Automatic Design of VLIW and EPIC Instruction Formats

Shail Aditya, B. Ramakrishna Rau, Richard Johnson†
Compiler and Architecture Research
HPL-1999-94
April, 2000

Email: {aditya,rau}@hpl.hp.com, rjohnson@transmeta.com

instruction format design, template design, instruction-set architecture, abstract ISA, concrete ISA, VLIW processors, EPIC processors, HPL-PD architecture, instruction encoding, bit allocation, affinity allocation, application-specific processors, design space exploration

Very long instruction word (VLIW), and in its generalization, explicitly parallel instruction computing (EPIC) architectures explicitly encode multiple independent operations within each instruction. The processor's instruction-set architecture (ISA) specifies the interface between hardware and software, while its instruction format specifies the precise syntax and binary encodings of all the instructions in the ISA. A designer of instruction formats must make efficient use of the available hardware resources and make intelligent trade-offs between decoder complexity and instruction width. Simple encodings lead to faster and less expensive decode hardware, but increase instruction width. Wider instruction formats lead to increased code size and more expensive instruction caches and instruction data paths. In embedded systems, code size is often a major component of total system cost, since the program is stored in ROM. In this report, we present an algorithmic approach to automatic design of high-quality VLIW/EPIC instruction formats. Our design process can be used to explore a large design space to find good designs at varying cost-performance points. This is also essential for automated design-space exploration of application-specific VLIW/EPIC processors.

† Richard Johnson is currently with Transmeta Corporation. He contributed to this research while he was still at Hewlett-Packard Laboratories.

1 Introduction

Whereas the workstation and personal computer markets are rapidly converging on a small number of similar architectures, the embedded systems market is enjoying an explosion of architectural diversity. This diversity is driven by widely varying demands on performance and power consumption, and is propelled by the possibility of optimizing architectures for particular application domains. Designers of these application specific instruction-set processors (ASIPs) make trade-offs between cost, performance, and power consumption, using automated tools wherever possible.

Although there has been a fair amount of work done on providing the capability to automatically design the architecture of a sequential ASIP – primarily a matter of designing the opcode repertoire – there has been relatively little work in the area of automatic architecture synthesis of very long instruction word (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 [6, 7, 10]. The automatic design of a non-trivial instruction format, and the synthesis of the corresponding instruction fetch and decode microarchitecture 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 microarchitecture phases of a VLIW design project.

The goal of our PICO (Program-In-Chip-Out) design project is to fully automate this process, so that a family of optimized ASIP designs is generated automatically from an application program. PICO is a system synthesis and design exploration tool which performs hardware-software co-synthesis. In addition to a 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.

1.1 Focus of this report

The subject of this report is automatic design of high quality instruction formats, which is a necessary capability for an automatic architecture design-space exploration tool such as PICO-VLIW. Design quality is a function of both instruction width, which determines code size (and hence memory cost), and decode complexity, which affects the processor cost and performance to be evaluated at each design point within the exploration space. Automatic instruction format design is a useful capability in other situations as well. For example, this capability would be a useful tool in manual processor design, or in re-architecting the next generation of an existing processor architecture, or for customizing an architecture to a specific application or application domain. In these cases, the design space constraints can also be extracted from a previously existing instruction format.

The main contribution of this report is to formalize and describe our methodology for automatic design of instruction formats for architectures based on the VLIW design philosophy, or its generalization, explicitly parallel instruction computing (EPIC). This methodology may also be used to design single-issue instruction formats. The output of our scheme is a set of instruction templates with bit specifications for its various fields as they may be described in an architecture manual. We describe how this information is used automatically in a retargetable assembler. The system also produces the decode tables necessary to generate instruction decode logic in the given processor. Finally, we describe how to customize the instruction format for a particular application thereby reducing its overall code size.

In the rest of this section, we present a brief overview of the PICO-VLIW design system, which provides the context within which instruction formats are designed automatically. Then, we present a brief overview of the instruction format design process which would also serve as an outline of the rest of this report.

1.2 The PICO-VLIW architecture synthesis system

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-

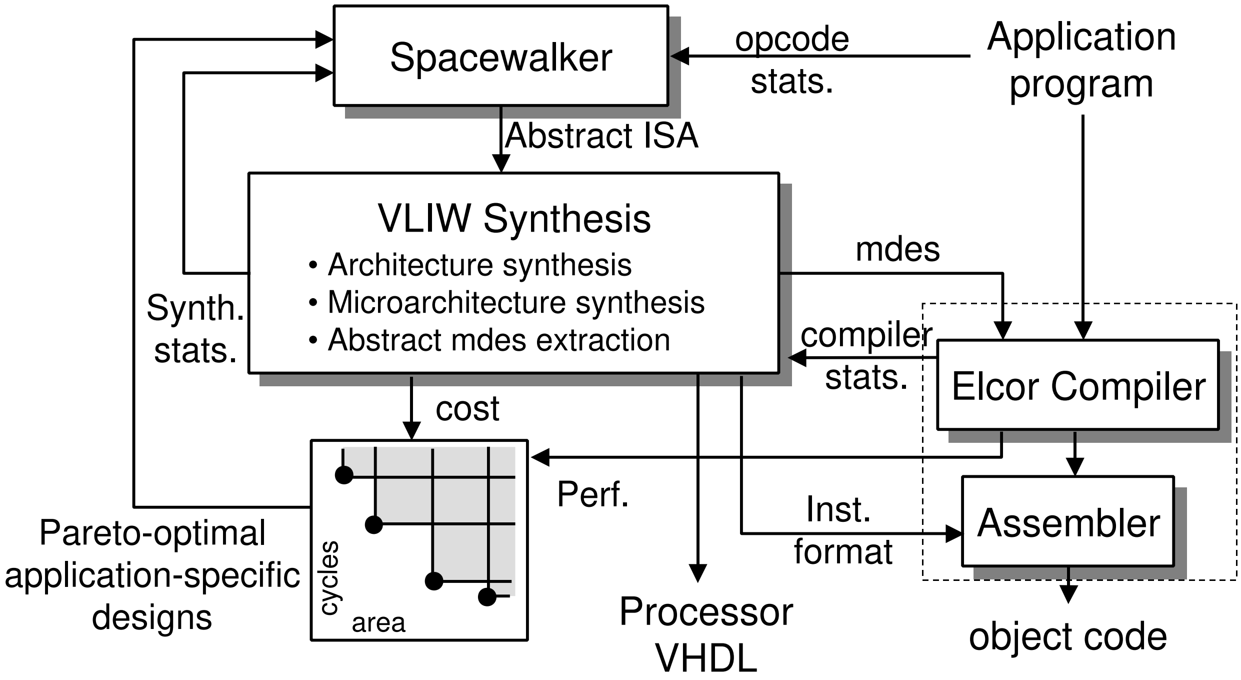


Figure 1: The PICO-VLIW design system.

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 report.

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, *i.e.*, the instruction format, and microarchitecture, *i.e.*, the datapath and controlpath, as well as a machine-description database used to retarget the compiler and the assembler. 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. More details on the PICO-VLIW system are provided in [18, 2].

1.3 Overview of instruction format design

One of the key steps in the design of a VLIW processor is defining its concrete instruction-set architecture (ISA) or its instruction format. It forms the interface between the hardware and the software and facilitates efficient utilization of the available hardware resources. The PICO-VLIW system automatically designs an optimized instruction format for each target processor it generates while making intelligent trade-offs between the application code size and the hardware control complexity. The various steps involved in this process are shown pictorially in Figure 2 and are described below.

Specifying an architecture. The primary input to the instruction format design system is a high-level, abstract specification of the ISA of the target processor called the *archspec*, which can either

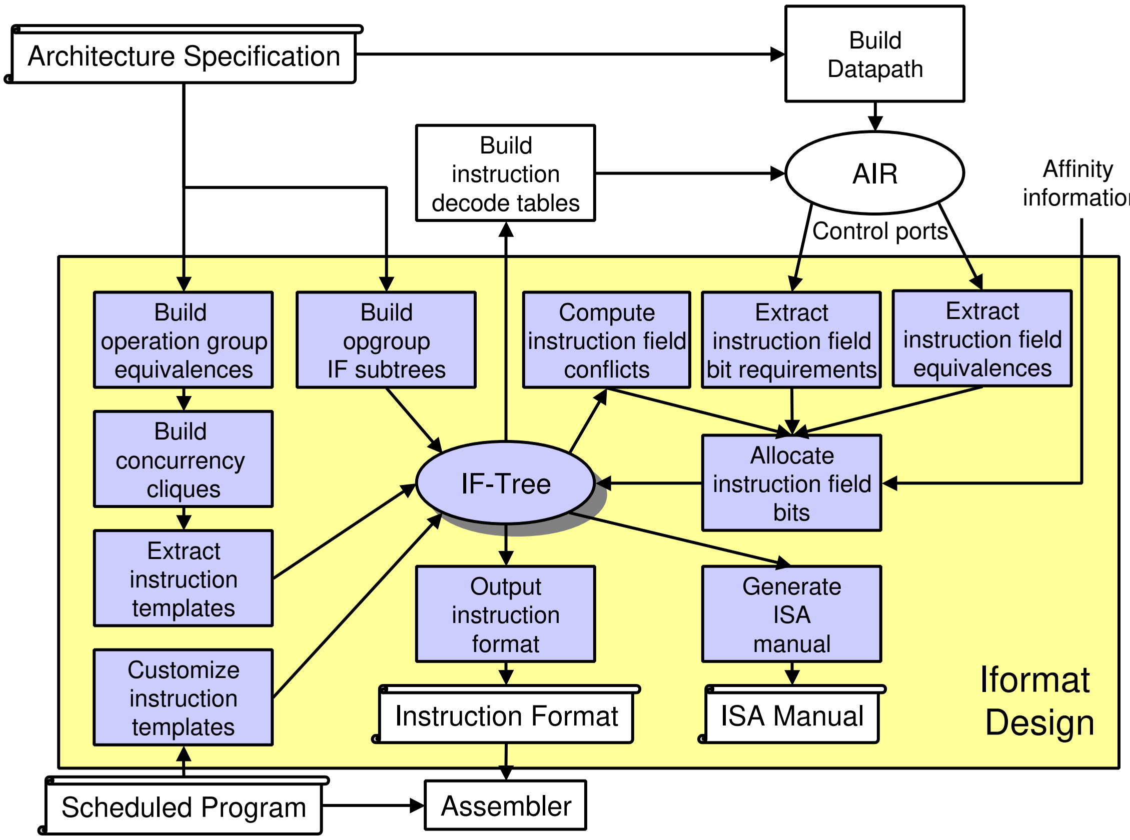


Figure 2: Instruction format design flow.

be specified manually or automatically by the Spacewalker. This specification drives both the instruction format design and the microarchitecture synthesis within the PICO framework. The archspec specifies the opcode repertoire and a set of ILP constraints for the target processor and is described in more detail in Section 2. A secondary input to the instruction format design process, specified in our architecture intermediate representation (AIR), is the set of control ports in the processor datapath that need to be provided with control information from the instruction.

The IF-tree. As shown in Figure 2, the instruction format design flow is centered around the *IF-tree* data structure which is a structural representation of a hierarchical grammar for the instruction format being designed. Various modules of the instruction format design system contribute in building and decorating the IF-tree. The structure of this tree is described in detail in Section 4.

The IF-tree also contains all the target-specific information needed by a retargetable assembler in order to assemble a given program to produce object code. This interface is described in detail in Section 9.

Building instruction templates. The key step in building the IF-tree is to identify one or more *instruction templates* possibly of different widths, and then to characterize its various instruction fields. An instruction template consists of one or more concurrent operation slots each of which encodes a set of mutually exclusive operations. The process of identifying the templates based on the specified archspec is described in detail in Section 5 and consists of the following steps.

Reducing the template design space – The space of valid instruction templates for a given processor is combinatorially huge. Therefore some systematic operation classification and grouping scheme is needed to reduce the space while obeying the ILP constraints laid out in the archspec.

Building concurrency cliques – Next, the instruction templates for the target processor are determined by finding unique sets of operations that may be issued concurrently on that processor without over subscribing the available hardware resources. Algorithmically, this corresponds to finding cliques in a graph formed by the concurrency relations among groups of operations as specified by the archspec.

Building the IF-tree – Each concurrency clique gives rise to a unique instruction template which is then represented in the IF-tree. Subtrees specifying the opcode and the operands of the various concurrent operations within each template are also represented in the IF-tree. The leaves of the tree, the *instruction fields*, encode information either directly obtained from the datapath such as register source and destination operands, or that derived from the instruction register such as opcodes, immediate literals, and operand select bits etc.

Setting up the bit allocation problem. Once the IF-tree is defined, we are ready to setup the instruction field bit allocation problem. This consists of the following steps that are described in detail in Section 6.

Identifying instruction field bit-requirements – Each instruction field specifies how many bits it needs in order to control the corresponding datapath control port in the context of its position in the IF-tree.

Setting up field conflicts – Next, we setup the bit allocation problem for all instruction fields by giving a data flow formulation for computing conflicts between them. Fields that are required within the same instruction template must be allocated to disjoint bit positions, and are therefore said to conflict.

Computing field affinities – Fields that do not conflict may be allocated to the same bit positions. Our bit allocation strategy permits certain fields to be aligned with each other based on their *affinity*, i.e. fields associated with the same datapath resources are aligned to the same bit positions within the instruction register, resulting in reduced control complexity in hardware.

Bit resource allocation. Given the instruction fields, their bit requirements, and their conflict and affinity specifications, a resource allocation algorithm is used to assign bit positions to all fields. Our algorithm is a generalization of Chaitin's graph coloring approach for register allocation [4]. Various heuristics are used to reduce the overall size of the instruction and the decode complexity. This step is described in Section 7.

Format Optimizations Prior to setting up the bit allocation problem, the inputs to the design step may be modified in order to produce a more efficient instruction format. These modifications are geared towards optimizing one or more cost metrics such as overall code size, dynamic cache performance, decode complexity, decode latency etc. In Section 8, we discuss the following two classes of instruction format optimizations.

Transforming the IF-tree – This class of optimizations modify the hierarchical structure of the IF-tree to create specialized choices (that would have shorter width) or modify existing choices so that the overall cost metric is improved.

Choosing better encoding – This class of optimizations chooses more efficient encodings (e.g. variable-width selection codes) of the various existing choices in the IF-tree so as to minimize the cost metrics.

Machine-description driven assembly. An assembler requires the exact instruction format of the target processor in order to translate assembly-level programs to object code. This information is usually built into the assembler and is geared towards a specific target architecture. Alternatively, our PICO-VLIW system uses a retargetable assembler driven by a **machine-description database** (**mdes**). The assembler is structured to use an abstract query interface during assembly that provides the details pertaining to a specific instruction format design. Section 9 discusses the structure of the interface and the process of assembly using it.

Instruction decode and report generation. After bit allocation, the IF-tree data structure carries the complete instruction format design information which may be used by various other modules of the PICO-VLIW system. The mQS interface driving the assembly process is one such client. The relevant information from the IF-tree is exported to the mQS interface using an external file format. The system also generates decode logic tables which define how to decode an instruction and provide control signals to the various control ports in the datapath. Finally, a human-readable instruction-set architecture manual is also generated automatically. These uses are described in Section 10.

2 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** (**archspec** 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.

2.1 Operation Groups and Exclusions

At an abstract level, an architecture specification need only specify the functionality of the hardware implementation in terms of its operation repertoire and the desired performance level. Then the exact structure of the implemented processor 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 operations that it implements and the level of parallelism that exists amongst them.

