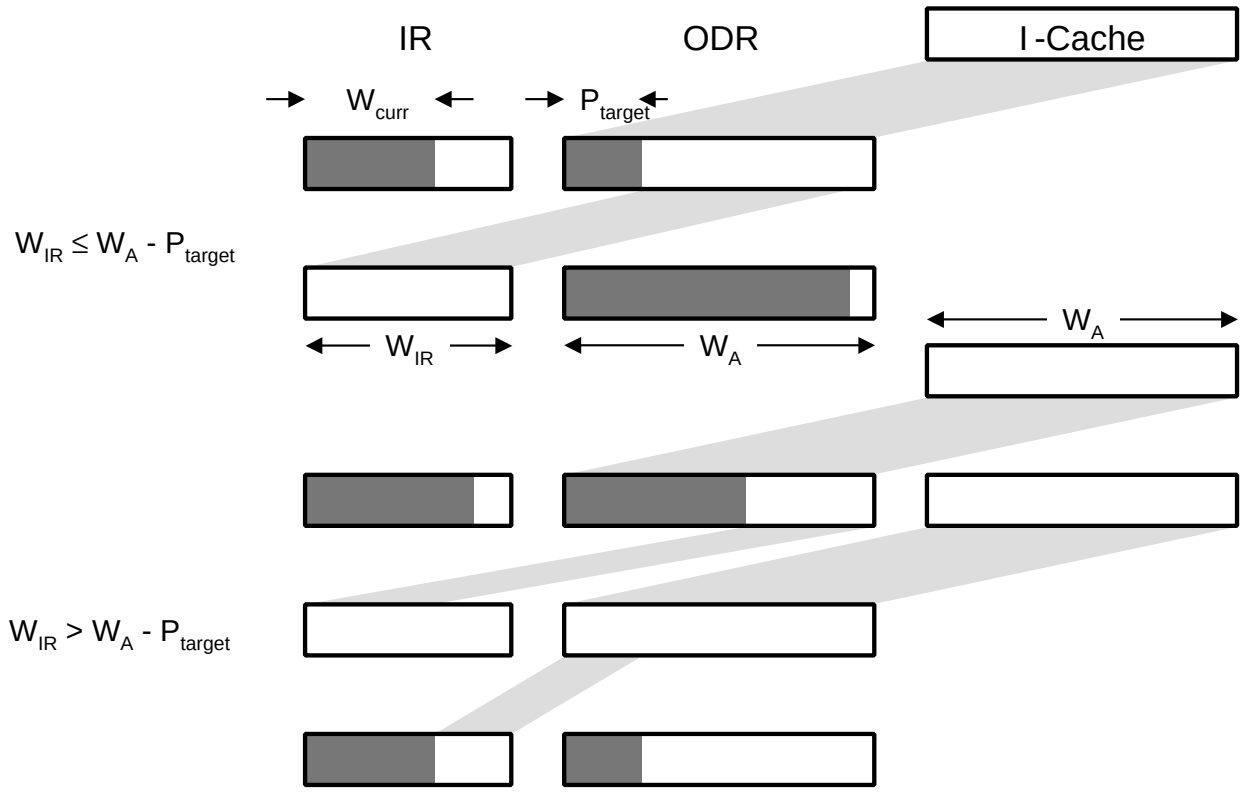

Figure 12: Branch target fetching.

that must be skipped during a fall-through branch. We accomplish this by adding a *consume-to-end-of-packet* bit to each instruction that, if set to 1, effectively increases the width of the current instruction to the next instruction packet boundary.

The above scheme demonstrates a tradeoff between the code size and the complexity of the instruction pipeline where we have chosen to simplifying the pipeline at the expense of some code space.

6.3 Designing the instruction pipeline

We will now describe how the various components of the above pipeline schema are designed and interconnected. Figure 13 shows the overall controlpath design schema. The figure also shows how the various datapath control ports that were left hanging in Figure 7 are now connected to the

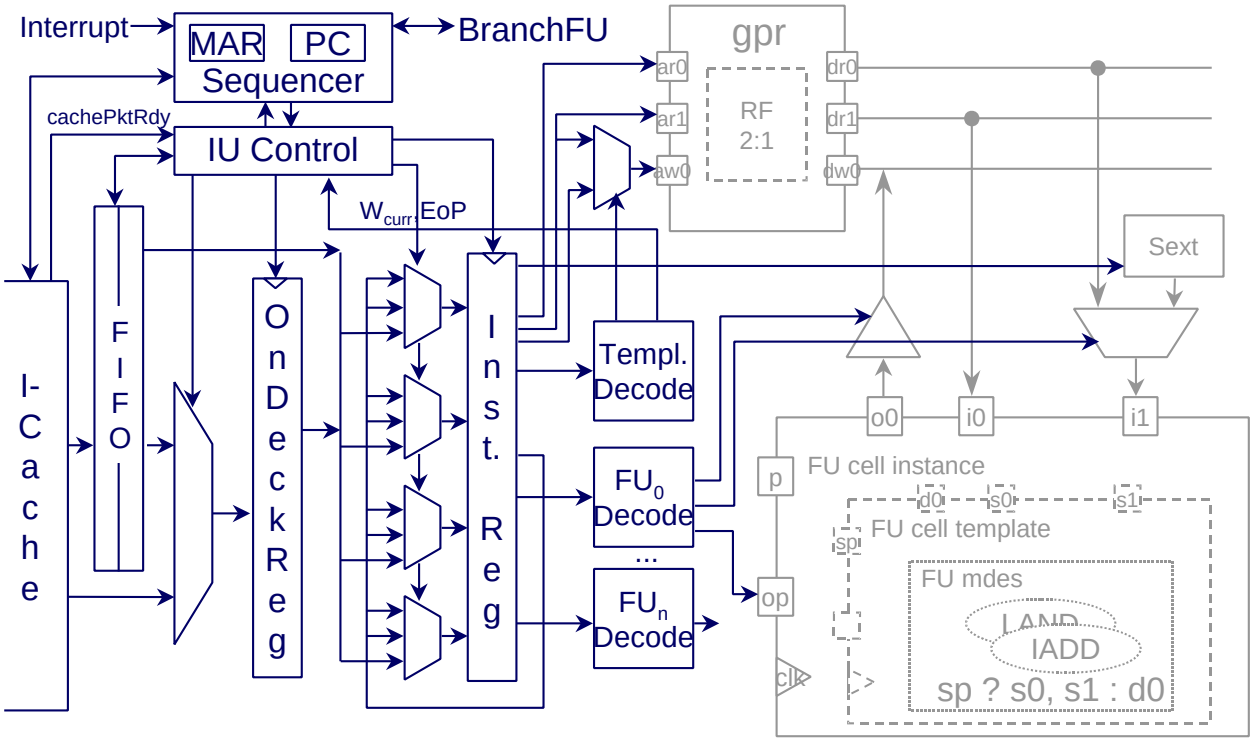

Figure 13: Controlpath schema.

appropriate control signals taken either directly from the output of the IR (e.g. register file address bits), or from the output of the control logic that decodes the current instruction.

The instruction pipeline consist of the following major components that are discussed below.

1. The instruction cache
2. The instruction prefetch buffer
3. The shift-and-align network
4. The instruction unit control logic
5. The instruction decode logic

6.3.1 Instruction Cache

The design of the instruction cache is done as part of designing the memory hierarchy and is outside the scope of this paper. This involves choosing its number of ports, line size, associativity, overall size, addressability etc. based on both architectural requirements and the application characteristics.

For the purpose of the controlpath design, the relevant instruction cache parameters that are needed are the fetch packet width W_A and the cache access time T_A. The fetch packet width determines the width of the instruction prefetch buffer and the access time is used to determine its depth as described below.

6.3.2 Instruction prefetch buffer

The purpose of the prefetch buffer is to smooth out the variability of the multiple sized instructions ($W_{Imin} \cdots W_{Imax}$) issued from the instruction register and match it to the fixed sized instruction packet (W_A) fetched from the instruction cache, balancing the overall instruction throughput. On the one hand, the prefetch buffer immediately provides the additional bytes needed to fill up the instruction register when a short instruction is followed by a series of long ones, while on the other hand, it buffers the already requested packets from the instruction cache when the rate of consumption falls due to encountering a series of short instructions.

The width of the prefetch buffer is matched to the instruction packet size in order to allow direct transfer of data from the instruction cache. The required depth of the buffer depends on the instruction fetch policy, namely, how is the inventory of packets being fetched managed. The *inventory* is defined as the sum of the number of packets already present in the buffer and those in flight in the instruction cache pipeline. Our instruction fetch policy is to keep this number at a constant by issuing just enough fetches at the right time to cover up the cache access latency while accommodating the variable rate of consumption of instruction bits. The number of packets needed in the inventory in the worst case, therefore, is given by $\lceil T_A \times W_{imax}/W_A \rceil$, which accounts for the maximum sized instructions being issued at each cycle.

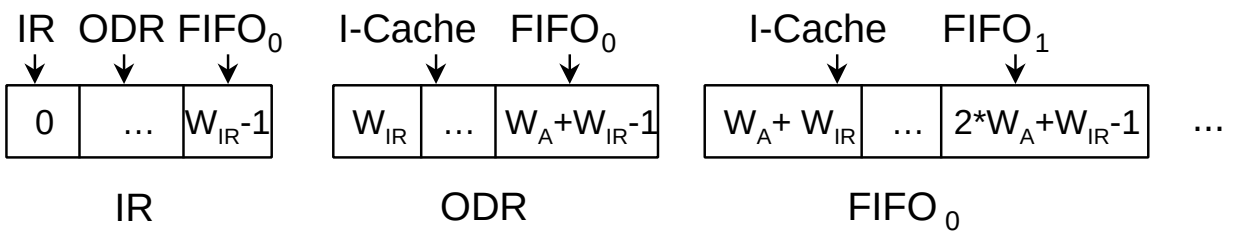

Figure 14: Treating IR, ODR, and FIFO as a logical shift register.

The inventory size is an upper bound for the prefetch buffer depth because, by definition, there are no other packets in the instruction pipeline. It is also a lower bound if the processor is allowed to stall due to some reason (e.g. data cache miss) because the packets in flight in the instruction cache may need to be drained into the prefetch buffer. Therefore, the depth of the prefetch buffer is given by,

$$N_{FIFO} = \lceil T_A \times W_{imax}/W_A \rceil$$

However, if the processor is not allowed to stall due to any reason, some fraction of the inventory may always be kept in flight and the actual buffer size needed may be reduced.