For convenience, the various instances of HPL-PD operations for a given processor are grouped into **Operation Groups** (**opgroups** for short) each of which is a set of operation instances that are similar in nature in terms of their latency and access to physical register files and are expected to execute always on the same functional unit. For this reason all operations within an opgroup are required to be mutually exclusive with respect to operation issue, e.g. add and subtract operations issued to the same ALU. Conversely, operations occurring within different opgroups are allowed to execute in parallel. The parallelism of the processor may be further constrained by placing two or more opgroups into an **Exclusion Group** which makes all their operation 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 processor is specified below[1]. The specification language we use is the database language HMDES Version 2 [9] that organizes the information into a set of interrelated tables containing rows of records each of which contain zero or more columns of named property values.

```
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));
```

[1]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.

```
  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 opgroups, two move operation groups, and one each of multiply, compare and shift groups. These opgroups are classified into several independent exclusion groups denoting sharing relationships among the opgroups. Each opgroup 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 opgroup but are not shown here since they are not relevant to this discussion.

2.2 Register Files and Operation Formats

The archspec specifies additional information to describe the physical register files of the processor and the desired connectivity of the operations to those files. A **Register File** entry defines a physical register file of the processor and identifies its width in bits, the registers it contains, and a virtual file specifier that specifies the types of data it can hold [12]. 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** (also known as **IO format** or **IO descriptor**) entries each specify the set of choices for source/sink locations for the operations in an opgroup. Each operation format consists of a list of **IO-sets** (also known as **Field Types**) that determine the set of physical register file choices for a particular operand. For predicated operations, a separate predicate input IO-set 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 processor 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 an IO-set specification as a source or sink of an operand. IO-sets for the predicate, source and destination operands are combined to form the valid operation formats for each opgroup. For example, the 2-input alu opgroup 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 IO-sets.

3 Instruction syntax

Before we get into the details of the instruction format design process, it is important to identify the general structure of the instruction formats that we design and the overall space of design choices available. This section describes the syntax of the types of instruction formats that we design automatically. It is important to recognize that these instruction formats are designed with the assumption that the processor is to be programmed in a high-level language, and that the assembly and machine code are by and large invisible to the programmer. Consequently, the instruction formats are not designed with any consideration given to human readability or convenience. Instead, the design process emphasizes two often competing objectives: the minimization of code size and hardware complexity. We briefly discuss this trade-off below while introducing the terminology used in the rest of the report.

Explicit instruction-level parallelism is a defining property of VLIW and EPIC; each **instruction** is able to specify a set of operations that are to be issued simultaneously. This property is termed MultiOp [15]. A MultiOp instruction contains multiple **operation slots**, each of which specifies one of the operations that are to be issued simultaneously. An **operation** is the smallest unit of execution in a VLIW or EPIC processor. Each operation is the equivalent of a conventional RISC or CISC instruction in that it, typically, specifies an opcode, one or two source operands, and a destination operand, although it may specify additional source or destination operands, depending on the nature of the opcode.

We define the **canonical instruction format** to be the MultiOp instruction format that has an operation slot per functional unit. The operation slots need not be of uniform width; each operation slot can use exactly as many bits as it needs. This conserves code space. Furthermore, the correspondence between an operation slot and a functional unit can be implicitly encoded by the position of the operation slot within the instruction–a further saving in code space over having to specify this mapping explicitly, somewhere in the instruction.

However, when either the parallelism in the hardware or that found by the compiler is unable to sustain the level of parallelism permitted by the canonical format, some of these operation slots will have to specify no-op operations. The exclusion groups specified in the archspec can lead to situations in which none of the operations in any of the opgroups corresponding to a given functional unit can be issued because each such opgroup is mutually exclusive with at least one other opgroup (on another functional unit) from which an operation is being issued. The compiler, for its part, may just be unable to find in the program the level of ILP needed to issue an operation in every operation slot. These no-ops lead to a wastage of code space.

Worse yet, the schedule created by the compiler might be such that there is no operation whatsoever scheduled to issue on certain cycles. If the processor does not support hardware interlocks on results that have not yet been computed, this situation requires the insertion of one or more MultiOp instructions containing nothing but no-ops. (The need for these no-op instructions reflects the fact that a program for such a processor represents a temporal plan of execution, not merely a list of instructions.) These explicit no-op instructions can be eliminated quite simply by the inclusion of a multi-noop field in the MultiOp instruction format which specifies how many no-op instructions are to be issued, implicitly, after the current instruction [3].

The more challenging problem is to get rid of code wastage caused by explicit no-op operations in an instruction that is not completely empty. A variety of no-op compression schemes can be devised to address this problem [14]. The one that we employ involves the use of multiple instruction formats or **templates**, each of which provide operation slots for just a subset of the functional units. The rest of the functional units implicitly receive a no-op, avoiding wastage of code space. The templates are selected in one of two ways. Firstly, the archspec might have been specified in such a way that it is impossible to issue an operation on all functional units simultaneously. Clearly, it only makes sense to provide those templates that correspond to those maximally concurrent sets of functional units upon which it is permissible to issue operations simultaneously. We refer to these as the **minimal templates**. No template, that is a superset of any of them, is of interest.

However, it could be the case that certain subsets of these maximally concurrent sets have a high frequency of occurrence in the scheduled code of the program for which we are customizing the instruction format. The provision of additional templates, that correspond to these statistically important subsets, further reduces the number of explicit no-ops. We refer to these as **custom templates**. By using a variable-width, multi-template instruction format, we are able to accommodate the widest instructions where necessary, and make use of compact, restricted instruction formats for much of the code. We discuss our strategy for custom template design in Section 8.

The trade-off involved in the choice of an instruction format is between code compaction and the complexity of the instruction datapath from the instruction cache, through the instruction register, and to the functional units, register files, and other portions of the datapath. The canonical format has instructions which are all of the same width. If the **instruction packet**–the unit of access from the instruction cache–has the same width as well, then instructions can be fetched from the cache and directly placed in the instruction register. On the other hand, no-op compression schemes yield variable-width instructions. The instruction packet size and the instruction register width must be at least as large as the longest instruction. When the current instruction is shorter than this, the unused portion of the contents of the instruction register must be shifted over to be correctly aligned to serve as the start of the next instruction and, if necessary, another word must be fetched from the instruction cache.

Another factor contributing to the hardware complexity is the distribution of the various instruction fields from the instruction register to the appropriate control ports in the datapath–opcode ports of the functional units, address ports of register files and data multiplexers and demultiplexors situated between the functional units and the register files. This is trivial with the canonical format. Each functional unit's operation slot is located in a specific segment of the instruction register and can be directly connected to it. When a no-op compression scheme is used, the fields of the corresponding operation slot may be in one of many places in the instruction register. A multiplexing network must be inserted between the instruction register and the control ports of the datapath. The shifting, alignment and distribution networks increase the complexity and the cost of the processor. These issues are discussed in greater detail in [18, 2].

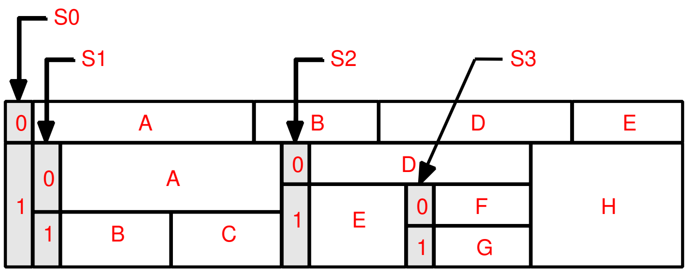
The cost of the distribution network can be reduced by designing the instruction templates in such a way that the instruction field corresponding to each control port in the datapath occupies, as far as possible, the same bit positions in every instruction template in which it appears. If one were completely successful in doing this, the distribution network would again become trivial. However, this can lead to a greater wastage of code space. Our instruction format design heuristics attempt to strike a compromise between these competing goals, reducing the amount of multiplexing in the distribution network without causing too much wastage of code space.

The complexity of the shifting and alignment network can be partially contained by requiring that the width of all instruction templates be a multiple of some number of bits, which we refer to as the **quantum**. As a result, all shift amounts can only be a multiple of this quantum, thereby reducing the degree of multiplexing in the shifting and alignment network. The adverse effect of quantization is that the template width must be rounded up to an integral number of quanta, potentially leading to some wastage of code space.

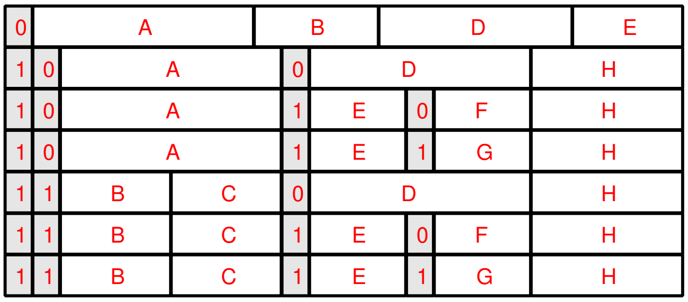
3.1 Hierarchical multi-template instruction formats

In the canonical instruction format, all instruction fields are encoded in disjoint positions within a single, wide instruction. A hierarchical, multi-template instruction format allows mutually exclusive instruction fields (those that are not used simultaneously in the same instruction template) to be encoded in overlapping bit positions, thereby reducing the overall instruction width. In discussing the syntax of such instruction formats, it is useful to think in terms of two levels of syntax. The first grammar, that of the instruction templates, has operation slots as its terminal symbols. Two designs of interest that we discuss here are multi-level templates and two-level templates. The second grammar, that of the operation slot, has instruction fields as its terminal symbols and is described in Section 3.2.

Multi-level templates. We describe the abstract syntax of the multi-level template meta-grammar using an extended BNF in which X^+ represents one or more instances of X. The syntax is as fol-



(a) A multi-level template format



(b) The seven distinct templates for the multi-level template format

Figure 3: Instruction template formats. The shaded fields are the select fields. (The width of an operation slot, as shown in the figure, is not intended to bear any relationship to the number of bits that it requires in the instruction format.)

lows:

$$
\begin{array}{rcl}
\text{template} & ::= & \text{OR-set} \\
\text{OR-set} & ::= & \text{alternative}^{+} \\
\text{alternative} & ::= & \text{operation-slot} \mid \text{AND-list} \\
\text{AND-list} & ::= & \text{OR-set}^{+}
\end{array}
$$

An OR-set represents a choice between the members of a set of alternatives. An alternative is either an operation slot or an AND-list. An AND-list consists of one or more OR-sets, and represents a choice between one of the tuples obtained by taking the Cartesian product of the alternatives in each of the OR-sets. A sentence in this meta-grammar is a tuple of operation slots, each of which can issue an operation concurrently. This set of tuples constitutes the grammar of one specific instruction template that supports a set of instructions.

The set of instructions supported by an instruction template grammar is determined by the choice of operations that can be issued from its various operation slots. An operation slot can specify one operation out of a set of mutually exclusive operations. Consequently, the natural set of operations to associate with an operation slot is an opgroup. This is because the operations that are grouped together in an opgroup are expected to be executable on the same functional unit and may share access to source and destination operands. Note that the width of the operation slot has to be that of the widest operation in the opgroup.

As an example, consider a processor with eight opgroups: A, B, C, D, E, F, G and H. A possible multi-level template syntax for it is shown pictorially in Figure 3a. Associated with each OR-set is a select field which specifies which of the choices is selected. The highest level OR-set consists of two alternatives. Select field S0 specifies the selection. One alternative is an AND-list consisting of the opgroups A, B, D and E. The other alternative is an AND-list consisting of three OR-sets. The alternatives of the first OR-set are the opgroup A and the AND-list consisting of B and C. The corresponding selector field is S1. The alternatives of the second OR-set, with selector S2, are the opgroup D and an AND-list consisting of E and an OR-set with two alternatives, F and G, and a selector S3. The third OR-set consists of just the single opgroup H. There are seven distinct templates corresponding to this multi-level format, which are shown in Figure 3b.

Two-level templates. The other end of the spectrum of possibilities is to use a two-level template format. The abstract syntax for the two-level template meta-grammar is:

$$\begin{aligned} \text{template} &::= \text{OR-set} \\ \text{OR-set} &::= \text{AND-list}^{+} \\ \text{AND-list} &::= \text{operation-slot}^{+} \end{aligned}$$

The two-level format provides a choice between one or more AND-lists, where each AND-list consists of one or more operation slots. Each AND-list represents a template. Using the same example, there are as before the same seven templates shown in Figure 3b, but there is only a single select field (labeled T in Figure 4) that specifies one of the seven templates.

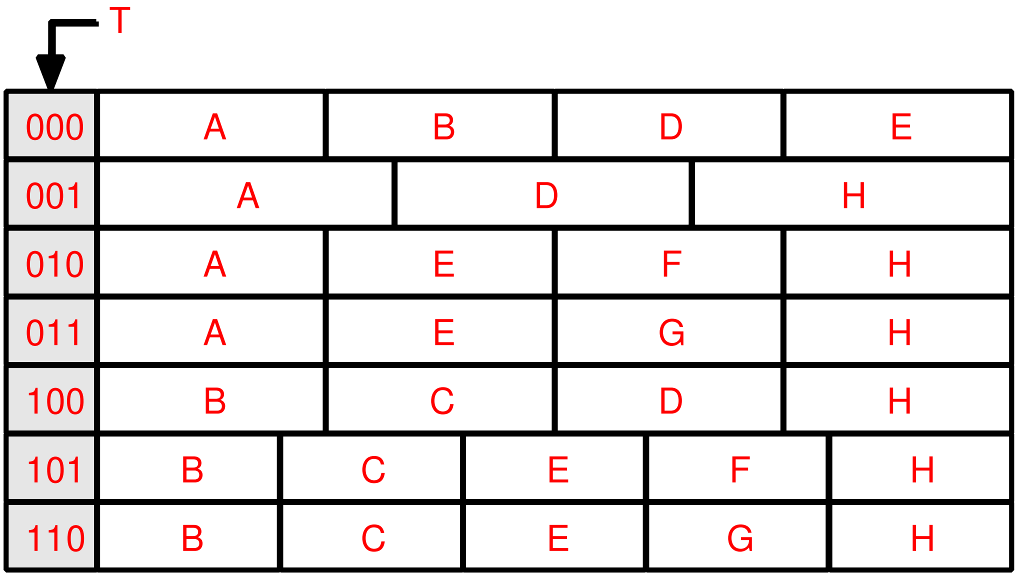


Figure 4: A two-level instruction template format. The shaded fields are the select fields. (The width of an operation slot, as shown in the figure, is not intended to bear any relationship to the number of bits that it requires in the instruction format.)

The multi-level syntax typically yields a more succinct specification of the template due to the Cartesian factoring provided by an AND-list of OR-sets. This leads to some significant benefits during the instruction format design process in terms of the sizes of data structures and the execution time of certain design algorithms. Another significant advantage is that it also leads to simpler decode hardware; in our example, whereas the two-level syntax requires one relatively large (3-input, 7-output) decoder, the multi-level syntax requires four relatively small (1-input, 2-output) decoders. In general, the number and the size of decoders needed in a multi-level scheme depends only on the number of OR-sets present and their immediate children, whereas in a two-level scheme it depends on the total number of distinct templates possible which could be very large even for architectures with a modest number of opgroups.

However, there are some important disadvantages of the multi-level syntax as well. With the multi-level syntax, determining the template corresponding to the current instruction is inherently a sequential process. For instance, one cannot know whether select field S3 even exists until S0 and S2 have been inspected and have both been found to have the value 1. Similarly, the position of select fields S2 and S3 (and hence the starting positions of opgroups D, E, F, G and H) can not be ascertained until the field S1 has been decoded because the size of opgroup A may not match the sum of the sizes of opgroups B and C. The consequence of this sequentiality is an increase in the time that it takes to determine the complete syntax of the current instruction, including the positions of various operations slots within it and its overall width. The latter is used by the instruction fetch control logic to identify the start of the next instruction.

The decoding can be parallelized, but at some cost in the complexity of the decoder; all bit positions that could possibly correspond to one of the select fields, in any one of the templates, must be supplied as inputs to the decoder. The situation can be improved somewhat by requiring that each select field, when present, is in the same bit position regardless of the values of the other select fields. (These fields could, if so desired, be assigned bit positions that are all contiguous, in effect yielding a single, variable-width template select field.) However, this only partially addresses the problem of decoding speed.

If fast, parallel decode is the priority, the two-level syntax is preferable. Decoder complexity for the resulting large number of templates may be reduced by minimizing the number of bits that serve as input to the decoder. This is achieved by having a single, fixed-width template select field which, for our example, requires that the decoder have only three input bits (from the template select field T) whereas the parallel implementation of the multi-level scheme requires four input bits (from select fields S0, S1, S2 and S3).

3.2 Our multi-template instruction format

The class of multi-template instruction formats we have chosen to design takes a middle ground between the above two schema. Although we basically use a two-level instruction format, we do make a small compromise in the direction of a multi-level syntax by treating each operation slot as an OR-set of opgroups instead of a singleton. This reduces the number of templates dramatically and, therefore, the width of the template select field and the template decoder's complexity. Nevertheless, the position and width of the various operation slots within each template is completely

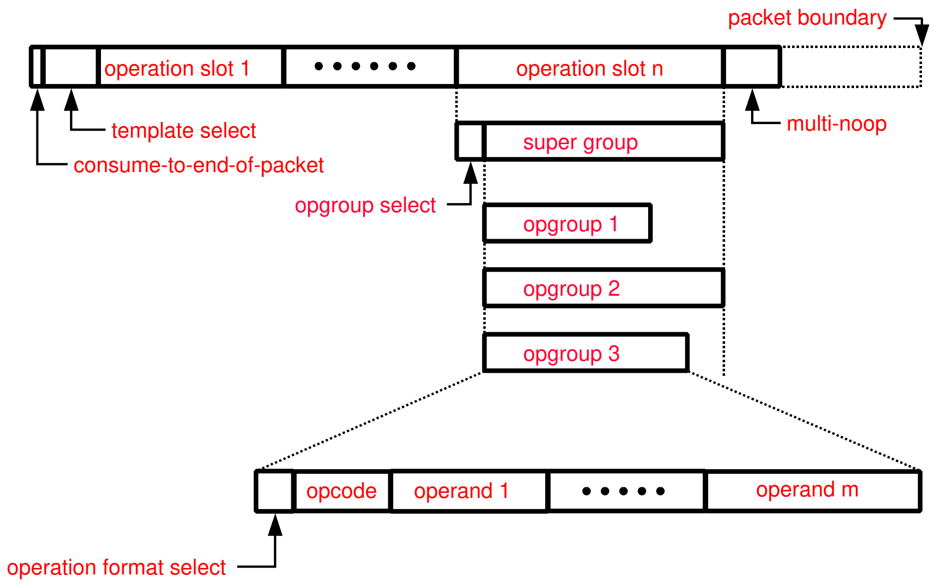


Figure 5: Our generic instruction format syntax.

determined by the value of the template select field and not by the choice of opgroups within each operation slot, thereby allowing us to determine the width of the current instruction faster.

The set of opgroups assigned to an operation slot is called a **super group**, which is taken to be a set of mutually exclusive opgroups all of which have identical mutual exclusion (or concurrency) relationships with every opgroup that is not part of the super group. This constraint ensures that such grouping does not affect the ILP of the target processor in any way as specified in the archspec. Each operation slot within an instruction now specifies one of the operations that are part of a super group. Sets of super groups that can be issued in parallel are combined into an instruction template.

The use of super groups can lead to a very great reduction in the number of templates. A template consisting of N super groups is the equivalent of a set of templates, each consisting of N opgroups, obtained by taking the Cartesian product of the N super groups. The reduction in the number of templates leads to the benefits that we are seeking: reduced template select width, reduced template decoder complexity and faster determination of the width of the instruction. But this is gained at a price; each operation slot in a template must be as wide as the widest opgroup in that super group. If the opgroups have highly disparate widths, it will result in a larger code size than would have been necessary. In Section 8, we outline a process of selectively splitting super groups into smaller super groups in order to achieve better compromises between code size and the number of templates.

We can now discuss the concrete syntax of our instruction format. Consider first the syntax of an instruction template as shown in Figure 5. The first bit of every instruction is a **consume-to-end-of-packet** (**EOP**) field that indicates whether the next instruction directly follows the current instruction or starts at the next instruction packet boundary. This capability is used by the assembler to prevent instructions that are branch targets from straddling an instruction packet boundary and is discussed in Section 9.

This is followed by the **template select** field that identifies one specific instruction template. This select field is in the same, fixed position within every instruction. An instruction format having t templates will need $\lceil \log_2(t) \rceil$ bits to encode the template select. From its value the instruction decoder understands the instruction's syntax and, therefore, how to interpret it, whereas the instruction sequencer determines the overall instruction width and, thus, the address of the next instruction.

Next come one or more operation slots. The template select identifies the number of operation slots, their width, and their bit positions. A template may contain some number of unused bits that arise due to quantizing the number of bits in the template. If so, these bits are opportunistically used to provide a **multi-noop** field that is used to specify the number of no-op cycles that are to follow the current instruction.

Next, consider the syntax of an operation slot. Each operation slot can specify one operation out of a super group. To fully specify an operation, the operation slot must unambiguously specify the syntax of the operation. To do so, the operation slot must specify both an opgroup within the super group and an operation format for that opgroup. The **opgroup select** field chooses amongst the various opgroups within the super group. Effectively, the opgroup select field also specifies the functional unit upon which the opcode is to be executed since, in general, different opgroups within the same super group may be assigned to different functional units but, by definition, all operations within one opgroup execute on the same functional unit.

Within an opgroup, the operations are partitioned based on their operation format. Accordingly, an operation slot has an **operation format select** field to choose amongst the various operation formats supported by the opgroup. As shown in Section 2, the operation format of an operation identifies the various choices of source and destination IO-sets for each operand. The need for multiple operation formats is illustrated there by the opgroup OG_alu_1. One operation format allows a literal field on the left port, while the other allows it on the right port. Presumably, a single, combined format allowing a literal field on either port was not specified in the archspec because it would then also permit the possibility of both ports being literals, which would widen the instruction template beyond what is acceptable.

In effect, this operation slot syntax factors a flat opcode name space into a multi-tier, variable-width encoding by selecting an opgroup within the slot's super group, an operation format within that opgroup, and finally an opcode with that format. In rare cases, this factorization may increase the encoding length by one bit per level. Note, however, that our approach does not preclude a flat encoding space; placing each operation in its own opgroup eliminates the factorization but requires a decoder, with a larger number of inputs, to jointly determine the functional unit, the operation format and the actual opcode.

Once the opgroup and the operation format have been determined, the number, width and position of each of its sub-fields is known. The syntax for an operation, as specified by the operation format, is similar to that of a traditional RISC or CISC instruction, consisting of an **opcode** field and a sequence of source and destination **operand specifiers**. In particular, the operation format may specify a predicate source operand if the processor supports predicated execution [15]. An operand specifier may, in general, be one out of a set of instruction fields that identify the exact kind and location of the operand. Operand specifiers with multiple choices have an **IO-set select** field to identify which instruction field is intended.

Instruction fields form the terminal symbols of our instruction format syntax. An instruction field is a set of bit positions intended to be interpreted as an atomic unit within some instruction context. Familiar examples are opcode fields, source and destination register specifier fields, and literal fields. Bits from each of these fields flow from the instruction register to control points in the datapath, often via decode logic. For example, opcode field bits flow to functional unit opcode ports, and source register field bits flow to register file read address ports. Another type of instruction field is the select field. Select fields encode a choice between disjoint alternatives and communicate this context to the decoder. For example, a select bit may indicate whether an operand field is to be interpreted as a register specifier or as a short literal value. In hardware terms, this select bit determines whether a multiplexer at the input of some functional unit selects a register file read port or some hardwired constant.

A systematic evaluation of the effectiveness of the various mechanisms presented above, such as the use of the EOP bit, the multi-noop field, and the variable-width and customized templates, appears in the paper [17]. In this report, we will restrict ourselves to the description of the process of automatically generating instruction formats of the form shown above and the algorithms used at various steps of that process.

3.3 Physical instruction format

The instruction format, as described thus far, could well convey the impression that the fields of an operation slot occupy contiguous bit positions within the instruction. We term this view of the instruction the **logical instruction format**. The actual or **physical instruction format** allows the fields within each template to be positioned in some permuted and discontiguous, but fixed, way that is specified by the template select. Furthermore, an individual field is also permitted to consist of a discontiguous set of bit positions. This aspect of the physical instruction format represents what is, perhaps, one of the more unconventional features of our instruction format, and reflects the fact that it is designed with hardware optimality in mind, and not the convenience of a human machine code programmer.

This new degree of freedom obtained in the physical format may be exploited to reduce the cost and the complexity of the hardware. The permutation applied to the fields of each template is selected in such a way as to minimize the complexity of the distribution network, i.e., to minimize the number of distinct positions, across all of the templates, in which the information required by a given datapath control port is to be found. This and other allocation heuristics are described in Section 7.2.

The correspondence between the logical and physical formats is established during the instruction format design process by specifying a mapping from the logical fields in each template to the bit positions occupied by that field in the physical format. An assembler and disassembler can make use of this map and the inverse map, respectively, to present the programmer with a view that corresponds to the logical instruction format.

Thus, the instruction format design process consists of two broad tasks. The first one is to define the one or more instruction templates of the processor, consistent with the constraints imposed by the archspec, and to identify the various logical instruction fields within each template. The second task is to assign bit positions to the logical fields in such a way that two fields, that can be present in the same instruction, occupy disjoint bit positions. Both tasks need to be performed in a manner that strikes a judicious compromise between minimizing code size and minimizing hardware complexity.

4 The instruction format tree

In order to facilitate the design of multi-template, hierarchical instruction formats as described in the last section, we first define an intermediate data structure, the **instruction format tree** (**IF-tree** for short), which represents the hierarchical relationship between various instruction fields. It can be viewed as a structural representation of the BNF grammar of a machine instruction as follows:

- An AND-list in the grammar is represented by an **AND-node** in the IF-tree, which is a conjunction (AND) of the subtrees at the next lower level.

- An OR-set in the grammar is represented by an **ANDOR-node** in the IF-tree, which is essentially a disjunction (OR) of the subtrees at the next lower level with one important addition–

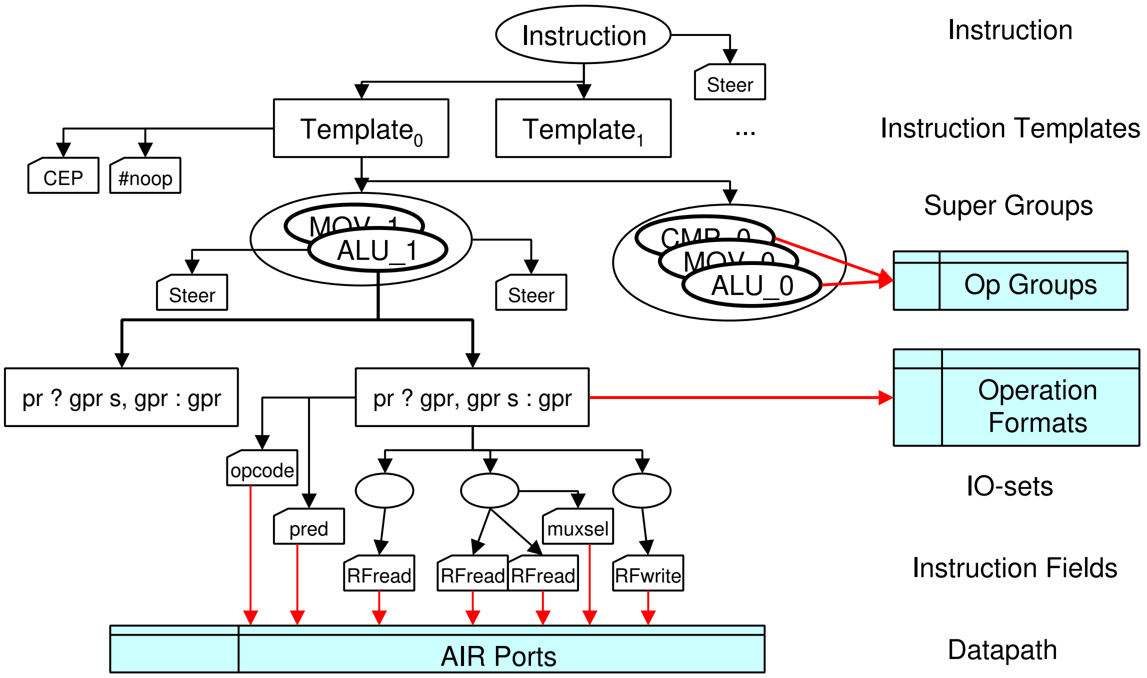


Figure 6: Structure of the instruction format tree.

there is an explicit select field placed in conjunction (AND) with all the subtrees that is used to select one of them. This is why this node is called an ANDOR-node.

- The leaves of the IF-tree are instruction fields; each leaf points to a control point in the data path.

Figure 6 illustrates the structure of an example IF-tree. The various levels of the tree are described below.

4.1 Structure of the instruction format tree

Instruction. The root of the tree is the overall machine instruction. This is an ANDOR-node consisting of a choice of instruction templates. A template select field is used to identify the particular template. An instruction format having t templates will need $\lceil \log_2(t) \rceil$ bits to encode the template select.

Templates. Each template is an AND-node that encodes sets of operations that issue concurrently. Since the number of combinations of operations that may issue concurrently is astronomical, it is necessary to impose some structure on the encoding within each template. Hence, each template is partitioned into one or more operation issue slots, each of which can specify one of a set of operations. Every combination of operations assigned to these slots may be issued concurrently.

Super Groups. The next level of the tree defines each of the concurrent issue slots. Each slot is an ANDOR-node supporting a super group which is a set of opgroups that are all mutually exclusive and have the same concurrency pattern. A select field chooses amongst the various opgroups within a super group.

Operation Groups. Below each super group lie operation groups as defined in the input archspec in Section 2. Each opgroup is an ANDOR-node that has a select field to choose amongst the various operation formats supported by the opgroup as shown in Figure 6.

Operation Formats. Each operation format is an AND-node consisting of the opcode field, the predicate field (if any), and a sequence of source and destination IO-sets. The traditional three-address operation encoding is defined at this level.

IO-sets. Each IO-set is an ANDOR-node consisting of either a singleton or a set of instruction fields that identify the register file(s) that can hold a particular operand. IO-sets with multiple choices have a select field to identify which instruction field is intended.

Instruction Fields. The leaves of the IF-tree consist of various instruction fields. Each instruction field corresponds to a datapath control port (refer Figure 7) such as register file read/write address ports, predicate and opcode ports of functional units, and selector ports of multiplexors. The various types of instruction fields are described below.

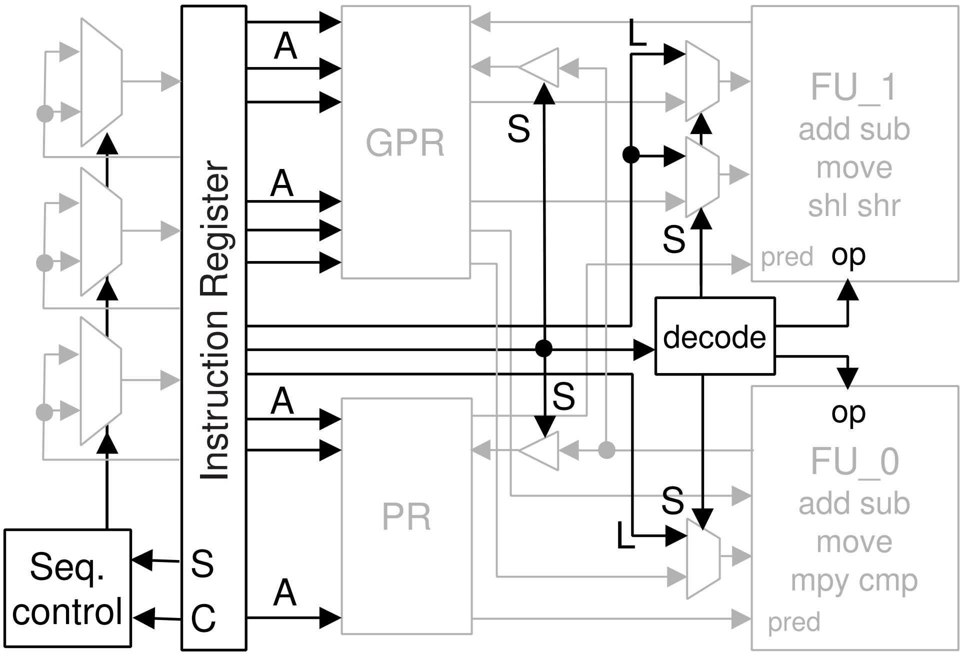


Figure 7: Various types of instruction fields controlling the datapath. Select fields (S), register address fields (A), literal fields (L), opcode fields (op), and miscellaneous control fields (C).

Select fields (S) – As mentioned earlier, at each level of the IF-tree that is an ANDOR-node, there is a select field that chooses among the various alternatives. The number of alternatives is given by the number of children, n, of the ANDOR-node in the IF-tree excluding the select field. Assuming a simple binary encoding, the bit requirement of the select field is then $\lceil \log_2(n) \rceil$ bits. We also consider variable-width encodings in Section 8.2.

Different select fields in the IF-tree are used to control different aspects of the datapath as shown in Figure 7. The root of the IF-tree has a template select field that is routed directly to the instruction unit control logic in order to determine the template width. Therefore, this field must be allocated at a fixed position within the instruction. The select fields at super group and opgroup levels determine how to interpret the remaining bits of the template and therefore are routed to the instruction decode logic for the datapath. The select fields at the level of IO-sets is used to control the multiplexors and tristate drivers at the input and output ports of the individual functional units to which that opgroup is mapped. These fields select among the various register and literal file alternatives for each source or destination operand.

Register address fields (A) – The read/write ports of various register files in the datapath need to be provided address bits to select the register to be read or written. The number of bits needed for these fields depends on the number of registers in the corresponding register file.

Literal fields (L) – Some operation formats specify an immediate literal operand that is encoded within the instruction. The width of these literals in specified externally in the archspec. Dense ranges of integer literals may be represented directly within the literal field, for example, an integer range of -512 to 511 requires a 10-bit literal field in 2's complement representation. On the other hand, a few individual program constants, such as 3.14159, may be encoded in a ROM or a PLA table whose address encoding is then provided in the literal field. If there are n such constants, the size of the literal field is then $\lceil \log_2 n \rceil$. In either case, the exact set of literals and their encodings must be specified in the archspec.

Opcode fields (op) – The opcode field bits are used to provide the opcodes encodings to the functional unit that has been assigned to execute them. If all the operations supported by a functional unit are represented within an opgroup that is assigned to execute on that functional unit, then it is possible to use the internal hardware encoding of opcodes within the functional unit directly as the encoding of the opcode field. In this case, the width of the opcode field is the same as the width of the opcode port of the functional unit and the bits are steered directly towards it.

It is often the case, however, that the functional unit assigned to a given opgroup may have many more opcodes than those present within the assigned opgroup. In this case, opcode field bits may be saved by encoding just the assigned opcodes in a smaller set of bits determined by the number of opcodes in that opgroup and then decoding these bits before supplying to the functional unit. In this case, the template specifier bits must also be used to provide the context for the opcode decoding logic.

Miscellaneous control fields (C) – Some additional control fields are present at the instruction level that help in proper sequencing of instructions. These consists of the consume to end-of-packet bit and the field that encodes the number of no-op cycles following the current instruction as shown in Figure 5.

5 Building minimal instruction templates

The design of instruction templates lies at the heart of the instruction format design process. As defined in Section 3.2, an instruction template in our scheme corresponds to an AND-list of super groups. As such, it defines a set, each of whose members is a set of operations that may be issued concurrently. In this section, we shall discuss the design of the minimal templates as specified by the archspec. In Section 8, we shall discuss the design of custom templates that take into account statistics pertaining to a given application.

5.1 Minimal template design flow

The pseudo-code representing the minimal template design flow appears in Figure 8. We discuss the various steps involved with the help of an example.

The archspec, described in Section 2, constrains which opgroups are mutually exclusive and, as a complementary relation, which opgroups may be executed in parallel. Consider the example in Figure 9. Starting from the archspec shown in Figure 9a as a mutual exclusion graph, which specifies the opgroups and the mutual exclusions between them, we can build the boolean exclusion matrix shown in Figure 9b. This matrix is just another representation of the graph in Figure 9a but is more convenient to work with. The complement of this matrix is the maximal concurrency matrix of Figure 9c. In both matrices, the diagonal entries are irrelevant. The corresponding concurrency graph (Figure 9e) is the complement of the mutual exclusion graph, i.e., every pair of nodes that are connected by an edge in one graph, have no connecting edge in the other graph, and vice versa. It is this graph that is the starting point for template design (Line 4 in Figure 8).

An exclusion constraint between two opgroups must be satisfied by all templates, i.e. operations within these two opgroups must never occur together in any template. On the other hand, a con-