6.3.3 Instruction alignment network

The heart of the instruction pipeline is the shift-and-align network that is responsible for bringing the correct set of bits left-aligned into the instruction register at each cycle. The sequential and branch target fetching schema described above illustrate the various cases that need to be considered.

Since the length of each instruction is always some multiple of quantum width, Q_I, we need to shift the bits in the IR only by these amounts and not by every bit position. As shown in Figure 13, this is accomplished by using a set of parallel multiplexors at the input of the IR for each quantum sized set of bits. The inputs to these multiplexors come from the corresponding bit positions in other quanta stored in the IR, the ODR, or the head of the FIFO.

In order to define precisely which multiplexor inputs come from which quantum bits, we treat the IR, ODR and the FIFO as a single logical shift register as shown in Figure 14. Each quantum sized set of IR multiplexors selects among some range of quanta to the right depending on the size of the current instruction. Specifically, the k^{th} quantum multiplexors ($0 \leq k \leq W_{IR} - 1$) select among the quanta between,

$$\begin{cases} k + W_{Imin}, & \text{when the last instruction was minimum size} \\ k + W_A + W_{IR} - 1, & \text{when the last instruction was maximum size and all of ODR was consumed} \end{cases}$$

Note that we have assumed $W_{IR} = W_{Imax}$ in the second case. Once W_A, W_{Imin} and W_{Imax} are defined at design time, the above range of inputs is statically fixed for each value of k. Therefore, the corresponding network of connections can be generated at design time.

At run time, the multiplexor for each quantum has to be provided with the appropriate select control depending on the width of the current instruction. Looking at Figures 11 and 12, the multiplexor select control for the k^{th} quantum multiplexors is given by,

$$\begin{cases} k + W_{curr}, & \text{sequential fetching, } k + W_{curr} < W_{IR} \\ k + W_{curr} + W_{consumed}, & \text{sequential fetching, } k + W_{curr} \geq W_{IR} \\ k + W_{IR} + P_{target}, & \text{branch target fetching} \end{cases}$$

The only dynamic quantities in the above computation are W_{curr} and $W_{consumed}$ that change as various instructions are fetched and decoded. At design time, we can generate pipeline control logic for each value of k that selects the appropriate case on the basis of given values of W_{curr} and $W_{consumed}$.

The bits in the ODR and the FIFO, on the other hand, need only be advanced by a full packet size, W_A. The ODR accepts its packet from either the head of the FIFO or the instruction cache directly, while the FIFO advances by one and, possibly, accepts a new packet from the instruction cache. On certain cycles, the ODR and the FIFO may not need to be advanced at all because the unused bits in the IR and the ODR are sufficient to prepare the IR for the next cycle. The selection control for the multiplexor at the input of the ODR and the advancement control of the FIFO are also part of the pipeline control logic as discussed below.

6.3.4 Instruction unit control logic

The instruction fetch policy described in Section 6.2 is implemented in control logic that keeps track of the packet inventory – the packets in flight, packets in the prefetch buffer, and the un-consumed part of the ODR. It also issues instruction cache fetch requests, FIFO load and advance requests, and ODR load request at the appropriate times, and provides the appropriate selection control for the shift and align network and other multiplexors in the instruction pipeline. Finally, the control logic is also responsible for flushing or stalling the pipeline upon request from the sequencer due to a branch or an interrupt, or from the instruction decode logic due to multi-noops.

The control logic may be expressed as pseudo-code as shown in Figure 15 that consists of a set of conditions and various actions to be performed under those conditions. The logic keeps track of the inventory of packets internally including those in flight in the instruction cache pipeline (numCachePkts) and those sitting in the prefetch buffer (numFIFOPkts). This is used to issue a fetch request whenever the inventory size falls below the threshold (Line 3). The corresponding instruction packet is ready to be read at the output of the cache T_A cycles after the fetch is initiated (Line 7). This packet may be loaded directly into the ODR if the rest of the pipeline is empty (Line 9), or it may be saved in the FIFO (Line 12). These packets are later loaded into the ODR as needed (Line 17).

Upon encountering a taken branch signal or an interrupt signal from the sequencer (flushPipe), the control logic flushes the instruction pipeline by reseting the internal state (Line 23). This enables the pipeline to start fetching instructions from the new address from the next cycle. Otherwise, the next instruction in sequence needs to be aligned into the instruction register (Line 28) after suitable number of multi-noop cycles. If the end-of-packet (EOP) bit is set, the current packet residing in the ODR is considered to be fully consumed and the IR is shifted to the next packet available. Otherwise, the IR is shifted by the width of the current instruction W_{curr}. In either case, the multiplexors of the shift and alignment network in front of the IR are provided with the appropriate selection control as described in the last section.

The control logic shown above may be translated to a formal hardware specification language such

```
module IUControl(bool cachePktRdy, bool flushPipe, bool EOP, int W_curr, int mNop)
1:  // Design time constants: pktSize (W_A), invSize (⌈T_A × W_imax/W_A⌉)
2:  // Internal(initial) state: numCachePkts(0), numFIFOPkts(0), W_consumed(W_A)
3:  if (numFIFOPkts + numCachePkts < invSize) then                  // launch fetches
4:    Request I-cache fetch ;
5:    numCachePkts++ ;
6:  endif
7:  if (cachePktRdy) then                                  // packets are ready T_A cycles later
8:    numCachePkts– ;
9:    if (W_consumed ≥ W_A and numFIFOPkts == 0) then          // load packet into ODR
10:     Load cachePkt into ODR ;
11:     W_consumed = 0 ;
12:   else                                                  // otherwise save packet in FIFO
13:     Load cachePkt into FIFO ;
14:     numFIFOPkts++ ;
15:   endif
16: endif
17: if (W_consumed ≥ W_A and numFIFOPkts > 0) then           // draw packet from FIFO
18:   Load FIFOPkt into ODR ;
19:   W_consumed -= W_A ;
20:   Advance FIFO ;
21:   numFIFOPkts– ;
22: endif
23: if (flushPipe) then                                     // branch or interrupt processing
24:   Flush I-cache and FIFO ;
25:   numCachePkts = 0 ;
26:   numFIFOPkts = 0 ;
27:   W_consumed = W_A ;
28: else                                                    // sequential instruction processing
29:   if (mNop > 0) then                                    // introduce no-op bubbles
30:     Bubble pipeline by mNop cycles ;
31:   endif
32:   if (EOP) then                                         // skip to the end-of-packet
33:     Shift IR by W_IR + W_A ;
34:     W_consumed = W_A ;
35:   else                                                  // shift to next instruction
36:     Shift IR by W_curr ;
37:     W_consumed += W_curr ;
38:   endif
39: endif
```

Figure 15: Pseudo-Code for instruction unit control logic.

as VHDL or Verilog and synthesized into a finite-state machine (FSM) using standard synthesis tools producing a concrete implementation in terms of gates or PLA logic along with control registers to keep track of the sequential state.

6.3.5 Instruction decode logic

The above hardware, when not stalled, guarantees that a valid instruction would be left-aligned in the instruction register at each cycle. This instruction needs to be decoded to determine the values of the various datapath control signals such as functional unit opcodes and register file read addresses, as well as feedback to the instruction unit to find the start of the next instruction.

In our current scheme, we use multiple, parallel decoders implemented as PLAs to control each functional unit and provide feedback to the instruction unit. Each PLA description is a sequence of control logic tables directly derived from the instruction template and field specifications as determined by the instruction format design. The instruction unit PLA decodes the template identifier and the consume-to-end-of-packet bit to determine the width of the current instruction and hence the number of quanta by which to shift the instruction register. It also decodes the number of multi-noops following the current instruction. The functional unit PLAs decode the operation fields within the current template that belong to each unit. If a template does not contain any operation for a functional unit, its PLA implicitly decodes a no-op.

7 VHDL and other outputs

7.1 Structural VHDL

The final step of the design process is to produce a structural description of the hardware at the RTL-level in a standard hardware description language such as VHDL. This description can be linked with the respective HDL component libraries pointed to by the macrocell database and processed further for hardware synthesis and simulation. The PICO system also optionally generates

a list of empty stubs for the library components used in the structural design. This allows for visual inspection of the design in a schematic viewer without linking to a real macrocell library.

7.2 Synthesis Feedback

The PICO system also produces a synthesis report consisting of a breakdown of the area estimate for the processor by components and feedback information for the architectural Spacewalker with regard to the utilization of various hardware resources and instruction format bits by the various components of the archspec.

The VLIW synthesis feedback information is output into a single file, organized into various sections according to the synthesis phase the information belongs to. Below, we describe the purpose and the textual format of the output in each section. The order of appearance of the sections within the output file can be arbitrary, but the order of fields within a section and their format (except whitespace) is fixed. The synthesis feedback output for the machine specified in Appendix A is shown in Appendix B.

The various information sections are defined as follows:

INFO-OUTPUT ::= [VLIW-COST-SECTION]
[VLIW-REGPORT-SECTION]
[VLIW-MACROCELL-SECTION]
[INST-TEMPLATE-SECTION]

7.2.1 VLIW Cost

In the VLIW cost section we report a grand total for the whole VLIW processor (excluding caches and local memory) followed by a cost breakup for interconnect, register files, and functional unit macrocells. The interconnect consists of multiplexors, de-multiplexors, tri-states, decode and control PLAs, and miscellaneous control logic. We divide the cost of interconnect into three subheadings: interconnect cost for datapath, interconnect cost for controlpath and the sequencing and

decode control logic cost. In case of register files and macrocells, individual AIR component cost is listed. All area numbers are in mm^2. The names of register files and macrocells correspond to the same names appearing in the register file porting section and the macrocell coverage section.

VLIW-COST-SECTION ::= `vliw_cost{`
`total_cost` = *float*
`dpath_ic` = *float*
`cpath_ic` = *float*
`control` = *float*
rf-name = *float*
...
mcell-name = *float*
...}

7.2.2 Register File Porting

In the register file porting section we report the number of input and output ports required for each register file during VLIW synthesis and the names of the operation groups that request each port.

VLIW-REGPORT-SECTION ::= `vliw_regports{`
`num_regfiles` = *int*
rf-name = RF-INFO
...}

RF-INFO ::= {`num_input_ports` = *int*
`num_output_ports` = *int*
`input_req` = PORT-INFO
`output_req` = PORT-INFO}

PORT-INFO ::= {OPGROUP-INFO...}

OPGROUP-INFO ::= {*opgroup-name*...}

7.2.3 Macrocell Coverage

In the VLIW macrocell coverage section we report the operation groups covered by each macrocell used within the VLIW processor.