```
procedure BuildMinimalTemplates(Graph archspec)
    // archspec specifies opgroup exclusion graph, we first build the concurrency graph
    BitMatrix exclusionMatrix = archspec.extractMatrix() ;
    BitMatrix concurMatrix = exclusionMatrix.complement() ;
    Graph concurGraph = Graph(concurMatrix) ;
    // reduce the concurrency graph with C-sets and E-sets
    IntSetList CESets = FindCESets(concurMatrix) ;
    for (each CEset in CEsets) do
        concurGraph.collapseNodes(CEset) ;
    endfor
    // find cliques in the reduced graph and expand into templates
    NodeSetList cliques = FindCliques(ϕ, concurGraph.allNodes() ) ;
    for (each clique in cliques) do
        Template newTemplate = new Template() ;
        for (each CEnode in clique) do
            if (CEnode is an E-set) then
                newTemplate.addSlot(CEnode.subNodes() ∪ NO-OP) ;
            else // (CEnode is a C-set)
                for (each node in CEnode.subnodes()) do
                    newTemplate.addSlot(node ∪ NO-OP) ;
                endfor
            endif
        endfor
        Record newTemplate ;
    endfor
```

Figure 8: Pseudo-Code for building the minimal instruction templates.

currency relation (i.e., the absence of an exclusion constraint) between two opgroups implies that the processor must be capable of issuing these operations simultaneously, within the same instruction, and therefore there should be some template in which these two operations can be specified together. More generally, for every set of opgroups that are pairwise concurrent, we need to have a template that permits the joint specification of that set of opgroups. That template may contain additional slots that can be filled with no-ops. Therefore, we do not have to generate a separate template for each possible set of concurrent opgroups; we only need a set of templates that together cover all possible sets of concurrent opgroups. In order to minimize the number of such templates, we need to find the largest possible sets of concurrent opgroups, *i.e.*, the cliques[2] in the concur-

[2]A clique of nodes within a graph is a subgraph in which every node is a neighbor of every other node and no other

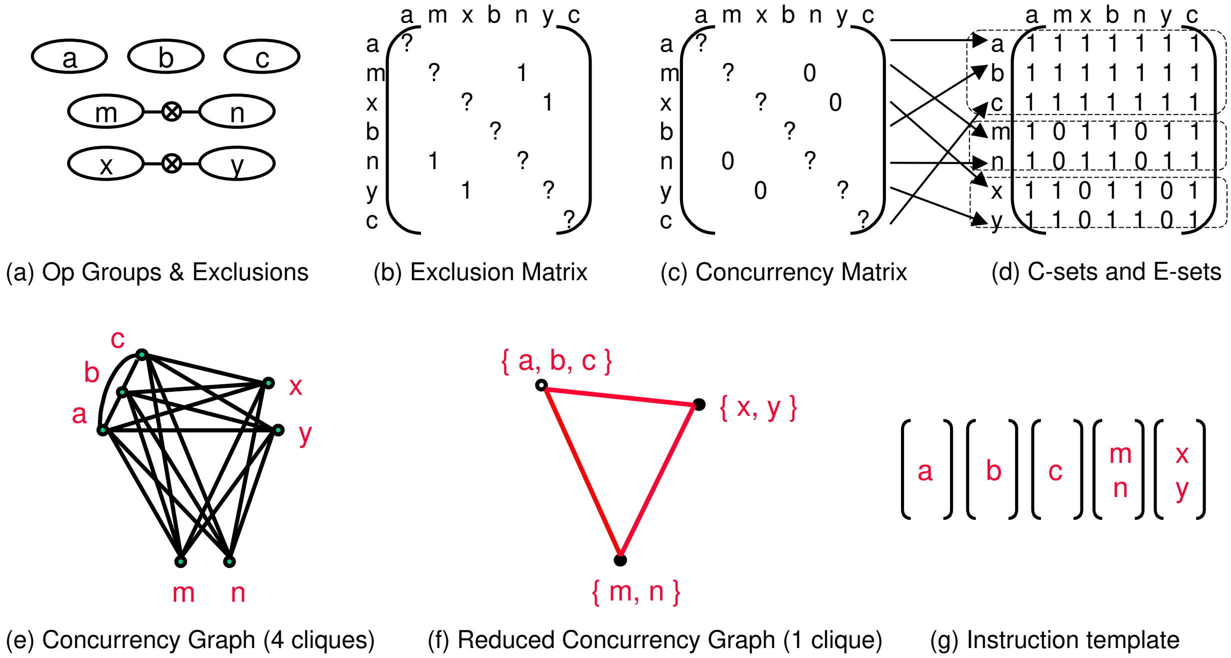


Figure 9: Using equivalent opgroups to reduce template design complexity.

rency graph. The set of templates, corresponding to the set of all cliques, constitute the minimal templates.

It is possible to use the opgroup concurrency matrix to find all the cliques. However, the number of cliques could be very large and this step may take a lot of time. Therefore, we first reduce the size of the concurrency graph by classifying the opgroups into sets of equivalent opgroups as shown in Figure 9d (Line 6). Two opgroups are said to be *equivalent* if they have the same set of concurrency neighbors. Two equivalent opgroups that are mutually exclusive are part of the same maximal **exclusion-equivalent set** (**E-set** for short). Such opgroups can replace each other in any template without violating any exclusion or concurrency constraint. Similarly, two equivalent opgroups, that are concurrent, are part of the same maximal **concurrency-equivalent set** (**C-set** for short). Such opgroups can always be placed together in the same template without violating any exclusion or concurrency constraints.

---

node from the graph may be added without violating this property.

The classification of opgroups into C-sets and E-sets induces a reduced concurrency graph as shown in Figure 9f (Line 8). We compute the cliques of this reduced graph (Line 11). For VLIW processors with multiple, identical functional units this graph reduction yields tremendous savings by reducing the complexity of the problem to just a few independent E-sets and a single clique. For a processor with shared resources and dissimilar functional units, the resulting number of cliques may be larger. Even so, the graph reduction reduces the complexity of the problem significantly. The cliques thus found can then be used to construct the instruction templates.

If we wanted templates to correspond to maximal sets of concurrent opgroups, each clique in the original concurrency graph would become a valid template. The classification of the opgroups into C-sets and E-sets is just an optimization to find all the templates quickly. In this case, the cliques found in the reduced concurrency graph would be expanded into a set of templates. This set is obtained by taking the Cartesian product of the E-sets in the clique; each template would contain one combination of opgroups out of the E-sets in the clique. There would be one operation slot per opgroup. In addition, the opgroups within each C-set would be expanded and would be present in every template in the set. There would be separate operation slots in the template for these opgroups.

However, as we noted in Section 3.2, we want templates to correspond to maximal sets of concurrent super groups, not opgroups. The larger these super groups are, the smaller will be the number of minimal templates. From this point of view, we would like each super group to be a maximal set of mutually exclusive opgroups which have identical concurrency relations with all opgroups that are not in that set. Of course, this is precisely the definition of an E-set, and so we define our super groups to be the E-sets that we initially constructed purely for reasons of computational complexity (Line 16). (In Section 8, we shall introduce other criteria for super group formation besides minimizing the number of templates. These will lead to a different set of super groups.)

If super groups are synonymous with E-sets, each clique in the reduced graph directly yields an instruction template. Each E-set in the clique corresponds to an operation slot, and all of the opgroups in the corresponding super group share that operation slot. A default no-op opgroup is also added to each operation slot. As before, each C-set in the clique is expanded and each of the opgroups (together with the no-op opgroup) gets a separate operation slot in the template (Line 19). For our example, we obtain a single template as shown in Figure 9g.

5.2 Building C-sets and E-sets

The concurrency- and exclusion-equivalence relations defined above share the following important property that lends itself to an efficient computation of these sets.

Lemma 1 (Disjointness) *Given a concurrency graph, a node may either be concurrency-equivalent to another node or be exclusion-equivalent to another node, or be neither. In particular, a node can never be part of both a C-set and an E-set.*

The proof of the above lemma is by contradiction. Let us suppose that a node A is concurrency-equivalent to a node B and exclusion-equivalent to a node C. The first relation directly implies that the edge (A,B) is present in the concurrency graph, while the second relation directly implies that the edge (A,C) is absent from the concurrency graph. The first relation also implies that the edge (B,C) is absent since A and B have similar neighbor relations. This implies that the edge (A,B) should be absent since A and C also have similar neighbor relations – a contradiction.

A direct consequence of this lemma is that the C-sets and E-sets are always mutually disjoint, which leads to a simple algorithm to construct C-sets and E-sets as shown in Figure 10. We go over each node in the graph once classifying it either into a C-set, an E-set or neither. The concurrency or exclusion equivalence check for each node can be performed quickly by employing the pigeon-hole principle. We simply hash each opgroup, using its set of neighbors in the concurrency matrix as the key. The neighbor relations are kept as a bit vector for speed. We hash in two ways, once by treating each opgroup as concurrent with itself to check whether it is equivalent to some C-set, and the second time by treating each opgroup as exclusive with itself to check whether it is equivalent to some E-set. By definition, opgroups hashing to the same bucket have the same concurrency neighbors and therefore become part of the same equivalent set. The final list of all distinct C-sets or E-sets is defined by all the distinct keys present in the hash map.

```
procedure FindCESets(BitMatrix concur)
1:  // “concur” is a (numNodes×numNodes) boolean matrix
2:  HashMap<BitVector, IntSet> CEmap;
3:  for (i = 0 to numNodes-1) do
4:     // Extract each node's vector of neighbors w/ and w/o self
5:     BitVector cKey = concur.row(i).set_bit(i) ;
6:     BitVector eKey = concur.row(i).reset_bit(i) ;
7:     // Check for existing C-set matching this node's key
8:     if (cKey is already present in CEmap) then
9:        Add node i to the C-set CEmap.value(cKey) ;
10:    // Check for existing E-set matching this node's key
11:    else if (eKey is already present in CEmap) then
12:       Add node i to the E-set CEmap.value(eKey) ;
13:    // If neither neighbor relation is present, start a singleton C-set and E-set
14:    else
15:       CEmap(cKey) = CEmap(eKey) = { i } ;
16:    endif
17: endfor
18: return list of C-sets and E-sets in CEmap with more than 1 member ;
```

Figure 10: Pseudo-Code for finding C-sets and E-sets.

An interesting observation about our algorithm is that when a node is initially added to the hash map, we need to start both a potential C-set and a potential E-set for it (Line 15). This is because, as a singleton, this node is not yet committed to participate in either one of them. Indeed, if no node in the graph is ever equivalenced with this node, it will remain as a singleton. However, if another node hashes to the same C-set key, a non-trivial C-set is defined between the two. Similarly, if another node hashes to the same E-set key, a non-trivial E-set is defined between the two. By the above lemma, only one of these sets may ever grow in membership, if at all. Therefore, we throw away all singleton sets and return only those with more than 1 member.

5.3 Building concurrency cliques

Finding all cliques of a graph is a well-known NP-complete problem [8]. Therefore, we use heuristics to enumerate them. Figure 11 shows our algorithm for finding all cliques in a graph.

The algorithm recursively finds all cliques of the graph starting from an initially empty current

```
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:      tryNodes = candidateNodes ;
9:      while (tryNodes is not empty)  do
10:        H1: if ((currentClique ∪ candidateNodes) ⊆ some previous clique) break ;
11:        H2: node = Pop(tryNodes) ;
12:        candidateNodes = candidateNodes - {node} ;
13:        if (currentClique ∪ {node} is not complete) continue ;
14:        H3: prunedNodes = candidateNodes ∩ Nhbrs(node) ;
15:        FindCliques(currentClique ∪ {node}, prunedNodes) ;
16:        H4: if (candidateNodes ⊆ Nhbrs(node)) break ;
17:        H5: if (this is first iteration) tryNodes = tryNodes - Nhbrs(node) ;
18:     endwhile
19:  endif
```

Figure 11: Pseudo-Code for finding cliques in a graph.

clique by adding one node at a time to it. The nodes are drawn from a pool of candidate nodes which initially 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 to grow the maximal cliques quickly and to avoid examining sub-maximal and previously examined cliques repeatedly [11]. 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. The second heuristic (H2) is that a candidate, once selected, is never considered again as a starting point for growing a clique. It is popped from the list of nodes to be tried as the starting node (Line 11). However, this node may still participate in a clique in subsequent iterations through a neighbor relation.

After selecting a candidate, we check to see if the selected candidate forms a complete graph with the current clique (Line 13). If so, we add it to the current clique and call the procedure recursively with the remaining candidates. The third heuristic used here (H3) 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 (H4), 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 (H5). 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 node that are to be dropped in each iteration.

5.4 Building the instruction format tree

Once the instruction templates are determined, all the information needed to build the IF-tree data structure is available. The top three levels consisting of the instruction, the templates, and the super groups are built to reflect the structure of the instruction template as determined above. The lower levels of the IF-tree consisting of the opgroups, the operation formats, and the IO-sets are determined from the input archspec as described already in archspec. Finally, the various instruction fields at the leaves of the tree are constructed by looking both at the contents of the various IO-sets in the input archspec and the individual control ports in the datapath that each field is supposed to control.

6 Setting up the resource allocation problem

6.1 Computing instruction field bit-requirements

As discussed in Section 4, each instruction field in the IF-tree reserves a certain number of instruction bits to control the corresponding datapath control port. The number of bits needed to encode the desired control information may, in general, depend on the following factors:

- the bit-width of the datapath control port, *i.e.*, the width of the decoded value,
- the encoding strategy for the various choices controlled by this field, *e.g.*, fixed-width *vs.* variable-width, and,
- the values of selector fields at higher-levels of the IF-tree lying on a path from the root of the IF-tree to the specific instruction field.

The register address and literal instruction fields usually have a fixed bit-width requirement as specified in the archspec and implemented in the datapath. It is possible, however, to define subsets of registers and literals that are accessible only in certain templates thereby reducing the bit-width requirements of their instruction fields and resulting in a shorter overall template. Such optimizations are discussed in Section 8.1.2.

The template, opgroup, and operation format select fields do not directly control a datapath control port. Instead, they simply provide control information to choose among the various templates, opgroups within each super group, and operation formats within each opgroup, respectively. It is possible to use a variable-width encoding for these fields in order to reduce the size of frequently occurring combination of operations. The opcode field may also be encoded with variable-width encoding, even though it controls a fixed-width opcode control port. In this case, the opcode may need to be decoded before being dispatched to the functional unit's opcode control port. This adds to the decode logic while reducing the size of the instruction format. Such optimizations and tradeoffs are discussed in Section 8.2.

The IO-set selection fields control the select inputs of the multiplexors and demultiplexors at the inputs and outputs of functional units, respectively. The choice of a template, opgroup or an operation format may restrict the choices that need to be encoded at this level. For example, a 4-input multiplexor at a functional unit input selects 1 out of 4 possible operand sources. But its allowable choices under a certain template may be restricted to be only 1 out of 2, in which case, the IO-set selection field for this input under this template needs to be only 1-bit wide. The width of the decoded value, however, remains at 2 bits since it drives the 2-bit multiplexor select port. The template select bits, together with the IO-set select bit, are used to generate the 2 bits required to control the corresponding select port.

6.2 Computing field conflicts

Before we can start allocating bit positions to various instruction fields we need to identify which fields are mutually exclusive and can be allocated overlapping bit positions and which fields need to be specified concurrently in an instruction and hence cannot overlap. Fields that are needed concurrently in an instruction are said to *conflict* with each other. Before we do bit allocation, we must compute the pairwise conflict relation between instruction fields, which we represent as an undirected conflict graph.

In the IF-tree, two leaf nodes (instruction fields) conflict if and only if their least-common ancestor is an AND-node. We compute pairwise conflict relation using a bottom-up data flow analysis of the IF-tree as shown in Figure 12. Our algorithm maintains a field set, F, and a conflict relation, C. Set F_n is the set of instruction fields in the subtree rooted at node n. Relation C_n is the conflict relation for the subtree rooted at node n. For the purpose of this analysis, an ANDOR-node in the