VLIW-MACROCELL-SECTION ::= `vliw_macrocells`{
`num_macrocells` = *int*
mcell-name = OPGROUP-INFO
...}

7.2.4 Instruction Templates

In the instruction format template section we report the total bit widths of all the instruction templates used within the VLIW processor. We also provide a break down of bit width for each template according to its constituent operation groups.

INST-TEMPLATE-SECTION ::= `vliw_inst_templates`{
`max_inst_size` = *int*
`min_inst_size` = *int*
`quantum_size` = *int*
`num_templates` = *int*
template-name = TEMPLATE-INFO
...}

TEMPLATE-INFO ::= {`bit_width` = *int*
`num_par_sets` = *int*
par-set-name = PAR-SET-INFO

$$
\begin{aligned}
& \ldots\} \\
\text{PAR-SET-INFO} \quad ::= \quad & \{\texttt{num\_opgroups} = \textit{int} \\
& \textit{opgroup-name} = \textit{int} \\
& \ldots\}
\end{aligned}
$$

7.3 Architecture Manual

Finally, the system automatically generates an architecture manual for the target processor complete with a reference manual for its concrete instruction set architecture. The architecture manual documents all the architectural features of the target processor including whether it is designed for predication and/or speculation, the type and number of physical registers, schematic datapath showing the register files and the functional units, and co-processor architecture (if any). The concrete ISA description includes the operation repertoire, operation formats and the detailed structure of all the instruction templates. It also optionally contains information on the application for which the target processor was customized and the performance of the processor for that application. A sample architecture manual appears in Appendix C for the specification given in Appendix A customized to the "jpeg" application.

8 Related work

The related work focuses on either the datapath design using a Spacewalker or the processor design from a concrete instruction set architecture (ISA), which contains the same type of detailed information as an architecture manual. The MOVE project at Delft University falls in the first category. The emphasis is on the design of processor datapaths for *Transport Triggered Architectures* [5]. The datapath template used by the Spacewalker consists of a set of functional units, a set of register files and a set of buses connecting the functional units and the register files. The Spacewalker works with a structural representation of the datapath, adding and deleting register files, functional units, buses and interconnection points to come up with a set of Pareto-optimal datapaths. The

philosophy for designing the control is simple, similar to horizontal microprogramming, *i.e.*, each control point is controlled by a separate field in the instruction word. Thus, the work doesn't address the design of sophisticated instruction formats optimized for code size and the corresponding instruction fetch and decode logic within the processor.

The work by Fisher *et al*. at HP Labs [7] is similar in nature and focuses on the design of processor datapath for a clustered VLIW architecture, similar to the Multiflow Trace architecture [4]. The datapath template used in the design process is highly stylized; for example, it doesn't permit register port sharing and assumes that each functional unit has dedicated ports to register files. A major component of their work is directed towards understanding how a processor designed for an application or a group of applications performs on other applications in the same domain, *e.g.*, image processing.

The approach presented by Hadjiyiannis *et al*. [11] uses Instruction Set Description Language (ISDL) to specify a concrete ISA, which includes not only the desired operations but also the detailed instruction format and the constraints on instruction issue. The specification is then used to design the processor hardware in the form of synthesizable Verilog and to retarget various tools, such as a code-generator, assembler and simulator, needed to evaluate the performance. ISDL is a very general language capable of specifying many different types of architectures. Since an ISDL specification is at the level of a concrete ISA, the designer (either a person or a Spacewalker) has to do most of the work (*e.g.*, instruction format design) that our system does automatically. In our opinion, this makes it less suitable as a tool for comprehensive design space exploration and more suitable for a design process that requires only small incremental changes to an existing specification.

9 Conclusions

PICO-VLIW is a synthesis system for automatically designing the architecture and the micro-architecture of VLIW and EPIC processors. It was designed with automatic design space exploration in mind; the VLIW synthesis in PICO-VLIW is driven by an abstract rather than a con-

crete ISA specification, since it is easier for the Spacewalker (or, for that matter, a human being) to specify the former. Starting from an abstract specification of the instruction-set architecture, PICO-VLIW automatically generates,

1. the concrete ISA for the processor,
2. the detailed micro-architecture output in the form of RTL-level structural VHDL,
3. a machine description for use by our retargetable compiler, assembler and simulator, and,
4. an architecture manual and detailed statistics for the Spacewalker.

PICO-VLIW designs sophisticated VLIW and EPIC processors with non-trivial requirements and constraints upon their ILP, shared register ports, variable-length multi-template instruction formats that minimize code size, an instruction prefetch unit that covers the instruction cache latency, and instruction alignment and distribution networks to deal with the variable length instructions. In the course of a typical exploration, VLIW synthesis is invoked tens or hundreds of time and each resulting design is evaluated in the context of the given application.

PICO-VLIW has been operational as a research prototype since late 1997. At this point, we have exercised it with several applications ranging from loop-intensive algorithms for signal and image processing to less structured ones such as compress and ghostscript.

References

[1] Shail Aditya, B. Ramakrishna Rau, and Richard C. Johnson. Automatic design of VLIW and EPIC instruction formats. Technical Report HPL-1999-94, Hewlett-Packard Laboratories, 1999.

[2] Arvind and Xiaowei Shen. Design and verification of processors using term rewriting systems. *IEEE Micro*, May/June 1999. Special Issue on Modeling and Validation of Microprocessors.

[3] G. J. Chaitin. Register allocation and spilling via graph coloring. In *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, pages 98–105, Boston, Massachusetts, June 23–25, 1982.

[4] R. P. Colwell, R. P. Nix, J. J. O'Donnell, D. P. Papworth, and P. K. Rodman. A VLIW architecture for a trace scheduling compiler. In *Second Intl. Conf. on Architectural Support*

for Programming Languages and Operating Systems (ASPLOS II), pages 180–192, Palo Alto, CA, October 1987.

[5] Henk Corporaal and Reinoud Lamberts. TTA Processor Synthesis. In *First Annual Conf. of ASCI*, Heijen, The Netherlands, May 1995.

[6] A. Fauth. Beyond tool specific machine descriptions. In P. Marwedel and G. Goossens, editors, *Code Generation for Embedded Processors*, pages 138–152. Kluwer Academic Publishers, 1995.

[7] Joseph A. Fisher, Paolo Faraboschi, and Giuseppe Desoli. Custom-Fit Processors: Letting Applications Define Architectures. In *29th Annual IEEE/ACM Symposium on Microarchitecture (MICRO-29)*, pages 324–335, Paris, December 1996.

[8] Michael R. Garey and David. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness*. W. H. Freeman & Company, 1979.

[9] John C. Gyllenhaal, Wen-mei W. Hwu, and B. Ramakrishna Rau. HMDES version 2.0 specification. Technical Report IMPACT-96-3, University of Illinois at Urbana-Champaign, 1996.

[10] G. Hadjiyiannis, S. Hanono, and S. Devadas. ISDL: An instruction set description language for retargetability. In *ACM/IEEE Design Automation Conference*, 1997.

[11] G. Hadjiyiannis, P. Russo, and S. Devadas. A Methodology for Accurate Performance Evaluation in Architecture Exploration. In *Design Automation Conference*, New Orleans, LA, June 1999.

[12] Ellis Horowitz and Sartaj Sahni. *Fundamentals of Computer Algorithms*. Computer Science Press, Rockville, MD, 1984.

[13] Vinod Kathail, Mike Schlansker, and B. Ramakrishna Rau. HPL PlayDoh architecture specification: Version 1.0. Technical Report HPL-93-80, Hewlett-Packard Laboratories, February 1994.

[14] D. Lanneer, J. Van Praet, A. K. Kifli, K. Schoofs, W. Geurts, F. Thoen, and G. Goossens. CHESS: retargetable code generation for embedded DSP processors. In P. Marwedel and G. Goossens, editors, *Code Generation for Embedded Processors*, pages 85–102. Kluwer Academic Publishers, 1995.

[15] R. Leupers and P. Marwedel. Retargetable generation of code selectors from HDL processor models. In *Proceedings of European Design and Test Conference*, pages 140–144, March 1997.

[16] B. Ramakrishna Rau, Vinod Kathail, and Shail Aditya. Machine-description driven compilers for EPIC and VLIW processors. Technical Report HPL-98-40, Hewlett-Packard Laboratories, September 1998. To appear in Journal of *Design Automation for Embedded Systems*, 1999.

A Example Architecture Specification

```
$include "VLIW_family.hmdes2"

SECTION Register
{
  gpr0(); gpr1(); gpr2(); gpr3(); gpr4(); gpr5(); gpr6(); gpr7(); gpr8();
  gpr9(); gpr10(); gpr11(); gpr12(); gpr13(); gpr14(); gpr15(); gpr16();
  gpr17(); gpr18(); gpr19(); gpr20(); gpr21(); gpr22(); gpr23(); gpr24();
  gpr25(); gpr26(); gpr27(); gpr28(); gpr29(); gpr30(); gpr31(); gpr32();
  gpr33(); gpr34(); gpr35(); gpr36(); gpr37(); gpr38(); gpr39(); gpr40();
  gpr41(); gpr42(); gpr43(); gpr44(); gpr45(); gpr46(); gpr47(); gpr48();
  gpr49(); gpr50(); gpr51(); gpr52(); gpr53(); gpr54(); gpr55(); gpr56();
  gpr57(); gpr58(); gpr59(); gpr60(); gpr61(); gpr62(); gpr63(); fpr0();
  fpr1(); fpr2(); fpr3(); fpr4(); fpr5(); fpr6(); fpr7(); fpr8(); fpr9();
  fpr10(); fpr11(); fpr12(); fpr13(); fpr14(); fpr15(); fpr16(); fpr17();
  fpr18(); fpr19(); fpr20(); fpr21(); fpr22(); fpr23(); fpr24(); fpr25();
  fpr26(); fpr27(); fpr28(); fpr29(); fpr30(); fpr31(); fpr32(); fpr33();
  fpr34(); fpr35(); fpr36(); fpr37(); fpr38(); fpr39(); fpr40(); fpr41();
  fpr42(); fpr43(); fpr44(); fpr45(); fpr46(); fpr47(); fpr48(); fpr49();
  fpr50(); fpr51(); fpr52(); fpr53(); fpr54(); fpr55(); fpr56(); fpr57();
  fpr58(); fpr59(); fpr60(); fpr61(); fpr62(); fpr63(); pr0(); pr1(); pr2();
  pr3(); pr4(); pr5(); pr6(); pr7(); pr8(); pr9(); pr10(); pr11(); pr12();
  pr13(); pr14(); pr15(); pr16(); pr17(); pr18(); pr19(); pr20(); pr21();
  pr22(); pr23(); pr24(); pr25(); pr26(); pr27(); pr28(); pr29(); pr30();
  pr31(); pr32(); pr33(); pr34(); pr35(); pr36(); pr37(); pr38(); pr39();
  pr40(); pr41(); pr42(); pr43(); pr44(); pr45(); pr46(); pr47(); pr48();
  pr49(); pr50(); pr51(); pr52(); pr53(); pr54(); pr55(); pr56(); pr57();
  pr58(); pr59(); pr60(); pr61(); pr62(); pr63(); cr0(); cr1(); cr2(); cr3();
  cr4(); cr5(); cr6(); cr7(); btr0(); btr1(); btr2(); btr3(); btr4(); btr5();
  btr6(); btr7(); btr8(); btr9(); btr10(); btr11(); btr12(); btr13(); btr14();
  btr15();
}

SECTION Register_File
{
  gpr           (width(32)
                 static(gpr0 gpr1 gpr2 gpr3 gpr4 gpr5 gpr6 gpr7 gpr8 gpr9
                        gpr10 gpr11 gpr12 gpr13 gpr14 gpr15 gpr16 gpr17
                        gpr18 gpr19 gpr20 gpr21 gpr22 gpr23 gpr24 gpr25
                        gpr26 gpr27 gpr28 gpr29 gpr30 gpr31 gpr32 gpr33
                        gpr34 gpr35 gpr36 gpr37 gpr38 gpr39 gpr40 gpr41
                        gpr42 gpr43 gpr44 gpr45 gpr46 gpr47 gpr48 gpr49
                        gpr50 gpr51 gpr52 gpr53 gpr54 gpr55 gpr56 gpr57
                        gpr58 gpr59 gpr60 gpr61 gpr62 gpr63)
                 speculative(0)
                 virtual("I"));
  fpr           (width(64)
                 static(fpr0 fpr1 fpr2 fpr3 fpr4 fpr5 fpr6 fpr7 fpr8 fpr9
                        fpr10 fpr11 fpr12 fpr13 fpr14 fpr15 fpr16 fpr17
                        fpr18 fpr19 fpr20 fpr21 fpr22 fpr23 fpr24 fpr25
                        fpr26 fpr27 fpr28 fpr29 fpr30 fpr31 fpr32 fpr33
                        fpr34 fpr35 fpr36 fpr37 fpr38 fpr39 fpr40 fpr41
                        fpr42 fpr43 fpr44 fpr45 fpr46 fpr47 fpr48 fpr49
                        fpr50 fpr51 fpr52 fpr53 fpr54 fpr55 fpr56 fpr57
                        fpr58 fpr59 fpr60 fpr61 fpr62 fpr63)
                 speculative(0)
                 virtual("F"));
  pr            (width(1)
                 static(pr0 pr1 pr2 pr3 pr4 pr5 pr6 pr7 pr8 pr9 pr10 pr11
                        pr12 pr13 pr14 pr15 pr16 pr17 pr18 pr19 pr20 pr21
                        pr22 pr23 pr24 pr25 pr26 pr27 pr28 pr29 pr30 pr31
                        pr32 pr33 pr34 pr35 pr36 pr37 pr38 pr39 pr40 pr41
```

```
                        pr42 pr43 pr44 pr45 pr46 pr47 pr48 pr49 pr50 pr51
                        pr52 pr53 pr54 pr55 pr56 pr57 pr58 pr59 pr60 pr61
                        pr62 pr63)
                 speculative(0)
                 virtual("P"));
  cr             (width(32)
                 static(cr0 cr1 cr2 cr3 cr4 cr5 cr6 cr7)
                 speculative(0)
                 virtual("C"));
  btr            (width(64)
                 static(btr0 btr1 btr2 btr3 btr4 btr5 btr6 btr7 btr8 btr9
                        btr10 btr11 btr12 btr13 btr14 btr15)
                 speculative(0)
                 virtual("B"));
  s              (width(6)
                 virtual("L")
                 intrange(-32 31));
  m              (width(9)
                 virtual("L")
                 intrange(-256 255));
  n              (width(32)
                 virtual("L")
                 intrange(-2147483648 2147483647));
  o              (width(17)
                 virtual("L")
                 intrange(-65536 65535));
  U              (width(0)
                 virtual("U"));
}

SECTION VLIW_RF_Map
{
  RFMap_i_f_p_c_a_s
                (vrf("I" "F" "P" "C" "B" "L" "U")
                 prf("gpr" "fpr" "pr" "cr" "btr" "s" "U"));
  RFMap_i_f_p_c_a_m
                (vrf("I" "F" "P" "C" "B" "L" "U")
                 prf("gpr" "fpr" "pr" "cr" "btr" "m" "U"));
  RFMap_i_f_p_c_a_n
                (vrf("I" "F" "P" "C" "B" "L" "U")
                 prf("gpr" "fpr" "pr" "cr" "btr" "n" "U"));
  RFMap_i_f_p_c_a_o
                (vrf("I" "F" "P" "C" "B" "L" "U")
                 prf("gpr" "fpr" "pr" "cr" "btr" "o" "U"));
}

SECTION VLIW_Operation_Group
{
  VOG_o79c4i0   (opset("EOS_brcond_branch")
                 src_rfmap(RFMap_i_f_p_c_a_s)
                 dest_rfmap(RFMap_i_f_p_c_a_s)
                 latency(OL_branch)
                 resv(RT_VOG_o79c4i0)
                 alt_priority(0));
  VOG_o80c4i0   (opset("EOS_brlink_branch")
                 src_rfmap(RFMap_i_f_p_c_a_s)
                 dest_rfmap(RFMap_i_f_p_c_a_s)
                 latency(OL_branch)
                 resv(RT_VOG_o80c4i0)
                 alt_priority(0));
  VOG_o78c4i0   (opset("EOS_brucond_branch")
                 src_rfmap(RFMap_i_f_p_c_a_s)
                 dest_rfmap(RFMap_i_f_p_c_a_s)
                 latency(OL_branch)
```

```
                resv(RT_VOG_o78c4i0)
                alt_priority(0));
VOG_o22c1i0    (opset("EOS_btr_literal_moves")
                src_rfmap(RFMap_i_f_p_c_a_n)
                dest_rfmap(RFMap_i_f_p_c_a_n)
                latency(OL_int)
                resv(RT_VOG_o22c1i0)
                alt_priority(1));
VOG_o38c2i0    (opset("EOS_convff_float")
                src_rfmap(RFMap_i_f_p_c_a_s)
                dest_rfmap(RFMap_i_f_p_c_a_s)
                latency(OL_float)
                resv(RT_VOG_o38c2i0)
                alt_priority(0));
VOG_o37c2i0    (opset("EOS_convfi_D_float")
                src_rfmap(RFMap_i_f_p_c_a_s)
                dest_rfmap(RFMap_i_f_p_c_a_s)
                latency(OL_float)
                resv(RT_VOG_o37c2i0)
                alt_priority(0));
VOG_o34c2i0    (opset("EOS_convif_S_float")
                src_rfmap(RFMap_i_f_p_c_a_s)
                dest_rfmap(RFMap_i_f_p_c_a_s)
                latency(OL_float)
                resv(RT_VOG_o34c2i0)
                alt_priority(0));
VOG_o31c2i0    (opset("EOS_floatarith2_D_float")
                src_rfmap(RFMap_i_f_p_c_a_s)
                dest_rfmap(RFMap_i_f_p_c_a_s)
                latency(OL_float)
                resv(RT_VOG_o31c2i0)
                alt_priority(0));
VOG_o27c2i0    (opset("EOS_floatarith2_S_floatdiv")
                src_rfmap(RFMap_i_f_p_c_a_s)
                dest_rfmap(RFMap_i_f_p_c_a_s)
                latency(OL_floatdiv)
                resv(RT_VOG_o27c2i0)
                alt_priority(0));
VOG_o28c2i0    (opset("EOS_floatarith2_S_floatmpy")
                src_rfmap(RFMap_i_f_p_c_a_s)
                dest_rfmap(RFMap_i_f_p_c_a_s)
                latency(OL_floatmpy)
                resv(RT_VOG_o28c2i0)
                alt_priority(0));
VOG_o56c3i0    (opset("EOS_floatload_load1")
                src_rfmap(RFMap_i_f_p_c_a_s)
                dest_rfmap(RFMap_i_f_p_c_a_s)
                latency(OL_load1)
                resv(RT_VOG_o56c3i0)
                alt_priority(0));
VOG_o62c3i0    (opset("EOS_floatstore_store")
                src_rfmap(RFMap_i_f_p_c_a_s)
                dest_rfmap(RFMap_i_f_p_c_a_s)
                latency(OL_store)
                resv(RT_VOG_o62c3i0)
                alt_priority(0));
VOG_o1c1i0     (opset("EOS_intarith2_int")
                src_rfmap(RFMap_i_f_p_c_a_s)
                dest_rfmap(RFMap_i_f_p_c_a_s)
                latency(OL_int)
                resv(RT_VOG_o1c1i0)
                alt_priority(1));
VOG_o1c1i1     (opset("EOS_intarith2_int")
                src_rfmap(RFMap_i_f_p_c_a_s)
```

```
                dest_rfmap(RFMap_i_f_p_c_a_s)
                latency(OL_int)
                resv(RT_VOG_o1c1i1)
                alt_priority(1));
VOG_o3c1i0     (opset("EOS_intarith2_intdiv")
                src_rfmap(RFMap_i_f_p_c_a_s)
                dest_rfmap(RFMap_i_f_p_c_a_s)
                latency(OL_intdiv)
                resv(RT_VOG_o3c1i0)
                alt_priority(1));
VOG_o4c1i0     (opset("EOS_intarith2_intmpy")
                src_rfmap(RFMap_i_f_p_c_a_s)
                dest_rfmap(RFMap_i_f_p_c_a_s)
                latency(OL_intmpy)
                resv(RT_VOG_o4c1i0)
                alt_priority(1));
VOG_o2c1i0     (opset("EOS_intarith2_intshift")
                src_rfmap(RFMap_i_f_p_c_a_s)
                dest_rfmap(RFMap_i_f_p_c_a_s)
                latency(OL_int)
                resv(RT_VOG_o2c1i0)
                alt_priority(1));
VOG_o2c1i1     (opset("EOS_intarith2_intshift")
                src_rfmap(RFMap_i_f_p_c_a_s)
                dest_rfmap(RFMap_i_f_p_c_a_s)
                latency(OL_int)
                resv(RT_VOG_o2c1i1)
                alt_priority(1));
VOG_o18c1i0    (opset("EOS_intcmpp_uncond")
                src_rfmap(RFMap_i_f_p_c_a_s)
                dest_rfmap(RFMap_i_f_p_c_a_s)
                latency(OL_intcmpp)
                resv(RT_VOG_o18c1i0)
                alt_priority(1));
VOG_o52c3i0    (opset("EOS_intload_load1")
                src_rfmap(RFMap_i_f_p_c_a_s)
                dest_rfmap(RFMap_i_f_p_c_a_s)
                latency(OL_load1)
                resv(RT_VOG_o52c3i0)
                alt_priority(0));
VOG_o5c1i0     (opset("EOS_intsext_int")
                src_rfmap(RFMap_i_f_p_c_a_s)
                dest_rfmap(RFMap_i_f_p_c_a_s)
                latency(OL_int)
                resv(RT_VOG_o5c1i0)
                alt_priority(1));
VOG_o60c3i0    (opset("EOS_intstore_store")
                src_rfmap(RFMap_i_f_p_c_a_s)
                dest_rfmap(RFMap_i_f_p_c_a_s)
                latency(OL_store)
                resv(RT_VOG_o60c3i0)
                alt_priority(0));
VOG_o6c1i0     (opset("EOS_moveii_int")
                src_rfmap(RFMap_i_f_p_c_a_s)
                dest_rfmap(RFMap_i_f_p_c_a_s)
                latency(OL_int)
                resv(RT_VOG_o6c1i0)
                alt_priority(1));
VOG_o21c1i0    (opset("EOS_pbr_int")
                src_rfmap(RFMap_i_f_p_c_a_s)
                dest_rfmap(RFMap_i_f_p_c_a_s)
                latency(OL_int)
                resv(RT_VOG_o21c1i0)
                alt_priority(1));
```

```
}

SECTION VLIW_Exclusion_Group
{
  VEG_RES_FU_4_0(opgroups(VOG_o79c4i0 VOG_o80c4i0 VOG_o78c4i0));
  VEG_RES_FU_1_0(opgroups(VOG_o22c1i0 VOG_o1c1i0 VOG_o3c1i0 VOG_o4c1i0
                          VOG_o2c1i0 VOG_o18c1i0 VOG_o5c1i0 VOG_o6c1i0
                          VOG_o21c1i0));
  VEG_RES_FU_2_0(opgroups(VOG_o38c2i0 VOG_o37c2i0 VOG_o34c2i0 VOG_o31c2i0
                          VOG_o27c2i0 VOG_o28c2i0));
  VEG_RES_FU_3_0(opgroups(VOG_o56c3i0 VOG_o62c3i0 VOG_o52c3i0 VOG_o60c3i0));
  VEG_RES_FU_1_1(opgroups(VOG_o1c1i1 VOG_o2c1i1));
}

SECTION Architecture_Flag
  OPTIONAL intvalue(INT);
  OPTIONAL doublevalue(DOUBLE);
  OPTIONAL stringvalue(STRING);
{
  predication_hw(intvalue(0));
  speculation_hw(intvalue(0));
  systolic_hw   (intvalue(0));
  technology_scale(doublevalue(0.25));
}
```