```
procedure ComputeConflicts(IFNode root, FieldSet F, ConflictRelation C)
    // dispatch on the basis of the node type
    case (root.nodeType()) of

    leaf-node:              // base case
       int n = root.nodeNumber() ;
       F = { n } ;
       C = φ ;

    OR-node:                // accumulate sub-tree specific fields and conflicts
       for (child ∈ root.children()) do
          FieldSet F_c = φ ;
          ConflictRelation C_c = φ ;
          ComputeConflicts(child, F_c, C_c) ;
          F = F ∪ F_c ;
          C = C ∪ C_c ;
       endfor

    AND-node:               // generate cross conflicts among sub-tree fields
       for (child ∈ root.children()) do
          FieldSet F_c = φ ;
          ConflictRelation C_c = φ ;
          ComputeConflicts(child, F_c, C_c) ;
          for (j ∈ F) do             // cross conflicts between all previous fields
             for (k ∈ F_c) do        // and fields in the current sub-tree
                C = C ∪ ⟨j, k⟩ ;
             endfor
          endfor
          F = F ∪ F_c ;              // accumulate sub-tree fields
          C = C ∪ C_c ;              // accumulate sub-tree specific conflicts
       endfor
    endcase
```

Figure 12: Pseudo-Code for computing field conflicts.

IF-tree is expanded into an AND-node consisting of the select field and a true OR-node consisting of the various subtrees in order to reflect its true meaning as discussed in Section 4.

The algorithm processes nodes in bottom-up order as follows. At a leaf node (Line 4), the field set is initialized to contain the leaf node, and the conflict relation is empty. At an OR-node (Line 9), the field set is the union of field sets for the node's children. Since an OR-node creates no new conflicts between children fields, the conflict set is the union of conflict sets for the node's children. Finally, at an AND-node (Line 18), the field set is the union of field sets for the node's children. An AND-node creates a new conflict between any pair of fields for which this node is the least-common ancestor; i.e. there is a new conflict between any two fields that come from distinct subtrees of the AND-node.

This algorithm can be implemented very efficiently by noting that the field sets are guaranteed to be disjoint coming from different sub-trees of the IF-tree. We can represent sets as linked lists, and perform each union in constant time by simply linking the children's lists (each union is charged to the child). Forming the cross-product conflicts between fields of distinct children of an AND-node can be done in time proportional to the number of conflicts. Since each conflict is considered only once, the total cost is equal to the total number of conflicts, which is at most n^2. For an IF-tree with n nodes and E field conflicts, the overall complexity is $O(n + E)$ time.

6.3 Assigning field affinities

The bit allocation algorithm described in the next section is also capable of aligning a set of non-conflicting instruction fields at the same bit position. This process is called **affinity allocation**. In order to make use of affinity allocation, we group instruction fields that point to the same datapath control port into a set called a **superfield**. All instruction fields within a superfield are guaranteed not to conflict with each other since they use the same hardware resource and therefore must be mutually exclusive. The bit allocation algorithm then tries to align instruction fields within the same superfield to the same bit position. Such alignment may simplify the multiplexing and decoding logic required to control the corresponding datapath control ports since the same instruction bits are used under different templates. On the other hand, such alignment may waste some bits in the template thereby increasing its width.

The superfield partitioning only identifies instruction fields that *may* share instruction bits. However, sometimes it is desirable that certain instruction fields *must* share the same bits. For example, if the address bits of a register read port are aligned to the same bit positions under all templates, then these address bits may be steered directly from the instruction register to the register file without requiring any control logic to select the right set of bits. At high clock rates, it is a bad idea to put a multiplexor in the critical path of reading operands out of a register file. To handle such a constraint, we also specify a subset of fields within a superfield that *must* share bits. This specification is in the form of a level mask that identifies the levels of the IF-tree below which all instruction fields that are in the same superfield must share bit positions. This mask is a parameter to the bit allocation algorithm described in the next section.

7 Resource allocation

Once the IF-tree and instruction field conflict graph are built, we are ready to allocate bit positions in the instruction format to instruction fields. In this problem, instruction fields are thought of as resource requesters. Bit positions in the instruction format are resources, which may be reused by mutually exclusive instruction fields. The resource allocation problem is to assign resources to requesters using a minimum number of resources, while guaranteeing that conflicting requesters are assigned different resources.

7.1 Resource allocation algorithm

Our allocation algorithm, shown in Figure 13, is a variant of Chaitin's graph coloring register allocation algorithm [4]. 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'.

```
procedure ResourceAlloc(IntVector Request, Graph conflicts)
  // compute resource request for each node + neighbors
  for each node n in conflict graph  do
     mark[n] = false ;
     AllocRes[n] = emptySet ;
     TotalRequest[n] = Request[n] + Request[neighbors of n] ;
  endfor
  // sort nodes by increasing remaining total resource request,
  // compute upper-bound on resources needed by allocation
  resNeeded = 0 ; pList = emptyList ;
  while unmarked nodes exist in conflict graph  do
     find unmarked node m such that TotalRequest[m] is minimum ;
     mark[m] = true ;
     pList.push(m) ;
     resNeeded = max(resNeeded, TotalRequest[m]) ;
     for each neighbor nhbr of m  do
        TotalRequest[nhbr] -= Request[m] ;
     endfor
  endfor
  // Adjust priority order of nodes as needed
  AdjustPriority(pList) ;
  // allocate nodes in priority order (e.g. decreasing total request)
  while pList not empty  do
     n = pList.pop() ;
     TotalRes = { 0 .. resNeeded-1 } ;
     // available bits are those not already allocated to any neighbor
     AvailRes[n] = TotalRes - AllocRes[neighbors of n] ;