B Synthesis Feedback Report

```
vliw_cost { // area in mm^2
  total_cost      =   25.515
  dpath_ic        =    1.496
  cpath_ic        =    1.020
  control         =    0.754
  gpr             =    2.432
  fpr             =    1.050
  pr              =    0.023
  cr              =    0.158
  btr             =    0.340
  PD_f_bas_sd_0   =    2.857
  PD_m_ifsd_1     =    2.857
  PD_i_bas_2      =    0.714
  PD_branch_3     =    0.714
  PD_pbr_4        =    0.357
  PD_f_div_sd_5   =    5.714
  PD_i_div_6      =    2.143
  PD_i_mpyadd_7   =    1.429
  PD_i_bas_8      =    0.714
  PD_i_shift_9    =    0.357
  PD_i_shift_10   =    0.357
  misc.           =    0.028
}

vliw_regports {
  num_regfiles = 5
  gpr             = {
    num_input_ports = 4
    num_output_ports = 7
    input_req = {
      { VOG_o37c2i0 }
      { VOG_o1c1i0 VOG_o3c1i0 VOG_o4c1i0 VOG_o2c1i0 VOG_o5c1i0 VOG_o6c1i0 }
```

```
    { VOG_o1c1i1 VOG_o2c1i1 }
    { VOG_o52c3i0 }
  }
  output_req = {
    { VOG_o34c2i0 }
    { VOG_o56c3i0 VOG_o62c3i0 VOG_o52c3i0 VOG_o60c3i0 }
    { VOG_o1c1i0 VOG_o3c1i0 VOG_o4c1i0 VOG_o2c1i0 VOG_o18c1i0 VOG_o5c1i0 VOG_o6c1i0 VOG_o21c1i0 }
    { VOG_o1c1i0 VOG_o3c1i0 VOG_o4c1i0 VOG_o2c1i0 VOG_o18c1i0 }
    { VOG_o1c1i1 VOG_o2c1i1 }
    { VOG_o1c1i1 VOG_o2c1i1 }
    { VOG_o60c3i0 }
  }
}
fpr             = {
  num_input_ports = 2
  num_output_ports = 3
  input_req = {
    { VOG_o38c2i0 VOG_o34c2i0 VOG_o31c2i0 VOG_o27c2i0 VOG_o28c2i0 }
    { VOG_o56c3i0 }
  }
  output_req = {
    { VOG_o38c2i0 VOG_o37c2i0 VOG_o31c2i0 VOG_o27c2i0 VOG_o28c2i0 }
    { VOG_o31c2i0 VOG_o27c2i0 VOG_o28c2i0 }
    { VOG_o62c3i0 }
  }
}
pr              = {
  num_input_ports = 1
  num_output_ports = 1
  input_req = {
    { VOG_o18c1i0 }
  }
  output_req = {
    { VOG_o79c4i0 }
  }
}
cr              = {
  num_input_ports = 3
  num_output_ports = 5
  input_req = {
    { VOG_o1c1i0 VOG_o3c1i0 VOG_o4c1i0 VOG_o2c1i0 VOG_o6c1i0 }
    { VOG_o1c1i1 VOG_o2c1i1 }
    { VOG_o52c3i0 }
  }
  output_req = {
    { VOG_o1c1i0 VOG_o3c1i0 VOG_o4c1i0 VOG_o2c1i0 VOG_o6c1i0 }
    { VOG_o1c1i0 VOG_o3c1i0 VOG_o4c1i0 VOG_o2c1i0 }
    { VOG_o1c1i1 VOG_o2c1i1 }
    { VOG_o1c1i1 VOG_o2c1i1 }
    { VOG_o60c3i0 }
  }
}
btr             = {
  num_input_ports = 3
  num_output_ports = 3
  input_req = {
    { VOG_o80c4i0 }
    { VOG_o22c1i0 VOG_o6c1i0 VOG_o21c1i0 }
    { VOG_o52c3i0 }
  }
  output_req = {
    { VOG_o79c4i0 VOG_o80c4i0 VOG_o78c4i0 }
    { VOG_o60c3i0 }
    { VOG_o6c1i0 VOG_o21c1i0 }
```

```
    }
  }
}

vliw_macrocells {
  num_macrocells = 11
  PD_f_bas_sd_0   = { VOG_o38c2i0 VOG_o37c2i0 VOG_o34c2i0 VOG_o31c2i0 VOG_o28c2i0 }
  PD_m_ifsd_1     = { VOG_o56c3i0 VOG_o62c3i0 VOG_o52c3i0 VOG_o60c3i0 }
  PD_i_bas_2      = { VOG_o1c1i0 VOG_o18c1i0 VOG_o5c1i0 VOG_o6c1i0 }
  PD_branch_3     = { VOG_o79c4i0 VOG_o80c4i0 VOG_o78c4i0 }
  PD_pbr_4        = { VOG_o22c1i0 VOG_o21c1i0 }
  PD_f_div_sd_5   = { VOG_o27c2i0 }
  PD_i_div_6      = { VOG_o3c1i0 }
  PD_i_mpyadd_7   = { VOG_o4c1i0 }
  PD_i_bas_8      = { VOG_o1c1i1 }
  PD_i_shift_9    = { VOG_o2c1i0 }
  PD_i_shift_10   = { VOG_o2c1i1 }
}

vliw_inst_templates {
  max_inst_size = 160
  min_inst_size = 32
  quantum_size = 32
  packet_width = 256
  num_templates = 8
  template_0 = {
    bit_width = 160
    num_par_sets = 5
    par_set_0 = {
      num_opgroups = 3
      VOG_o79c4i0(EOS_brcond_branch)           = 11
      VOG_o80c4i0(EOS_brlink_branch)           = 8
      VOG_o78c4i0(EOS_brucond_branch)          = 5
    }
    par_set_1 = {
      num_opgroups = 9
      VOG_o22c1i0(EOS_btr_literal_moves)       = 38
      VOG_o1c1i0(EOS_intarith2_int)            = 28
      VOG_o3c1i0(EOS_intarith2_intdiv)         = 25
      VOG_o4c1i0(EOS_intarith2_intmpy)         = 24
      VOG_o2c1i0(EOS_intarith2_intshift)       = 25
      VOG_o18c1i0(EOS_intcmpp_uncond)          = 26
      VOG_o5c1i0(EOS_intsext_int)              = 13
      VOG_o6c1i0(EOS_moveii_int)               = 16
      VOG_o21c1i0(EOS_pbr_int)                 = 19
    }
    par_set_2 = {
      num_opgroups = 6
      VOG_o38c2i0(EOS_convff_float)            = 13
      VOG_o37c2i0(EOS_convfi_D_float)          = 12
      VOG_o34c2i0(EOS_convif_S_float)          = 12
      VOG_o31c2i0(EOS_floatarith2_D_float)     = 20
      VOG_o27c2i0(EOS_floatarith2_S_floatdiv)  = 18
      VOG_o28c2i0(EOS_floatarith2_S_floatmpy)  = 18
    }
    par_set_3 = {
      num_opgroups = 4
      VOG_o56c3i0(EOS_floatload_load1)         = 13
      VOG_o62c3i0(EOS_floatstore_store)        = 13
      VOG_o52c3i0(EOS_intload_load1)           = 16
      VOG_o60c3i0(EOS_intstore_store)          = 16
    }
    par_set_4 = {
      num_opgroups = 2
```

```
    VOG_o1c1i1(EOS_intarith2_int)            = 28
    VOG_o2c1i1(EOS_intarith2_intshift)       = 27
  }
}
template_1 = {
  bit_width = 64
  num_par_sets = 2
  par_set_0 = {
    num_opgroups = 1
    VOG_o1c1i0(EOS_intarith2_int)            = 28
  }
  par_set_1 = {
    num_opgroups = 1
    VOG_o52c3i0(EOS_intload_load1)           = 16
  }
}
template_2 = {
  bit_width = 64
  num_par_sets = 2
  par_set_0 = {
    num_opgroups = 1
    VOG_o1c1i0(EOS_intarith2_int)            = 28
  }
  par_set_1 = {
    num_opgroups = 1
    VOG_o60c3i0(EOS_intstore_store)          = 16
  }
}
template_3 = {
  bit_width = 64
  num_par_sets = 1
  par_set_0 = {
    num_opgroups = 1
    VOG_o1c1i0(EOS_intarith2_int)            = 28
  }
}
template_4 = {
  bit_width = 32
  num_par_sets = 1
  par_set_0 = {
    num_opgroups = 1
    VOG_o21c1i0(EOS_pbr_int)                 = 19
  }
}
template_5 = {
  bit_width = 32
  num_par_sets = 1
  par_set_0 = {
    num_opgroups = 1
    VOG_o6c1i0(EOS_moveii_int)               = 16
  }
}
template_6 = {
  bit_width = 32
  num_par_sets = 1
  par_set_0 = {
    num_opgroups = 1
    VOG_o18c1i0(EOS_intcmpp_uncond)          = 26
  }
}
template_7 = {
  bit_width = 32
  num_par_sets = 1
  par_set_0 = {
```

```
      num_opgroups = 1
      VOG_o2c1i0(EOS_intarith2_intshift)       = 25
    }
  }
}
```