     // select requested number of bits from available positions
     // according of one of several heuristics
     AllocRes[n] = select Request[n] bits from AvailRes[n] ;
           H1: affinity allocation
           H2: leftmost allocation
           H3: contiguous allocation
  endwhile
```

Figure 13: Pseudo-Code for resource allocation.

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 (Line 5).

The first loop shown in Figure 13 initializes the graph data structures and computes the total resource request for each node and its neighbors. Next, the algorithm repeatedly reduces the graph by selecting and eliminating the node with the current lowest total resource request. This is done in the second loop in Figure 13. At each reduction step, we keep track of the worst-case resource limit needed to guarantee a coloring. If the minimum total resources required exceeds the current value of k, we increase k so that the reduction process can continue (Line 14). The selected node is eliminated by subtracting its contribution from its neighbors' total resource request (Line 16).

In the original Chaitin's formulation, nodes are pushed onto a stack as they are removed from the graph in the order of increasing total request. In the allocation step, nodes are popped from the stack in the reverse order, i.e., in the order of decreasing total request in order to minimize the total number of resources allocated. In the context of bit allocation, this minimizes the width of the widest template. In our formulation, we permit slight adjustments to this priority order to satisfy additional requirements (Line 20). For example, we may give higher priority to nodes belonging to shorter templates to minimize their width as well.

The final step is to actually allocate the resources to the nodes in the desired priority order (third loop of Figure 13). At each iteration, a node is popped from the priority list and added to the graph of allocated nodes so that it conflicts with its neighbors that have already been allocated. The bits available for allocation to the current node are computed to be disjoint from bits assigned to the current node's neighbors (Line 26). Finally, the bits assigned to the current node are selected from those available using one or more heuristics described below.

7.2 Allocation heuristics

Affinity allocation (H1). As shown earlier in Section 6.3, non-conflicting instruction fields that control the same datapath control port may be grouped together to have *affinity*, implying that there is an advantage to assigning them the same bit positions. For example, consider two non-conflicting fields that drive the same register file read address port. By assigning the same set of bit positions to the two fields, we avoid multiplexing at the read address port, reducing the interconnect hardware as well as improving the critical path timing.

Each instruction field has a set of affinity siblings within the same superfield. During allocation, we attempt to allocate the same bit positions to all affinity siblings. We also take into account the subset of siblings that are required to share the same bits. This heuristic works as follows. When a node is first allocated, its allocation is also tentatively assigned to the node's affinity siblings within the same superfield. When a tentatively allocated node is processed, we make the tentative allocation permanent provided it does not conflict with the node's neighbors' allocations. Conflict may also occur if the tentatively allocated sibling needs more bits than the originally allocated node from the same superfield. If the tentative allocation fails, we allocate available bits to the current node using other heuristics, and we then attempt to re-allocate all previously allocated affinity siblings to make use of the current node's allocated bits. As yet unallocated siblings are also given this allocation, tentatively. Because nodes are processed in decreasing order of bits needed, tentative allocations often succeed.

When the tentative allocation fails for a field that is not necessarily required to share its bits with a previously allocated sibling, the algorithm may choose to retain the previous allocation for the sibling rather than reallocating it. This is a tradeoff between increasing the size of the template containing the sibling *vs.* reducing the overall hardware complexity. A useful criterion in making this decision is to differentiate among fields on the basis of their type (refer Section 4). One may choose to honor affinity constraints just for the instruction fields that control register file read address ports. This is because affinity allocation for these fields eliminates multiplexors from the operand fetch critical path without causing any reallocation because these fields have the same width. Another aspect that dramatically affects the result of affinity allocation is the priority order in which the instruction fields belonging to different sized templates are tried for allocation. One would want the fields belonging to shorter templates to be allocated before the fields for longer templates, otherwise affinity allocation among these fields may unnecessarily pull the fields in the shorter templates towards the right with wasted bits in the middle. This is an area of active research and more work is needed to use affinity allocation effectively.

Leftmost allocation (H2). The number of required bit positions computed during graph reduction is the number needed to *guarantee* an allocation. In practice, the final allocation often uses fewer bits. By allocating requested bits using the left-most available positions, we can often achieve a shorter, overall instruction format. Subject to the constraints of the heuristic above, this heuristic picks the lowest numbered bit positions that are available for a given request at Line 30. This causes the fields to be packed towards the left leaving out unused bits to the right.

Contiguous allocation (H3). Since bit positions requested by an instruction field generally flow to a common control point in the data path, we can simplify the interconnect layout by allocating requested bits to contiguous bit positions. However, this may increase the overall width of the template because a field that does not fit the leftmost contiguous set of available bits may have to be moved to a new position rather than split and overlap it partially with a shorter, mutually exclusive field that has been allocated already. Therefore, currently we use this heuristic only when a contiguous field may be found without increasing the overall width of the template.

8 Code size optimizations

Our instruction format design process, as described thus far, is focused primarily on designing an instruction format which reduces hardware complexity while complying with the concurrency requirements of the archspec. With a goal of minimizing the template select width and, hence, the decode complexity, the design process has been skewed towards minimizing the number of templates. In particular, we grouped the members of E-sets together into super groups, and picked the minimal set of templates that would provide the concurrency specified by the archspec. As we shall see below, this is at the expense of code size. In this section, we outline a number of instruction format optimizations which are, instead, aimed at reducing the code size. Since this is a topic of ongoing research[3], in this report, we shall only present the problem formulations, the solution approaches and certain bounds, leaving detailed discussions of the algorithms and proofs for a subsequent technical report.

As outlined in Section 3, there are three primary causes of wasted code size. These arise when there are frequent occurrences of:

- the explicit specification of no-op operations in one or more operation slots of an instruction,
- no-op instructions, i.e., an extreme case of the previous situation in which every operation slot of the instruction specifies a no-op operation, and
- the specification of operations that require considerably fewer bits than the width of the corresponding operation slot.

No-op instructions are dealt with quite successfully by the use of the previously described multi-noop capability. We explain the other two problems below, and describe their solutions in greater detail in Section 8.1.

Our initial design produces the set of minimal templates. In practice, this tends to produce a few long templates since the machines we are interested in have quite a bit of expressible instruction-

[3]At the time of writing this report, only one of these optimizations, custom templates, has been implemented in PICO.

level parallelism (ILP). But not all that parallelism is used at all times by the scheduler. This may be due either to the lack of ILP in the application or to the compiler's inability to exploit the ILP that is present. In either case, if we assemble programs using only the minimal templates, a lot of operation slots will end up specifying no-ops in the low ILP parts of the code, often leading to very large amounts of wasted code space.

Furthermore, the class of instruction formats that we entertain have the property that each operation slot in a template can, in general, specify one of many opgroups out of a super group. The operation slot is constrained to be as wide as the widest opgroup in the super group. Consequently, if there is a large difference between the widths of the narrowest and the widest opgroups in a super group, a large number of bits are wasted in every instruction which specifies an operation from the narrow opgroup. This creates a problem if the narrow opgroup contains frequently occurring operations.

This problem repeats itself at a lower level in the IF-tree. Each opgroup's width is determined by the widest operation format that it possesses, which in turn is determined by the widest operand specifier in each IO-set. In general, this is a potential problem at every OR-set level in the IF-tree except at the root level. Here, it is not an issue since variable-width templates are acceptable.

There are some secondary opportunities for saving code space that arise whenever the number of bits allocated for specifying an OR-set or an AND-list exceed the information content of that OR-set or AND-list. We discuss optimization techniques involving more efficient encodings that address this problem in Section 8.2. However, the number of bits that can be saved are, typically, rather small, and the increased decoding complexity could outweigh the benefits of the code size savings.

8.1 Transformations of the IF-tree

We first consider certain transformations on the IF-tree which lead to reduced code size at the expense of some increase in decode complexity. These transformations are guided by post-scheduling statistics gathered on the application program that indicate how often each member of an OR-set is selected and, for each AND-list, how often each combination of the members of its constituent OR-sets occurs. These transformations are performed prior to bit allocation. Thereafter, the instruction format design process proceeds as described previously.

8.1.1 The distribution transformation

All of the transformations upon the structure of the IF-tree that we shall consider are variations of a single transformation upon two contiguous levels in the IF-tree, where the higher level is an AND-list and the lower level is all of the OR-sets that are children of the AND-list. The AND-list under consideration can be either a subset or all of an AND-list in the IF-tree, i.e., either some or all of the OR-sets at the lower level may participate in the transformation. We refer to this transformation as the **distribution transformation** or just **distribution**. We first discuss the mechanics of two versions of this transformation, irredundant and redundant, and then consider various situations in which this transformation is beneficial.

Irredundant distribution

Common to both the redundant and irredundant versions of the transformation is an initial step that partitions each OR-set into one or more subsets. (At least one OR-set must be partitioned into multiple subsets, else this becomes a null transformation.) Consider the example of Figure 14a which consists of an AND-list of two OR-sets. Assume that each OR-set is partitioned into three subsets, as shown in Figure 14b. In the case of the irredundant transformation, we form the Cartesian product of the two partitioned OR-sets, and get an OR-set of nine AND-lists, as shown in Figure 14c. The original AND-list is replaced by this OR-set. This transformation is analogous to taking a Boolean expression, which is in the form of an AND of OR expressions, and distributing the AND operator across the OR operators–hence the name.

Redundant distribution

The redundant version of the transformation shares the same first step with its irredundant counterpart. However, instead of replacing the original AND-list with the Cartesian product set of

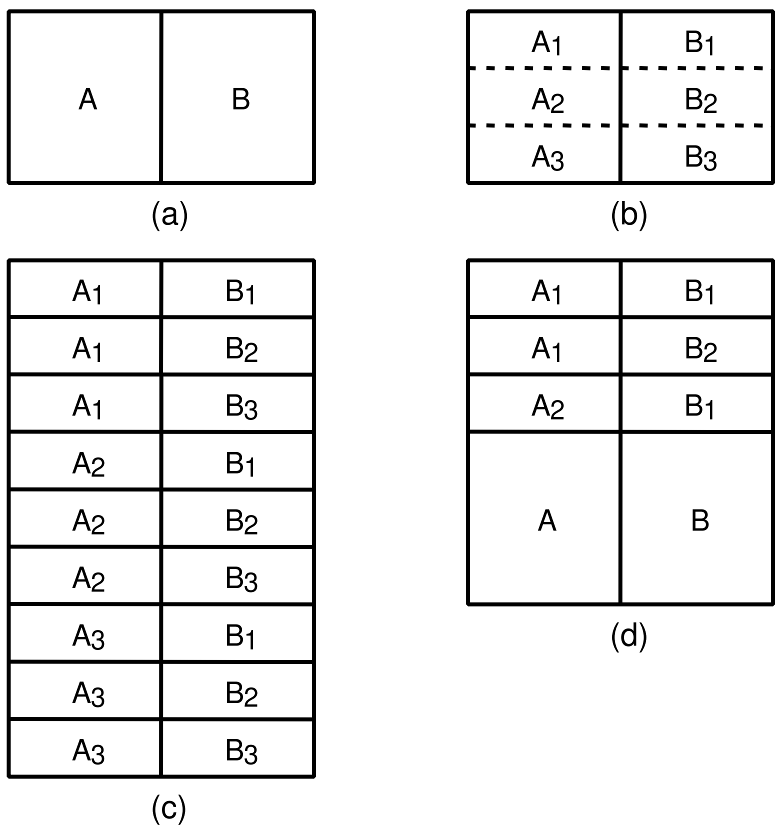


Figure 14: The distribution transformation. (a) An AND-list consisting of two OR-sets, A and B. (b) The two OR-sets after partitioning. (c) The nine irredundant AND-lists obtained by replacing the original AND-list with the Cartesian product of the two partitioned OR-sets. (d) The four redundant AND-lists obtained by augmenting the original AND-list with three of the nine AND-lists in (c).

AND-lists, it is instead augmented with some distinguished subset of the latter AND-lists. In the case of our example, the original AND-list is augmented with three of the nine AND-lists in Figure 14c to yield the OR-set of AND-lists in Figure 14d. This OR-set replaces the original AND-list in the IF-tree. The choice of these three AND-lists could, for instance, have been motivated by the fact that the remaining six have a low frequency of use. This transformation is redundant since any combination that can be specified using any of the three new AND-lists can also be specified using the original AND-list (which is still present).

The redundant transformation presumes two conditions. Firstly, the three new AND-lists must have some preferential properties with respect to code size which are not shared by the original AND-list. Otherwise, this transformation would be pointless. For instance, in our example, it could be that A1 and B1 require significantly narrower containers than do A and B, respectively, a virtue that is shared by the three new AND-lists in comparison to the original AND-list. In addition, these three AND-lists must be frequent for this code size benefit to be realized. Conversely, all of the remaining six AND-lists from the full Cartesian product set must either be infrequent or lack the preferential code size property. Were this not the case, they should have been added to the set of redundant, augmenting AND-lists. In the extreme case, if the full Cartesian product set of AND-lists is included, the original AND-list is completely redundant, and may be eliminated. This, of course, is the irredundant distribution.

In the event that the original AND-list was a member of an OR-set, rather than a subset of a larger AND-list, both versions of this transformation produce a (new) OR-set which is a member of another (pre-existing) OR-set. The new OR-set can be eliminated by making its members part of the higher level OR-set.

8.1.2 Strategies for re-structuring the IF-tree

One can devise a variety of IF-tree transformations based on how the OR-sets are partitioned, whether redundant or irredundant distribution is used and, if the former, which subset of the Cartesian product set of AND-lists is chosen to augment the original AND-list. Furthermore, distribution can be used at either one, or both, of the AND-list levels in the IF-tree: the template level or the operation format level.

Custom templates. The most important strategy for reducing code space wastage is to design a set of application-specific, custom templates; application-specific because they reflect the concurrency statistics of the scheduled application. One situation in which distribution is beneficial is when a frequently occurring opgroup is part of a super group that also contains other, significantly wider opgroups. As noted earlier, this is one of the primary causes of wasted code space. We can choose this frequent opgroup as the distinguished subset, apply redundant distribution and augment the original template with a new one which differs from the original one in only one operation slot. Instead of the ability to specify any operation from any opgroup in the super group, this template can only specify an operation from the distinguished opgroup. The operation slot in this template need only be as wide as the distinguished opgroup.

The extreme case of a narrow opgroup is the no-op opgroup. Each super group contains the no-op opgroup which consists of just one operation: the no-op. Since the no-op has no opcode and no operands, this opgroup is of zero width. When no other operation from this super group is to be issued, the no-op is specified, thereby wasting the full width of the container. If this is a frequent situation, the no-op opgroup can be selected as the distinguished opgroup, resulting in a new, augmenting template that has a zero-width container for the no-op opgroup, i.e., the no-op is implied by the template select code rather than being explicit.

This transformation need not be applied one super group at a time. One frequent opgroup can be selected as the distinguished opgroup from each super group in such a way that the combination of distinguished opgroups is frequent. (The distinguished opgroup could be the no-op opgroup.) Redundant distribution is used to yield a new, augmenting template which is only able to specify that particular combination of opgroups. This new template wastes no bits on explicit no-ops and no bits are wasted due to any of the operation slots being wider than the corresponding opgroup. This simultaneously addresses both of the primary causes of code space wastage.

This process may be repeated until it is rarely necessary either to use a template in which an explicit no-op has to be specified or to specify a narrow opgroup in a significantly wider container. (This is the strategy currently implemented in PICO.) Alternatively, we can do this in one step. We could identify, for each super group in the template, the frequent, subset of opgroups, all of which are narrower than the remaining members of the super group. This defines a distinguished subset of opgroups per super group, the Cartesian product of which yields a number of opgroup combinations. Those combinations that are frequent define new, augmenting templates.

Custom super groups. The above procedure could result in a very large number of custom templates. The reason is that each custom template consists of super groups, each of which contains just one opgroup. The number of new templates can be reduced by defining custom super groups. If two templates are identical in every operation slot but one, they can be combined into a single template if the differing opgroups are of approximately the same width. The only difference in this new template is that the two differing opgroups have been replaced by a custom super group that contains both opgroups. This substitution can be applied repeatedly to yield a smaller set of templates comprised of newly formed custom super groups.

Instead of creating a large number of templates and then having to re-combine them, it is preferable to create fewer templates in the first place. One can do this by defining the desired custom super groups before performing distribution. First, we partition the set of frequent opgroups in each super group into distinguished subsets of roughly the same width. (The infrequent opgroups are ignored.) These distinguished subsets constitute the custom super groups. Next, we form the Cartesian product of these custom super groups, and pick the frequent combinations as the custom templates.

Custom opgroups. An opgroup is an OR-set of operation formats, and just as a super group can waste code space when it contains opgroups of disparate width, so can an opgroup cause wastage when it has a frequent operation format that is significantly narrower than the widest format. In such cases, it is beneficial to define custom opgroups. This proceeds in much the same manner as for custom super groups. The frequent operation formats of each opgroup are grouped together into subsets of roughly equal width. These subsets constitute the new custom opgroups, and they get to participate in the definition of custom super groups which, in turn, define custom templates.

Custom operation formats and custom IO-sets. Operation formats are the second level of AND-lists in the IF-tree. One can, with benefit, apply distribution jointly to this level and the next lower level of OR-sets. For instance, consider an operation format for an opgroup that consists of dyadic operations. This AND-list consists of an opcode OR-set and a number of OR-sets each of which is an IO-set. The OR-sets of particular interest are the IO-sets for the source operands. For instance, assume that the source operands can each be either a 5-bit register specifier or a 10-bit literal, but that a register is specified more often then not. This creates an opportunity to save code space when registers, rather than literals, are specified.

As we did to create custom templates, we can first define custom IO-sets by grouping together frequently occurring operand specifiers into new, custom IO-sets. We can then form the Cartesian product of the custom IO-sets and, finally, pick the frequent combinations to get a set of custom operation formats for each opgroup. In our example, this would result in a new, shorter operation format where both source operands can only be registers. This new custom operation format is 10 bits narrower than the widest operation format (when both are literals). Since the narrow operation format is frequently used, these 10 bits are wasted quite often. This might argue for creating a new custom opgroup which only has this one narrow operation format.

Custom IF-subtrees. The customization procedure outlined above is fundamentally bottom-up, although we have described it in top-down order for clarity. The definition of custom IO-sets leads to custom operation formats, custom opgroups, custom super groups and custom templates-in that order. When applied systematically and comprehensively to the IF-tree in a bottom-up manner using redundant distribution at every step, this results in a custom, redundant IF-subtree replete with custom templates consisting of custom super groups, custom opgroups, custom operation formats, and custom IO-sets. In effect, this custom IF-subtree represents a subset instruction set architecture (ISA). Since it is based on program statistics, this is the portion of the overall ISA that is used predominantly, in a static sense, throughout the program. By using this subset ISA whenever possible, the size of the program is minimized.

Such subset ISAs are important when the program' s static and dynamic statistics are quite different. Most of a program' s execution time is typically spent in a very small fraction of the program. To achieve high performance, the processor must be capable of high levels of ILP, which implies wide minimal templates. But the vast majority of the code, which is infrequently executed, does not require high levels of ILP. Instead, compact code size is what is needed. The subset ISA makes this possible. Whereas the wide templates are used in the dynamically frequent portions of the program, the subset ISA is used in the remaining part of the statically frequent portions of the program.

As we have attempted to demonstrate, a wide variety of strategies can be employed in customizing the IF-tree based on the statistics of the scheduled application. We next describe what we currently do in PICO.

8.1.3 Our current procedure for designing custom templates

Our currently implemented procedure for designing custom templates corresponds to the simplest strategy described above–the definition of redundant templates in which each super group consists of a single opgroup, possibly the no-op opgroup. The space of possible custom templates of this sort is large; if there are N opgroups in the archspec, then there are 2^N possible combinations of these opgroups. Each combination defines one possible template. Hence, there are 2^N possible custom templates (less the minimal ones). The task is to pick the best ones.

Our approach is to identify the most frequently used combinations of opgroups that occur in the program and design shorter templates corresponding to them. Whereas these shorter templates have enough operation slots to accommodate the frequently occurring combinations of operations, they contain fewer operation slots than do the minimal templates. In particular, they do not contain some or all of the operation slots that would have specified a no-op. The no-op specification has become implicit and consumes no code space. The overall process consists of two passes: statistics gathering and custom template selection.

Statistics gathering

During the first pass, we define the instruction templates in accordance with the archspec, yielding the minimal templates. This first pass also produces the mdes that the compiler uses to produce a scheduled version of the application program. The exact schedule of the program can now be used to select the custom templates. For this purpose, we scan the scheduled code and generate a histogram of the combinations of opgroups that are scheduled as a single instruction. This is done by mapping the scheduled opcodes of an instruction back to their respective opgroups and counting the number of times that each combination of opgroups occurs. A static histogram records the frequency of static occurrences of each combination within the program and may be used to optimize the static code size. A dynamic histogram weights each opgroup combination with its dynamic execution frequency and may be used to improve the instruction cache performance by giving preference to the most frequently executed sections of the code. Currently, we use the static histogram in our optimization to give preference to the overall static code size.

Custom template selection

This histogram is used during a second pass of instruction format design. We start with the previously defined minimal templates. The histogram statistics are used to select a few new opgroup combinations, defining the custom templates, that are subsets of one or more minimal templates. We formulate the task of selecting the best combinations as an optimization problem to be described shortly. These custom templates are in addition to the set of minimal templates which must be retained to cover all possible concurrency relations of the machine as specified by the arch-spec. Together, they constitute the final set of templates which then go through the process of bit allocation to yield the final instruction format.

The additional templates are narrower than the minimal templates, but they increase the size of the template selection field and, hence, the decode logic (and, to a small extent, the code size). The other significant increase in decode cost is due to the fact that the same operation may now be represented in different positions in the instruction format and, as a consequence, the instruction bits from these positions will have to be multiplexed based on the template selected. (This cost may be partially or completely eliminated by performing affinity allocation as discussed earlier in Section 7.2.)

The selection of the custom templates may be stated as an optimization problem as follows. Given a budget of k custom templates, we need to identify a set of templates, $\mathcal{T}$, which minimize the total program size, W, given by

$$W = \sum_{i=1}^{m} f_i \times w(N(\mathcal{T}, C_i))$$

where:

- $T_1, .., T_n$ are the minimal templates,
- $T_{n+1}, .., T_{n+k}$ are the k custom templates selected from the set of all possible templates,
- $\mathcal{T}$ is the union of the set of minimal templates and the set of custom templates,
- C_1, .., C_m are all of the distinct opgroup combinations that are found to occur, as single instructions, in the scheduled code,
- $f_1, .., f_m$ are their static frequencies of occurrence,
- $N(\mathcal{T}, C_i)$ is the narrowest template in $\mathcal{T}$ that can be used to specify C_i, and
- $w(T_i)$ is the width of template T_i.

This optimization problem is NP-complete. In practice, heuristic optimization techniques must be employed. Our algorithm shown in Figure 15 is a simple and greedy solution to the above optimization problem. Its description involves the following additional definitions:

- $\mathcal{T}$ is re-defined as the current set of templates at any point during the selection process, consisting of the minimal templates plus any custom templates that have been selected up to that point.
- $v(C_i)$ is a lower bound on the width of the template defined by C_i and is given by the sum of the number of bits used by each opgroup in the combination C_i, where the width of an opgroup is given by its width in the minimal templates. (Note that this does not include super group select bits, template select bits, the EOP bit and the multi-noop bits.)

```
procedure SelectCustomTemplates(TemplateSet templates, CombinationList stats, int k)
    // Initialize the width lower bound for initial templates.
    for (each template T in templates) do
        T.lbwidth = sum of widths of T's operation slots ;
    endfor
    // We pick k most beneficial combinations as custom templates.
    repeat k times
        Template maxBenefitTemplate ;
        int maxBenefit = 0 ;
        for (each opgroup combination C_j in stats) do
            // Compute the benefit of adding C_j as a custom template
            Template T_{C_j} = Template(C_j) ;
            T_{C_j}.lbwidth = v(C_j) ;                    // initialize width lower bound
            int benefit = 0 ;
            for (each opgroup combination C_i in stats) do
                int freq = C_i.frequency ;
                Template old = NarrowestTemplate(templates, C_i) ;
                Template new = NarrowestTemplate(templates∪{T_{C_j}}, C_i) ;
                benefit = benefit + freq * (old.lbwidth - new.lbwidth) ;
            endfor
            if (maxBenefit < benefit) then
                maxBenefit = benefit ;
                maxBenefitTemplate = C_j ;          // record template with maximum benefit
            endif
        endfor
        templates = templates ∪ maxBenefitTemplate ;
    endrepeat
```

Figure 15: Pseudo-Code for custom template selection.

At any point during the template selection process, the lower bound, V, on the size of the program, with the current set of templates $\mathcal{T}$, is given by

$$V = \sum_{i=1}^{m} f_i \times v(N(\mathcal{T}, C_i))$$

If a single, additional custom template, T_{C_j}, corresponding to an opgroup combination, C_j, is added to $\mathcal{T}$, the size of the program will decrease because T_{C_j}, rather than some superset template, will be used to encode C_j as well as other subsets of C_j. The amount of this reduction, i.e., the benefit of including the custom template corresponding to C_j is given by

$$b_j = \sum_{i=1}^{m} f_i \times \Big[v(N(\mathcal{T}, C_i)) - v(N(\mathcal{T} \cup \{T_{C_j}\}, C_i)) \Big]$$

If D is the set of all opgroup combinations that use this new template, *i.e.*, $N(\mathcal{T} \cup \{T_{C_j}\}, C_i)) = T_{C_j}$ for all $C_i \in D$, then b_j can be expressed as

$$b_j = \sum_{C_i \in D} f_i \times [v(N(\mathcal{T}, C_i)) - v(C_j)]$$

After computing the benefit of the custom template corresponding to each opgroup combination, the combination C_i with the largest benefit is selected, and the corresponding template is added to $\mathcal{T}$. In general, the addition of this template will reduce the benefit of certain other candidate templates. Therefore, we recompute the benefits b_j in the context of the new set of templates before selecting the next custom template as shown in Figure 15.

Ideally, one would repeat this process, greedily selecting a custom template on each iteration, until the marginal code size cost of encoding additional templates and their decoding cost outweighs the marginal code size savings. Since the decoding cost is not quantified in our benefit expression, our strategy (refer Figure 15) is to iterate k times, where k is a parameter (with a default value of 7) that constitutes the budget for the number of custom templates[4]. The results presented in [17] show the effectiveness of this strategy in reducing program code size.

8.2 Efficient encoding of select fields

We now look at a set of code size optimizations which, by and large, do not entail the restructuring of the IF-tree. Rather, they revolve around techniques for encoding the select field for an OR-set so as to achieve high encoding efficiency.

[4]In fact, our current implementation is even simpler. We evaluate the benefit of all the opgroup combinations just once, and then pick the k opgroup combinations with the highest benefit.

8.2.1 Variable-width encoding of the select field for OR-sets

An OR-set is a mutually exclusive set of items. The encoding of an item has two parts. One is the content field which specifies the content of the item (e.g., an opcode specifier at one level in the IF-tree, or an opgroup specifier at a higher level). The second part is the select field which indicates which particular item from the OR-set is being specified. The default strategy for encoding the select field is to provide a fixed-width field of width $\lceil \log_2 N \rceil$, where N is the number of items in the OR-set. The overall item width is the sum of the widths of the content and select fields.

The container for an OR-set is the set of bits allocated to encode the OR-set. A property of the instruction formats designed by PICO is that, except for the OR-set at the root level of the IF-tree, the container for an OR-set must be wide enough to accommodate the widest item. If w_{max} is the width of the widest content field across all of the items, the container is designed to be $w_{max} + \lceil \log_2 N \rceil$ bits wide, by default. If there is a high variance in the content field widths of the items, however, a variable-width encoding of the select field can reduce the *maximum* item width and hence the width of the container. On the other hand, at the root level of the IF-tree where variable-width items (templates) are already permissible, a variable-width encoding of the select field can reduce the *average* item width, given a high variance in the frequency of occurrence of the items.

Of course, there is a tradeoff between the code size reduction achieved by variable-width encoding and the resulting increase in decode complexity. The number of input bits to the select field decode PLA is equal to the width of the widest select field, which will be greater than the $\lceil \log_2 N \rceil$ bits of input for a fixed-width encoding.

The setup for the bit allocation step must deal with an additional detail. With variable-width encoding, the select field assumes some small number of distinct widths. For instance, let's say that it can assume three distinct widths x, y and z, where $x < y < z$. The select field is partitioned into three segments: the first one is the first x bits, the second one is the next $y-x$ bits and the third one is the last $z-y$ bits. Each segment is treated as a separate field for bit allocation purposes. The select field for an item in the OR-set will, in general, occupy the first n segments. The content field, and all of its sub-fields, conflict with these n segments, but not with the remainder, as far as allocation is concerned. Thereafter, bit allocation proceeds as usual.

We now consider in greater detail the two flavors of variable-width encoding optimization: minimization of the maximum item width and minimization of the average item width.

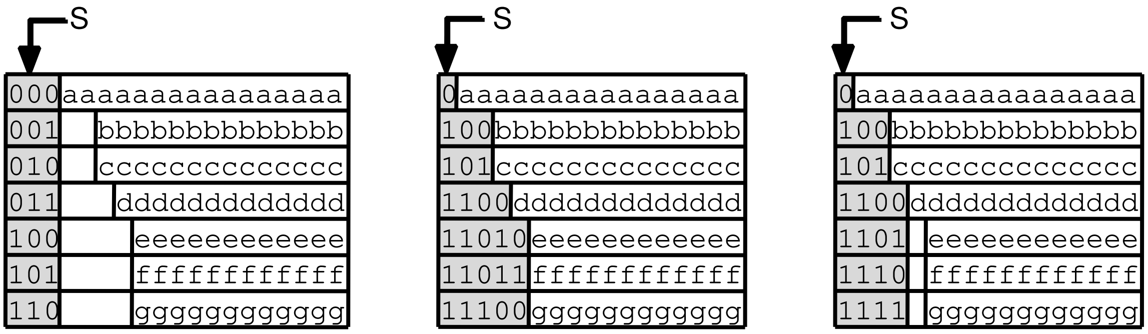
Minimizing the width of the container

Consider an OR-set in which the items' content widths are quite disparate. Using a variable-width encoding, we can allocate short select codes to items having the greatest content width, while using longer select codes for items having smaller content width. As a result, the width of the widest item can be reduced.

For instance, the OR-set in Figure 16a consists of 7 items whose content fields require from 10 to 16 bits. With fixed-length select codes, we require an additional 3 bits to encode the select field, S, bringing the overall size of the widest item and, hence, the container to 19 bits. On the other hand, the variable-width encoding of the select field used in Figure 16b reduces the width of the widest item, and the width of the container, to 17 bits. However, the widest select field is now 5 bits versus 3 bits with fixed-width encoding. Since the complexity of the decode logic for the select field is determined by the width of the widest select field, it is desirable that this maximum width be minimized. The variable-width encoding in Figure 16c does so, reducing the widest select field to 4 bits without increasing the width of the widest item above 17 bits.

The statement of the problem is as follows: Given an OR-set, design a variable-width encoding for the select field that, as a primary objective, minimizes the width of the widest item (select field plus content field) and which, as a secondary objective, minimizes the width of the widest select field.

In general, if an OR-set has N items, and w_i is the content width of the i-th item, the total number of distinct alternatives that can be specified by the OR-set is given by $\sum_{i=1}^{N} 2^{w_i}$ since the i-th item can specify 2^{w_i} alternatives. Thus a lower bound, W, on the container width is given by



(a) (b) (c)

Figure 16: Encoding strategies for the select field, S, of an OR-set. (a) The fixed-width encoding for the selector field yields a container width of 19 bits. (b) The variable-width encoding results in a container width of 17 bits, and a maximum select field width of 5 bits. (c) A further optimization reduces the maximum select field width to 4 bits without increasing the container width.

$$W = \left\lceil \log_2 \sum_{i=1}^{N} 2^{w_i} \right\rceil$$

We will limit our discussion to only valid encodings for the select field. A **valid encoding** is one in which the (narrower) select code for one item is never the prefix of the (wider) select code for another item. Such codes are also known as **prefix codes** [5]. Valid encoding simplifies the decoding by recognizing narrow select codes uniquely without looking at additional bits.

To be optimal, a variable-width encoding of the select field must be such that the select field width, s_i, for the i-th item is no more than $(W - w_i)$ bits. We use the following lemma, which we state without proof, to show that such an optimal variable-width encoding always exists[5].

Lemma 2 (Variable-width Encoding) *Let s_i be the width of the select field for the i-th item in an OR-set of N items. This set of select field widths possesses a variable-width encoding of the select field if and only if*

$$\sum_{i=1}^{N} 2^{-s_i} \leq 1$$

[5]Even though this lemma is introduced in the context of minimizing the maximum item width, it applies to the variable-width encoding of a select field in any context.

Furthermore, for every choice of select field widths that satisfy the above condition, and only for those that do, it is always possible to define a select code for each item, that fits within the width of the corresponding select field, such that the set of select codes together constitute a valid encoding.

The intuitive explanation for the above lemma is that a select field of width s_i bits uses up 2^{-s_i} fraction of the encoding space. Now, if $s_i = W - w_i$, where W is as defined above, then

$$\sum_{i=1}^{N} 2^{-s_i} = \sum_{i=1}^{N} 2^{w_i - W} = \frac{\sum_{i=1}^{N} 2^{w_i}}{2^W} \leq 1, \text{ since } 2^W \geq \sum_{i=1}^{N} 2^{w_i}$$

Thus, a valid, optimal, variable-width encoding always exists when the select field widths are selected in this manner.

Our process for designing optimal variable-width encodings consists of two algorithms. The first one minimizes the width of the widest select field without increasing the width of the container beyond W bits. In general, if $s_i = W - w_i$, the condition of the lemma, when evaluated, yields a strict inequality. This is the case with our example in Figure 16b. Our algorithm takes advantage of this situation and reduces the width of the widest select field until the condition of the lemma evaluates to an equality. Thereby, within the constraint of a container width of W, the algorithm minimizes the maximum value of s_i, across all the items, without violating the condition imposed by the lemma. This is the case in Figure 16c. The second algorithm takes a set of select field widths that satisfy the lemma's condition, and generates a valid set of select codes of the specified widths. A detailed description of these algorithms is beyond the scope of this report.

If w_{max} is the widest content field across all of the items, W must necessarily be at least $(w_{max}+1)$. As we observed earlier, the container width with fixed-width encoding would be $(w_{max} + \lceil log_2 N \rceil)$ bits, where N is the number of items in the OR-set. Thus, an upper bound on the savings from the use of variable-width encoding of the select field is $(\lceil log_2 N \rceil - 1)$ bits. In the above example, this upper bound is achieved.

Minimizing the average width of the container

Instead of minimizing the width of the widest item in an OR-set, one could choose to minimize the average width of an item. Since the content width is fixed, this amounts to minimizing the average width of the select field. When the frequencies of the items are quite disparate, well known techniques such as Huffman coding [5] can be employed to minimize the average width of the select field by allocating short select codes to frequent items and long select codes to the infrequent ones.

In general, for an OR-set with N items, the frequency of whose i-th item is given by f_i, a lower bound on the average number of bits needed by the select field when using a frequency-based, variable-width encoding of the select field is given by its information theoretic entropy [13]

$$-\sum_{i=1}^{N} p_i \log_2 p_i, \text{ where } p_i = f_i/F \text{ and } F = \sum_{i=1}^{N} f_i$$

Here, p_i is the empirical probability of the i-th item, and F is the total number of instances of the items (*e.g.*, the total number of instructions if the OR-set is the set of instruction templates). Consequently, an upper bound on the savings that can be achieved is given by

$$F \times \left(\lceil \log_2 N \rceil + \sum_{i=1}^{N} p_i \log_2 p_i \right)$$

For any OR-set containing more than one item, an upper bound on the savings, across all possible frequency statistics, is

$$F \times (\lceil \log_2 N \rceil - 1)$$

since the select field of even the most frequent item must be at least one bit wide. This upper bound is approached as p_i approaches 1 for some one item.

In the context of the IF-tree, however, this optimization has limited applicability except at the root level. For OR-sets at lower levels, the size of the container is determined by the width of the widest item. A consequence of Huffman coding is that infrequent items will get select codes that are longer than $\lceil \log_2 N \rceil$ bits. Should items with greater content width happen to be infrequent, Huffman coding will actually increase the size of the container.

Nevertheless, Huffman coding can be valuable in encoding the template select field in order to minimize the average width of the template select field, thereby minimizing the average width of an instruction and, thus, the size of the program. However, the standard Huffman coding algorithm will not give the best results; a variable-width, frequency-based encoding algorithm is required which takes into account the quantized nature of the instructions. Due to rounding the instruction templates up to the next multiple of the quantum, each template will have some number of unused bits which are ”free” as far as the select field is concerned. Should a high frequency template happen to have a large number of unused bits, it might make sense to give this template a longer select code than the Huffman algorithm would have provided. As a result, some other less frequent template, which has very few unused bits can be given a select code that is shorter than what its Huffman code would have been.

8.2.2 Efficient joint encoding of the select fields in AND-lists

Given an AND-list whose OR-sets do not utilize their encoding space fully, one can design a joint encoding of those OR-sets that is more efficient. (Note that the AND-list in question can either be one item in a higher level OR-set, or it can be a subset of an AND-list.)

We shall illustrate the problem and two solution strategies using the example of Figure 17 which shows an AND-list consisting of three OR-sets, each of which contains five items. (These three OR-sets could, for example, be the register specifiers for a three-operand operation format, and the AND-list shown in Figure 17a would then represent a subset of the operation format AND-list.) The content fields are not shown in Figure 17; only the select fields are. The content fields may be non-existent if, for instance, these OR-sets correspond to the opcode or register specifier fields. Even if the content fields exist, they are irrelevant to this optimization and are ignored.

Using the most obvious encoding, this would require 3 bits per OR-set, for a total of 9 bits, as

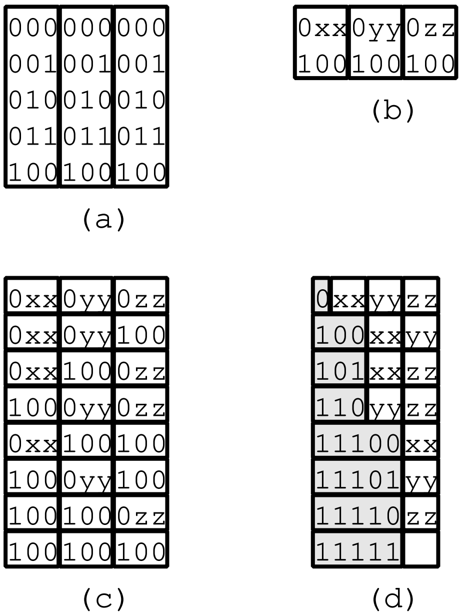


Figure 17: Partitioning of OR-sets into power-of-two (POT) blocks. (a) An AND-list consisting of three OR-sets with five items per OR-set. Only the selector fields for the OR-sets are shown. A total of 9 bits are needed to represent this AND-list. (b) Each OR-set after partitioning into two POT blocks, one with four items and the other with one. (c) The resulting eight AND-lists after taking the Cartesian product of the POT blocks corresponding to each OR-set. (d) An encoding for the eight AND-lists using a variable-width encoding for the selector field. Unvarying bits of the original selector fields are implicit based on the selector field. Only 7 bits are now needed to represent these eight AND-lists.

shown in Figure 17a. On the other hand, since this AND-list specifies one of $5^3 = 125$ possible combinations of register specifiers, $\lceil \log_2 125 \rceil = 7$ bits should be enough. The fact that each OR-set is inefficiently encoded, using 3 bits to encode just 5 items, results in an overall wastage of 2 bits. What we wish to do is to come up with a joint encoding of the three select fields which avoids this wastage. This joint encoding, when decoded, should yield the original select fields of Figure 17a, after which they can be used as needed.

One strategy is to use irredundant distribution to replace the AND-list by a single OR-set of 125 items. This would yield an efficient encoding requiring 7 bits, but the decoding would be relatively complicated; since each decoded register specifier is 3 bits and there are three register specifiers, a decode PLA with 7-inputs and 9-outputs is needed as shown in Figure 18a.

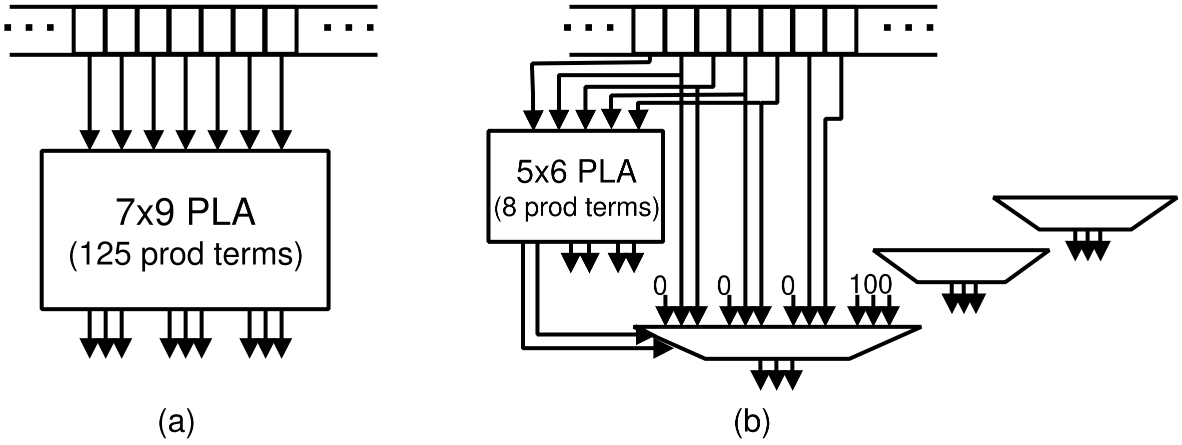


Figure 18: Decoding hardware needed for joint encoding of select fields in AND-lists. (a) Direct decoding of 125 joint OR-set items requiring one 7-input, 9-output PLA. (b) POT-block decoding requiring one 5-input, 6-output PLA and three 4:1 multiplexors.

The second strategy reduces the decode complexity by first partitioning each OR-set into two subsets, as shown in Figure 17b. The first subset consists of the four specifiers whose high-order bit is 0. The second subset consists of the singleton specifier "100". The key property of these subsets is that the cardinality of each is a power of 2. Hence, we refer to them as power-of-two blocks or POT-blocks. The strategy is to partition each OR-set into the smallest possible number of maximal POT-blocks. After this is done, each OR-set in our example consists of just two items whose content widths are unequal. For the first POT-block in each OR-set, since it is known that the high-order bit is 0, only 2 bits are needed (shown as "xx", "yy" and "zz", respectively) to specify the register unambiguously. Likewise, for the second POT-block, 0 bits are needed; once the select field specifies this item, all three bits of the register specifier are known to be "100".

Irredundant distribution can now be applied to the AND-list of Figure 17b, which gets replaced by the single OR-set of Figure 17c. The Cartesian product of the three OR-sets of Figure 17b, each of which has two items, yields an OR-set consisting of 8 items, each of which is a 3-tuple of register specifiers. These 8 items have unequal lengths. One item is of length 6, three are of length 4, three are of length 2, and one is of length 0. The use of a variable-width select field, as shown in

Figure 17d, results in an efficient encoding with a container width of 7 bits. This is the narrowest possible encoding given that there are 125 different possible combinations of register specifiers.

The advantage of this scheme is reduced decode complexity. For each register specifier, the 3 bits come from one of four places. When it corresponds to the first POT-block, the two lower-order bits can come, in general from one of three positions in the container, as illustrated by Figure 17d. (The high-order bit is identically "0".) In the case of the second POT-block, the entire specifier is identically "100". Thus, a 4:1 multiplexer is required as shown in Figure 18b, which in turn requires 2 steering bits that must come from the select field decoder. Since each register specifier needs two such bits, and since the maximum width of the select field is 5 bits, the decode PLA will have 5 inputs and 6 outputs.

Although this decode PLA is smaller than the previous one with 7-inputs and 9-outputs, this strategy does require the three multiplexers as well. The relative merits of the two encoding strategies must be evaluated on a case by case basis. Both encoding strategies are identical with respect to the resulting code size.

An upper bound on the savings is one bit less than the number of OR-sets in the AND-list. To see this, consider an AND-list of N OR-sets, and suppose that the original, fixed-width encoding for the i-th select field is w_i bits wide. Therefore, the minimum possible number of items in the i-th OR-set is $2^{w_i-1}+1$, and the minimum possible cardinality of the Cartesian product of the N OR-sets is given by

$$\prod_{i=1}^{N}\left(2^{w_i-1}+1\right) > \prod_{i=1}^{N} 2^{w_i-1} = 2^{\sum_{i=1}^{N}(w_i-1)}$$

Consequently, a lower bound on the number of bits needed to jointly encode the N select fields is

$$1+\sum_{i=1}^{N}(w_i-1) = \sum_{i=1}^{N} w_i - (N-1)$$

for a maximum savings of $N-1$ bits. This upper bound on the savings is achieved in our example.

This optimization, which is motivated by the inefficient utilization of the encoding space, can be viewed as a combination of three steps: the partitioning of OR-sets into POT-blocks, the irredundant distribution of the AND-list over the OR-sets, followed by a minimal container width encoding.

8.2.3 Merits of variable-width encoding

The savings, due to jointly encoding the select fields of an AND-list, are typically quite limited. At best, the savings are one less bit than the number of OR-sets. The savings, due to variable-width encoding of the select field of an OR-set, are more attractive. Under the best of circumstances, the impact of the select field on the container width or on the average item width can be reduced to one bit. Aggregated over the entire instruction, these savings could be significant. However, these savings are at the cost of increased decode PLA complexity.

There is one situation in which these savings could be extremely important. The instruction packet-the unit of access from the instruction cache-has to be at least as wide as the widest instruction template to ensure that an instruction can be issued every cycle. It is often further required that it be rounded up to the next power-of-two bytes in width. If the widest template happens to be just a few bits wider than a power of two, the instruction packet is approximately doubled. Since the data paths and the storage elements between the instruction cache and the instruction register must all be as wide as the instruction packet, this can have a major impact on the cost of the hardware. Under such circumstances, the use of variable-width encoding, to bring the width of the widest template down below the power to two boundary, can be entirely worthwhile.

9 Machine-description driven assembly

Our overall objective is to enable widespread, on-demand design and use of customized architectures that provide better performance at a lower cost for a given application. However, such customizations and design trade-offs are tedious, at best, to apply manually to a given design, let alone to a series of designs in a design space exploration. Therefore, an important requirement of fulfilling our objective is a capability to automatically retarget the entire software tool chain, in-

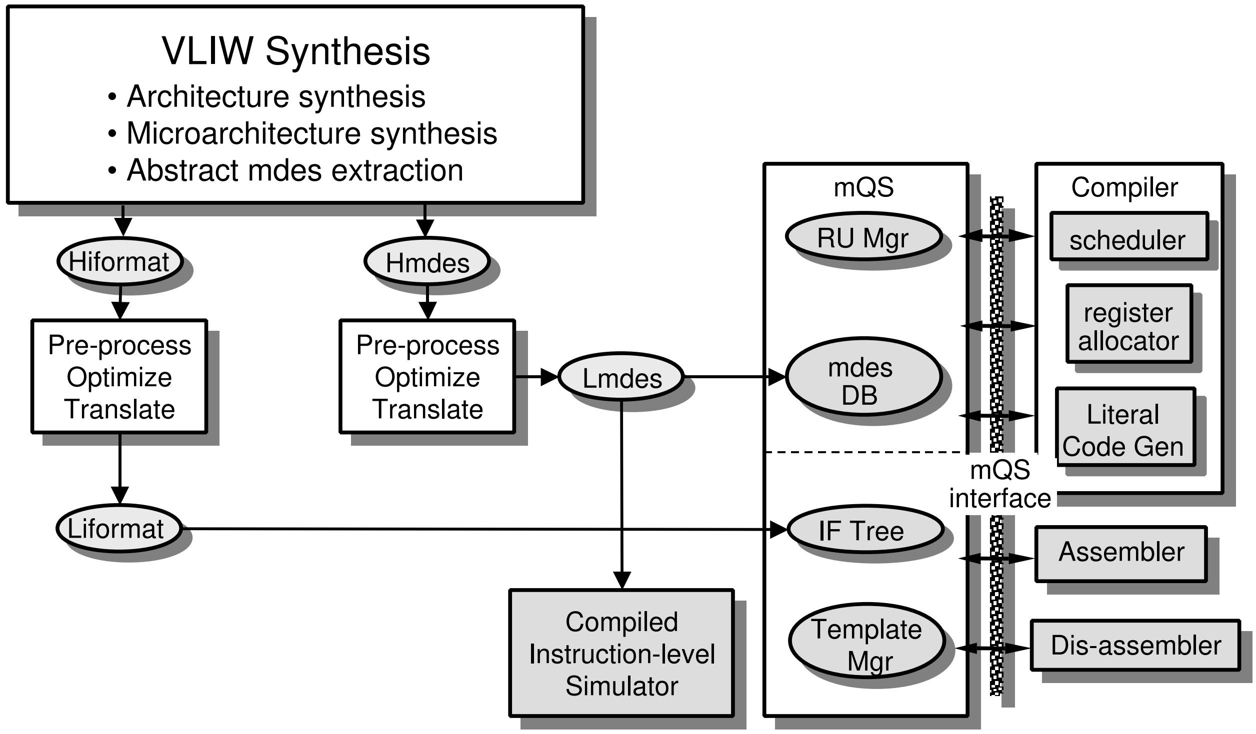


Figure 19: A retargetable tool chain automatically driven by the information produced by the VLIW synthesis subsystem.

cluding the compiler, assembler, disassembler, simulator etc. to an arbitrary machine architecture within the design space. The PICO-VLIW system is designed to allow such retargeting automatically by separating the architecture-dependent information from the software tools that make use of that information in a table-driven manner. In this section, we describe PICO's tool chain retargeting technology in the context of designing custom VLIW instruction formats.

The various components of our automatically retargetable tool chain are shown in Figure 19. The VLIW Synthesis subsystem produces an mdes and an instruction format corresponding to each processor configuration it designs. These data structures encapsulate all the essential information needed by the software tools in order to retarget them to the new architecture. These data structures abstract away from a detailed hardware description of the processor and present a tool-centric view of the machine. The interface to the various tools is managed by an **mdes query system** (**mQS**) that can satisfy a fixed set of queries regarding the processor. The mQS is initialized with the machine-specific mdes and instruction format data structures to fulfill these queries on behalf of the various tools. In this way, the tools themselves are kept machine-independent and easily retargetable.

We have shown previously [1, 16] how to design mdes-driven compilers and the mQS interface for VLIW architectures primarily supporting the instruction scheduler and the register allocator modules. In this report, we describe the additional mechanisms and queries needed to allow the assembler[6] and the disassembler to be retargeted to different machine instruction formats. As shown in Figure 19, the extension to the mQS consists of two components, the IF-tree data structure described earlier and a **template manager**. The IF-tree contains all the machine-specific instruction format information that is needed by the assembler and the disassembler. The template manager manages this machine-specific information and provides a fixed set of queries to be used during assembly and disassembly. It also carries and manipulates internal state representing the partial state of assembling or disassembling an instruction.

Before delving deeper into the process of assembly and linking using the mQS, we first discuss some important issues in the design of retargetable assemblers for VLIW architectures. Next, we describe the the mQS interface queries used by the assembler, followed by a description of the data structures and mechanisms used by the mQS during the process of assembly and disassembly.

9.1 Issues in assembler design

A VLIW assembler is provided with the scheduled program consisting of a sequence of sets of architectural operations. Each set consists of the operations scheduled on the same cycle. The job of the assembler is to encode each set of operations into a single instruction and consequently the entire program into a single stream of instruction and data bits. In order to make the assembler easily retargetable to different architectures, the instruction format dependent aspects of the assembly process are factored out into the mQS interface which is described later. The assembler still retains many important architecture-independent responsibilities including code layout and in-

[6] In all our discussion, we use the term *assembly* to denote the combined processes of instruction assembly and program linking.

struction template selection policies. Below, we briefly discuss the relevant issues that differ from a traditional design.

9.1.1 Code layout

As described in the previous sections, our system permits multiple instruction templates with possibly different widths. Furthermore, not all operations within a template may have the same width. This implies that the width of each instruction and consequently its address offset within a procedure can not be determined simply by counting the number of operations. A consequence of this is that the resolution of symbolic addresses within the program including forward branch offsets and data and procedure references becomes a two-pass process. In the first pass, each instruction in a procedure is assigned an address offset by selecting an instruction template for it and possibly aligning it to some address boundary. In the second pass, each instruction is assembled into the template selected for it while resolving symbolic addresses to actual addresses assigned earlier.

9.1.2 Template selection and assembly

The assembler does not have any direct knowledge of the instruction format of the machine. Instead, it makes queries to the template manager within the mQS to identify the available choices for templates that may encode a given set of operations. The template selection policy, however, is controlled by the assembler. The choice of a template may either depend on its encoding properties (e.g. its length, the number of multi-noops available etc.) or the assembly context (e.g. whether a subset architecture mode is set). Our current policy, which is geared towards reducing the overall code size, is to choose the shortest template that may encode a given set of operations, and in case of a tie, the one with the maximum number of allowable multi-noop cycles. If the chosen template can not encode all the multi-noop cycles that are needed, the next instruction is chosen to be the shortest template, which will consist entirely of no-ops, that can encode all the remaining no-op cycles.

Given an appropriate template, the assembler needs to fill it with the proper encoding of the given set of co-scheduled operations. However, the assembler does not have any knowledge of the template structure either. This format-specific information is kept within the mQS and managed by the template manager. Therefore, the assembler provides the opcode and operand information for each of the co-scheduled operations to the template manager, which assembles the instruction on assembler's behalf using the selected template. This division ensures that the assembler need not know anything about the target architecture and its instruction format, it only needs to understand the structure of the program that is supplied to it and make policy decisions regarding its encoding.

9.1.3 Branch target alignment

The assembler is also responsible for making branch target alignment decisions while laying out the code in memory. Enforcing all branch targets to be aligned to packet boundaries may blow up the code size by a large amount due to wasted bits. On the other hand, for frequently visited branch targets outside the sequential flow of control, the accumulated stall penalty due to the target instruction not being contained within one instruction packet may be substantial. Therefore, we use a profile driven heuristic to ensure that the most frequently executed targets do not cross a packet boundary. This eliminates the branch penalty for such targets requiring minimal increase in code size.

In this approach, the compiler needs to profile the program and annotate the assembly code with the frequency with which each branch target is visited via a taken branch. Before starting the actual process of assembly, the assembler first sorts all branch targets from the highest to the lowest dynamic frequency and then classifies them one-by-one as not permitted to cross a packet boundary (non-crossing) by keeping track of two cumulative values: dynamic fraction of targets already classified as non-crossing (initially 0.0) and static fraction of targets that may potentially cross a packet boundary (initially 1.0). At the point when the dynamic fraction of non-crossing targets is equal or larger to the static fraction of potentially crossing targets, the process stops and all the remaining unprocessed targets are classified as potentially crossing a packet boundary. Next, the assembler assembles the code assigning addresses to each instruction in sequential order. If a branch target instruction that has been classified as non-crossing is found to cross a packet boundary, it is aligned to the next packet boundary and the EOP bit is set in the preceding instruction.

This heuristic essentially tries to achieve an even balance between dynamic fraction of non-crossing targets (causing increase in code size) and static fraction of potentially crossing targets (causing branch penalty) to effectively trade off branch penalty and code size. The results presented in [17] show the effectiveness of this heuristic.

9.2 The query interface

The set of mQS interface functions related to the process of assembly and disassembly can be classified into the following categories:

Template selection - These functions enable the selection of the most appropriate instruction format template for a given set of architectural operations scheduled in the same cycle.

Template assembly - These functions help to assemble a single VLIW instruction by filling the operation information provided by the assembler into the template and then returning the bit encoding of the fully assembled instruction.

Template disassembly - These functions help to disassemble an instruction byte stream, identifying the set of operations scheduled within one instruction.

Miscellaneous instruction format information - These functions provide general information regarding the instruction format of the machine.

We describe the functions provided in each of these categories.

9.2.1 Template selection queries