C Architecture Manual

In the following pages, the architecture manual for a machine specified in Appendix A is presented. The machine is customized to the “jpeg” application. The entire manual is generated automatically.

January 25, 2000

Application Specific Architecture for the Benchmark **jpeg99**

generated by Elcor reporting facility
Hewlett-Packard Laboratories

1 Architecture Description

This section provides the description of the processor architecture and the memory hierarchy.

1.1 Processor architecture

Architectural feature checklist

Predication	not supported
Speculation	conventional support

Table above shows which of the architectural features that are available are used in this experiment. The support for "predication" implies that operations can be conditionally executed depending on the value of their boolean input operand.

The following two tables shows the register file configuration of the processor and the sizes of literal fields in instructions.

Register files

Integer register	64
Floating-point registers	64
Branch target registers	16

Literal fields

Short literal size	6 bits
Memory literal size	9 bits
Branch literal size	32 bits
Long literal size	17 bits

1.2 Memory hierarchy

Level 1 data cache configuration

Cache line size	64 bytes
Cache associativity	2 way
Cache size	8192 bytes
Number of ports	1

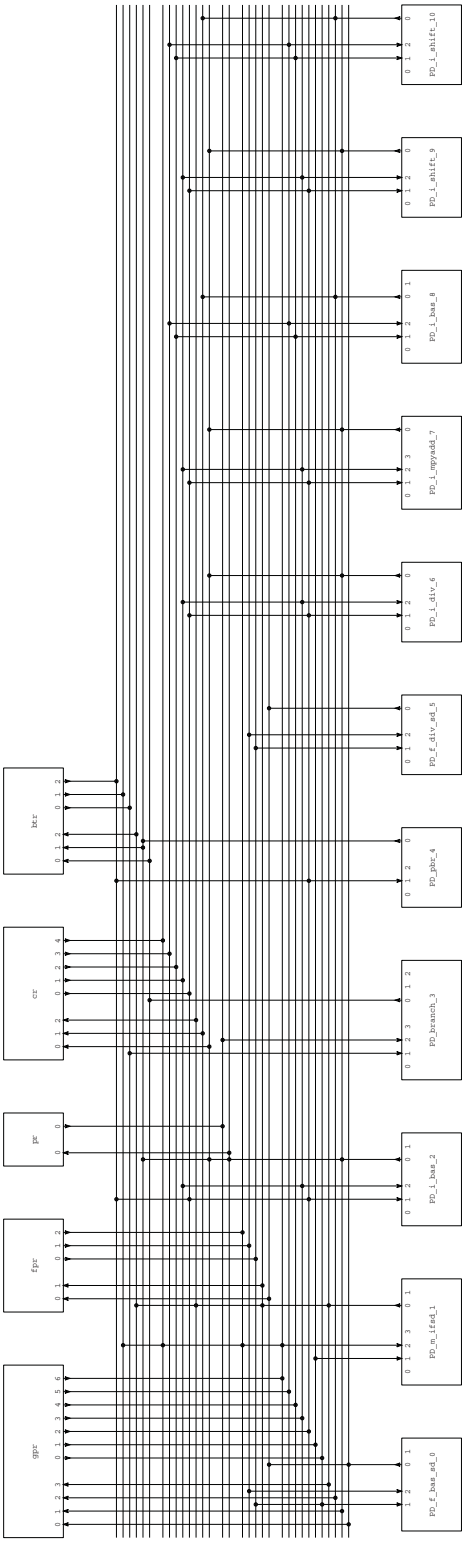

Figure 1: VLIW processor datapaths

Level 1 instruction cache configuration

Cache line size	64 bytes
Cache associativity	2 way
Cache size	8192 bytes
Number of ports	1

Level 2 cache configuration

Cache line size	64 bytes
Cache associativity	direct mapped
Cache size	65536 bytes
Number of ports	1

1.3 System area

The following table shows the chip area breakdown for the system

System area breakdown

VLIW processor area	25.515 mm^2
Cache area	28.619 mm^2
Total area	54.134 mm^2

2 Instruction Format

This section provides the instruction format for the processor.

2.1 Instruction Issue Templates

EOP< 3 > T0< 0 · · · 2 >

VOG_o79c4i0 VOG_o80c4i0 VOG_o78c4i0	VOG_o22c1i0 VOG_o1c1i0 VOG_o3c1i0 VOG_o4c1i0 VOG_o2c1i0 VOG_o18c1i0 VOG_o5c1i0 VOG_o6c1i0 VOG_o21c1i0	VOG_o38c2i0 VOG_o37c2i0 VOG_o34c2i0 VOG_o31c2i0 VOG_o27c2i0 VOG_o28c2i0	VOG_o56c3i0 VOG_o62c3i0 VOG_o52c3i0 VOG_o60c3i0	VOG_o1c1i1 VOG_o2c1i1

EOP< 3 > T1< 0 · · · 2 > VOG_o1c1i0 | VOG_o52c3i0

EOP< 3 > T2< 0 · · · 2 > VOG_o1c1i0 | VOG_o60c3i0

EOP< 3 > T3< 0 · · · 2 > VOG_o1c1i0

EOP< 3 > T4< 0 · · · 2 > VOG_o21c1i0

EOP< 3 > T5< 0 · · · 2 > VOG_o6c1i0

EOP< 3 > T6< 0 · · · 2 > VOG_o18c1i0

EOP< 3 > T7< 0 · · · 2 > VOG_o2c1i0

2.2 IO Formats for Operation Sets

Opset: EOS_brcond_branch

SRC1	SRC2
4 bits	6 bits
btr	pr

Opset: EOS_brlink_branch

SRC1	DEST1
4 bits	4 bits
btr	btr

Opset: EOS_brucond_branch

SRC1
4 bits
btr

Opset: EOS_btr_literal_moves

SRC1	DEST1
32 bits	4 bits
n	btr

Opset: EOS_intarith2_int

SRC1		SRC2		DEST1	
	6 bits		6 bits		6 bits
00	gpr	00	gpr	0	gpr
01	cr	01	cr	1	cr
10	s	10	s		

Opset: EOS_intarith2_intdiv

SRC1		SRC2		DEST1	
	6 bits		6 bits		6 bits
00	gpr	00	gpr	0	gpr
01	cr	01	cr	1	cr
10	s	10	s		

Opset: EOS_intarith2_intmpy

SRC1		SRC2		DEST1	
	6 bits		6 bits		6 bits
00	gpr	00	gpr	0	gpr
01	cr	01	cr	1	cr
10	s	10	s		

Opset: EOS_intarith2_intshift

SRC1		SRC2		DEST1	
	6 bits		6 bits		6 bits
00	gpr	00	gpr	0	gpr
01	cr	01	cr	1	cr
10	s	10	s		

Opset: EOS_intcmpp_uncond

SRC1		SRC2		DEST1
	6 bits		6 bits	6 bits
0	gpr	0	gpr	pr
1	s	1	s	

Opset: EOS_intsext_int

SRC1	DEST1
6 bits	6 bits
gpr	gpr

Opset: EOS_moveii_int

SRC1		DEST1	
	6 bits		6 bits
00	gpr	00	gpr
01	cr	01	cr
10	btr	10	btr
11	s		

Opset: EOS_pbr_int

SRC1		SRC2	DEST1
	6 bits	6 bits	4 bits
00	gpr	s	btr
01	btr		
10	s		

Opset: EOS_convff_float

SRC1	DEST1
6 bits	6 bits
fpr	fpr

Opset: EOS_convfi_D_float

SRC1	DEST1
6 bits	6 bits
fpr	gpr

Opset: EOS_convif_S_float

SRC1	DEST1
6 bits	6 bits
gpr	fpr

Opset: EOS_floatarith2_D_float

SRC1	SRC2	DEST1
6 bits	6 bits	6 bits
fpr	fpr	fpr

Opset: EOS_floatarith2_S_floatdiv

SRC1	SRC2	DEST1
6 bits	6 bits	6 bits
fpr	fpr	fpr

Opset: EOS_floatarith2_S_floatmpy

SRC1	SRC2	DEST1
6 bits	6 bits	6 bits
fpr	fpr	fpr

Opset: EOS_floatload_load1

SRC1	DEST1
6 bits	6 bits
gpr	fpr

Opset: EOS_floatstore_store

SRC1	SRC2
6 bits	6 bits
gpr	fpr

Opset: EOS_intload_load1

SRC1	DEST1	
6 bits		6 bits
gpr	00	gpr
	01	cr
	10	btr

Opset: EOS_intstore_store

SRC1	SRC2	
6 bits		6 bits
gpr	00	gpr
	01	cr
	10	btr
	11	s

2.3 Instruction Formats for Operation Groups

VOG_o79c4i0 : (Opset: EOS_brcond_branch)
IO descriptor: pr ? btr , pr :

Template	OPCODE	SRC1	SRC2
T0	$< 31 >$	$< 32, 33 >< 40, 41 >$	$< 48 \cdots 53 >$

VOG_o80c4i0 : (Opset: EOS_brlink_branch)
IO descriptor: pr ? btr : btr

Template	SRC1	DEST1
T0	$< 32, 33 >< 40, 41 >$	$< 31 >< 48 \cdots 50 >$

VOG_o78c4i0 : (Opset: EOS_brucond_branch)
IO descriptor: pr ? btr :

Template	OPCODE	SRC1
T0	$< 31 >$	$< 32, 33 >< 40, 41 >$

VOG_o22c1i0 : (Opset: EOS_btr_literal_moves)
IO descriptor: pr ? n : btr

Template	OPCODE	SRC1	DEST1
T0	< 4, 5 >	< 6 · · · 13 >< 18 · · · 22 >< 24 · · · 29 >< 58 · · · 70 >	< 14 · · · 17 >

VOG_o1c1i0 : (Opset: EOS_intarith2_int)
IO descriptor: pr ? gpr cr s , gpr cr s : gpr cr

Template	OPCODE	SRC1	SRC2	DEST1
T0	< 22 >< 24 · · · 27 >	< 28, 29 >< 6 · · · 11 >	< 58, 59 >< 4, 5 >< 12, 13 >< 20, 21 >	< 60 >< 14 · · · 19 >
T1	< 22 >< 24 · · · 27 >	< 28, 29 >< 6 · · · 11 >	< 58, 59 >< 4, 5 >< 12, 13 >< 20, 21 >	< 60 >< 14 · · · 19 >
T2	< 22 >< 24 · · · 27 >	< 28, 29 >< 6 · · · 11 >	< 58, 59 >< 4, 5 >< 12, 13 >< 20, 21 >	< 60 >< 14 · · · 19 >
T3	< 22 >< 24 · · · 27 >	< 28, 29 >< 6 · · · 11 >	< 58, 59 >< 4, 5 >< 12, 13 >< 20, 21 >	< 60 >< 14 · · · 19 >

VOG_o3c1i0 : (Opset: EOS_intarith2_intdiv)
IO descriptor: pr ? gpr cr s , gpr cr s : gpr cr

Template	OPCODE	SRC1	SRC2	DEST1
T0	< 22 >< 24 >	< 25, 26 >< 6 · · · 11 >	< 27, 28 >< 4, 5 >< 12, 13 >< 20, 21 >	< 29 >< 14 · · · 19 >

VOG_o4c1i0 : (Opset: EOS_intarith2_intmpy)
IO descriptor: pr ? gpr cr s , gpr cr s : gpr cr

Template	OPCODE	SRC1	SRC2	DEST1
T0	< 22 >	< 24, 25 >< 6 · · · 11 >	< 26, 27 >< 4, 5 >< 12, 13 >< 20, 21 >	< 28 >< 14 · · · 19 >

VOG_o2c1i0 : (Opset: EOS_intarith2_intshift)
IO descriptor: pr ? gpr cr s , gpr cr s : gpr cr

Template	OPCODE	SRC1	SRC2	DEST1
T0	< 22 >< 24 >	< 25, 26 >< 6 · · · 11 >	< 27, 28 >< 4, 5 >< 12, 13 >< 20, 21 >	< 29 >< 14 · · · 19 >
T7	< 22 >< 24 >	< 25, 26 >< 6 · · · 11 >	< 27, 28 >< 4, 5 >< 12, 13 >< 20, 21 >	< 29 >< 14 · · · 19 >

VOG_o18c1i0 : (Opset: EOS_intcmpp_uncond)
IO descriptor: pr ? gpr s , gpr s : pr , pr

Template	OPCODE	SRC1	SRC2	DEST1
T0	< 14 · · · 19 >	< 22 >< 6 · · · 11 >	< 58 >< 4, 5 >< 12, 13 >< 20, 21 >	< 24 · · · 29 >
T6	< 14 · · · 19 >	< 22 >< 6 · · · 11 >	< 58 >< 4, 5 >< 12, 13 >< 20, 21 >	< 24 · · · 29 >

VOG_o5c1i0 : (Opset: EOS_intsext_int)
IO descriptor: pr ? gpr : gpr

Template	OPCODE	SRC1	DEST1
T0	< 4 >	< 6 · · · 11 >	< 14 · · · 19 >

VOG_o6c1i0 : (Opset: EOS_moveii_int)
IO descriptor: pr ? gpr cr btr s : gpr cr btr

Template	SRC1	DEST1
T0	< 4, 5 >< 6 · · · 11 >	< 12, 13 >< 14 · · · 19 >
T5	< 4, 5 >< 6 · · · 11 >	< 12, 13 >< 14 · · · 19 >

VOG_o21c1i0 : (Opset: EOS_pbr_int)
IO descriptor: pr ? gpr btr s , s : btr

Template	OPCODE	SRC1	SRC2	DEST1
T0	< 4 >	< 5 >< 12 >< 6 · · · 11 >	< 13 >< 18 · · · 22 >	< 14 · · · 17 >
T4	< 4 >	< 5 >< 12 >< 6 · · · 11 >	< 13 >< 18 · · · 22 >	< 14 · · · 17 >

VOG_o38c2i0 : (Opset: EOS_convff_float)
IO descriptor: pr ? fpr : fpr

Template	OPCODE	SRC1	DEST1
T0	< 74 >	< 76 · · · 81 >	< 82 · · · 87 >

VOG_o37c2i0 : (Opset: EOS_convfi_D_float)
IO descriptor: pr ? fpr : gpr

Template	SRC1	DEST1
T0	< 76 · · · 81 >	< 74, 75 >< 82 · · · 85 >

VOG_o34c2i0 : (Opset: EOS_convif_S_float)
IO descriptor: pr ? gpr : fpr

Template	SRC1	DEST1
T0	< 74 · · · 79 >	< 82 · · · 87 >

VOG_o31c2i0 : (Opset: EOS_floatarith2_D_float)
IO descriptor: pr ? fpr , fpr : fpr

Template	OPCODE	SRC1	SRC2	DEST1
T0	< 74, 75 >	< 76 · · · 81 >	< 88 · · · 93 >	< 82 · · · 87 >

VOG_o27c2i0 : (Opset: EOS_floatarith2_S_floatdiv)
IO descriptor: pr ? fpr , fpr : fpr

Template	SRC1	SRC2	DEST1
T0	< 76 · · · 81 >	< 88 · · · 93 >	< 82 · · · 87 >

VOG_o28c2i0 : (Opset: EOS_floatarith2_S_floatmpy)
IO descriptor: pr ? fpr , fpr : fpr

Template	SRC1	SRC2	DEST1
T0	< 76 · · · 81 >	< 88 · · · 93 >	< 82 · · · 87 >

VOG_o56c3i0 : (Opset: EOS_floatload_load1)
IO descriptor: pr ? gpr : fpr

Template	OPCODE	SRC1	DEST1
T0	< 42 >	< 34 · · · 39 >	< 43 · · · 47 >< 97 >

VOG_o62c3i0 : (Opset: EOS_floatstore_store)
IO descriptor: pr ? gpr , fpr :

Template	OPCODE	SRC1	SRC2
T0	< 42 >	< 34 · · · 39 >	< 43 · · · 47 >< 97 >

VOG_o52c3i0 : (Opset: EOS_intload_load1)
IO descriptor: pr ? gpr : gpr cr btr

Template	OPCODE	SRC1	DEST1
T0	< 97, 98 >	< 34 · · · 39 >	< 99, 100 >< 42 · · · 47 >
T1	< 97, 98 >	< 34 · · · 39 >	< 99, 100 >< 42 · · · 47 >

VOG_o60c3i0 : (Opset: EOS_intstore_store)
IO descriptor: pr ? gpr , gpr cr btr s :

Template	OPCODE	SRC1	SRC2
T0	< 97, 98 >	< 34 · · · 39 >	< 99, 100 >< 42 · · · 47 >
T2	< 97, 98 >	< 34 · · · 39 >	< 99, 100 >< 42 · · · 47 >

VOG_o1c1i1 : (Opset: EOS_intarith2_int)
IO descriptor: pr ? gpr cr s , gpr cr s : gpr cr

Template	OPCODE	SRC1	SRC2	DEST1
T0	< 103 · · · 107 >	< 108, 109 >< 110 · · · 115 >	< 116, 117 >< 118 · · · 123 >	< 124 >< 125 · · · 130 >

VOG_o2c1i1 : (Opset: EOS_intarith2_intshift)
IO descriptor: pr ? gpr cr s , gpr cr s : gpr cr

Template	OPCODE	SRC1	SRC2	DEST1
T0	< 103, 104 >	< 105, 106 >< 110 · · · 115 >	< 107, 108 >< 116 · · · 123 >	< 109 >< 125 · · · 130 >

3 Application statistics

Application performance summary

VLIW execution time with perfect cache	3.235e+07 cycles
Total execution time with cache	34758920 cycles

The table above shows the overall application performance for the given architecture with a perfect cache and with the cache whose parameters are in Section 1.2.

3.1 Achieved instructions per cycle

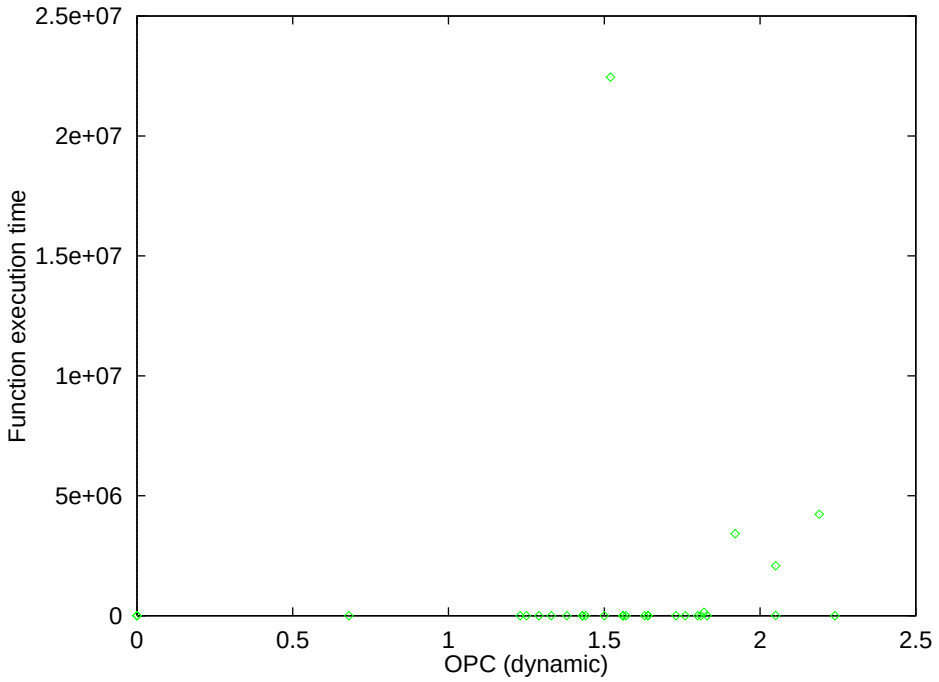

Figure 2: Plot of function execution time vs. OPC

Figure 2 is a scatter plot of all the functions that were called at least once during the execution of the program. The x-axis is the dynamic average of instructions scheduled per cycle for a function and y-axis is the total execution time of a function over all calls.

The OPC shown in this graph does not take cache effects into account.

3.2 Number of functions responsible for cumulative execution time

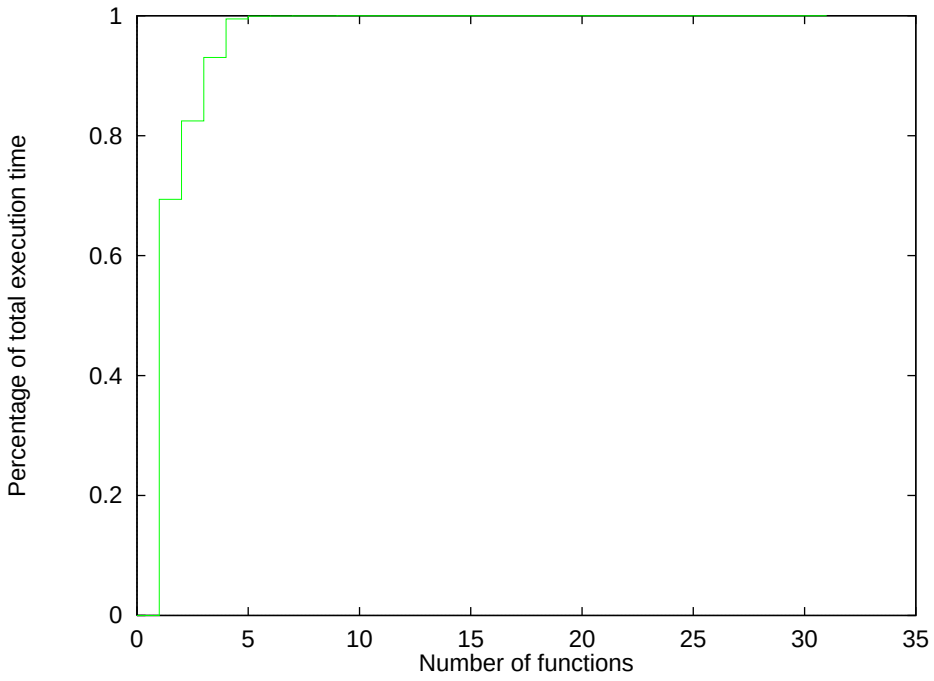

Figure 3: Cumulative execution time distribution over the most important functions

Figure 3 shows the cumulative execution for the most important functions in this application. The x-axis is the number of functions in the execution time and y-axis is the percentage of overall execution time these functions are responsible for.

As shown in this figure, top ten functions account for 99 percent of the total execution time. The top three most important functions are _systolic_dct, _EN_Encode_Block, and _EN_Encode_Scan_Gray responsible for 93 percent of execution.

3.3 Most important functions

Table 9 lists the top 25 most important functions in their order of importance in their contribution to total execution time. The columns in this table are the name of the function, execution time of this function as a percentage of total execution time, the cumulative percentage of total execution time including this function, dynamic average issued operations

Table 9: Most important functions

Function	exec. time %	cumul. exec. time %	OPC	static op count	cumul. static op count %
_systolic_dct	69	69	1.52	693	5
_EN_Encode_Block	13	82	2.19	1083	14
_EN_Encode_Scan_Gray	10	93	1.92	2965	39
_EB_Write_Bits	6	99	2.05	228	41
_EN_Encode_DC	0	99	1.82	132	42
_BuildHuffmanTable	0	99	2.05	2757	65
_Scale_Matrix	0	99	1.64	1276	75
_Fill_Winograd_Quant_Table	0	99	0.68	40	76
_EP_Write_DHTs	0	99	2.24	685	81
_Read_Next_Rows_From_File	0	99	1.29	18	82
_EB_Read_Next_Rows	0	99	1.25	15	82
_EP_Write_DQT	0	99	1.80	42	82
_Write_Bytes_To_File	0	99	1.33	16	82
_getint	0	99	1.76	157	84
_EB_Write_Bytes	0	99	1.63	78	84
_main	0	99	1.73	407	88
_EN_Encode	0	99	1.56	709	93
_read_ppm	0	99	1.64	134	95
_EP_Write_SOF	0	99	1.81	113	96
_EJ_Encode_JPEG	0	99	1.57	47	96
_EP_Write_SOS	0	99	1.83	75	97
_EP_End	0	99	1.56	25	97
_EJ_Encode_JPEG_File_To_File	0	99	1.56	25	97
_EP_Begin	0	99	1.50	27	97
_EP_Write_EOI	0	99	1.43	20	97

per cycle, the number of operations in the function body and the cumulative percentage of total number of operations including this function.

3.4 Breakdown of static operation count

Table 10: Static operation count components

Function	int%	float%	mem%	cmpp%	pbr%	branch%
_systolic_dct	60	0	37	0	0	0
_EN_Encode_Block	44	0	17	9	14	14
_EN_Encode_Scan_Gray	57	0	27	3	6	6
_EB_Write_Bits	43	0	31	5	10	10
_EN_Encode_DC	45	0	26	3	12	12
_BuildHuffmanTable	53	0	33	3	4	4
_Scale_Matrix	51	0	32	3	6	6
_Fill_Winograd_Quant_Table	40	15	20	5	10	10
_EP_Write_DHTs	52	0	35	3	4	4
_Read_Next_Rows_From_File	50	0	27	0	11	11
_EB_Read_Next_Rows	40	0	33	0	13	13
_EP_Write_DQT	52	0	30	2	7	7
_Write_Bytes_To_File	43	0	31	0	12	12
_getint	31	0	22	9	17	17
_EB_Write_Bytes	42	0	33	3	10	10
_main	39	0	25	4	15	15
_EN_Encode	45	0	20	4	14	14
_read_ppm	37	0	29	4	14	14
_EP_Write_SOF	53	0	29	3	7	7
_EJ_Encode_JPEG	53	0	34	0	6	6
_EP_Write_SOS	46	0	26	5	10	10
_EP_End	40	0	32	4	12	12
_EJ_Encode_JPEG_File_To_File	44	0	40	0	8	8
_EP_Begin	40	0	25	3	14	14
_EP_Write_EOI	45	0	35	0	10	10

Table 10 shows the breakdown of static operations into categories for the same top 25 most important functions as in Table 9. The columns in this table are the name of the function, and the percentage of integer, floating-point, memory, compare-to-predicate, prepare-to-branch and branch instructions in the static operation count.

3.5 Opcode usage

Table 11 lists the static and dynamic usage statistics of all PlayDoh opcodes in the order of their dynamic importance. The **integer** operations are shown in black. The floating-point operations are in red, the compare-to-predicate, prepare-to-branch, and branch operations are in green and finally load and store operations are in cyan.

Table 11: Opcode usage statistics

Opcode	stat. wght	dyn. wght	dyn. sum
ADD_W	33.2	37.5	37.5
L_W_C1_C1	14.4	15.1	52.6
S_W_C1	12.1	13.4	66
SHL_W	2.8	6.1	72.1
SHRA_W	1.9	5.9	78
PBRR	7.3	3.2	81.2
BRCT	4	2.8	84
ADDL_W	3.7	2.5	86.6
SUB_W	0.4	2.1	88.7
L_H_C1_C1	0.3	1.6	90.4
S_H_C1	0.9	1.4	91.8
EXTS_H	0.2	1.4	93.3
MOVE	8.4	1.1	94.3
CMPP_W_LEQ_UN_UN	0.7	0.9	95.2
CMPP_W_LT_UN_UN	0.8	0.8	96
L_B_C1_C1	0.7	0.8	96.8
MPY_W	0.3	0.8	97.6
CMPP_W_GT_UN_UN	0.8	0.6	98.2
CMPP_W_EQ_UN_UN	0.4	0.5	98.7
AND_W	0.7	0.4	99.1
S_B_C1	0.7	0.2	99.3
OR_W	0	0.2	99.4
BRL	1.5	0.1	99.6
PBRA	0.3	0.1	99.7
RTS	0.3	0.1	99.8
BRU	1.9	0.1	99.9
CMPP_W_GEQ_UN_UN	0.8	0	100
CMPP_W_LGEQ_UN_UN	0	0	100
CMPP_W_NEQ_UN_UN	0.4	0	100
DIV_W	0.1	0	100
FL_S_C1_C1	0	0	100
FL_D_C1_C1	0	0	100
CONVDW	0	0	100

Table 11: Opcode usage statistics

Opcode	stat. wght	dyn. wght	dyn. sum
CONVWS	0	0	100
FDIV_S	0	0	100
CONVSD	0	0	100
FADD_D	0	0	100
FMPY_S	0	0	100
CMPP_W_LGT_UN_UN	0	0	100
EXTS_B	0	0	100
MPYL_W	0	0	100