```
void set_operation_tuple(List<opcode, ioformat>);
templateid get_next_template();
void set_template(templateid);
```

For each set of co-scheduled operations to be assembled, the assembler uses the first function to provide their opcodes and the operation formats to the template manager. The template manager uses this information to identify all templates that can encode the given set of operations. An instruction filled with only no-ops is specified as the empty operation tuple. We assume that the choice of a template to encode a given set of operations may depend only on the opcodes and the operation formats and not, for example, on the specific register number, or the literal value that needs to be encoded. The second function behaves like an iterator providing an integer handle to the next available template that can encode the previously identified set of co-scheduled operations. If there is no next template, it returns -1. The last function instructs the template manager to use the specified template to encode the current set of co-scheduled operations.

```
int get_template_size(templateid);
int get_template_unused_bits(templateid);
int get_template_max_multi_noops(templateid);
```

This set of queries help to select the best template from the ones that can encode the given set of co-scheduled operations. The first query returns the size of the given template in bits. This may be used to find the shortest template that would encode the current set of operations. The second query returns the number of the unused bits in the given template after or during the process of assembling a set of operations into it. This may be used to select a template based on the static bit utilization. The third query returns the maximum number of multi-noop cycles that may be accommodated within the current template. This may be used to select a template based on whether it is able to encode the desired number of multi-noop cycles that follow the current instruction.

9.2.2 Template assembly queries

```
void assemble_op(opcode);
void assemble_pred(regnum);
void assemble_pred_lit(bool);
```

```
void assemble_src(regnum);
void assemble_src_lit(int);
void assemble_dest(regnum);
```

For each operation in the set of co-scheduled operations, the assembler repeatedly calls the above functions to provide its opcode and operand information to the template manager. The opcodes and the operands must be provided in the same order as that specified during template selection and must also correspond to the operation format specified at that time. The template manager consults the IF-tree to identify the instruction field, its bit position, and the encoding corresponding to each operation component and sets the appropriate bits in the selected template.

```
void assemble_multi_noop(int);
void assemble_EOP(bool);
```

The assembler uses the above functions to set template-wide properties. The first function is used to encode the number of cycles following the current instruction filled completely with no-ops. The second function is used to set or reset the EOP bit of the current template which identifies the fact whether the next instruction is or is not aligned to the packet boundary respectively.

```
Bitvector get_instruction();
```

Finally, the above function returns the completed bit encoding of the fully assembled instruction to the assembler.

9.2.3 Template disassembly queries

The mQS interface for the disassembler is the reverse of that for the assembler. Essentially, the job of the disassembler is to decode the instruction in software in much the same way as the instruction decode logic does it in hardware. Since the architecture may contain variable length instruction templates, the width of the current instruction and hence the position of the next instruction are determined only after identifying the current instruction template which is then decoded to obtain the set of co-scheduled operations within it. The following mQS queries support this process:

```
templateid set_instruction(Bitvector);
```

This function takes a stream of instruction bits (equal to the maximum size instruction) and identifies the template that it belongs to. The template manager records the bits internally for subsequent decoding. The returned templateid may be used by the disassembler to determine the size of the template which identifies the starting position of the next instruction in sequence.

```
List<opcode, ioformat> get_operation_tuple();
```

This query returns the opcodes and the operation formats corresponding to the set of co-scheduled operations in the current instruction. These are used to identify the various components of the instruction using the following queries.

```
void disassemble_op(opcode);
regnum disassemble_pred();
bool disassemble_pred_lit();
regnum disassemble_src();
int disassemble_src_lit();
regnum disassemble_dest();
```

The first query indicates to the mQS which operation out of the list returned above is to be disassembled next. The remaining queries are used to disassemble the operands of that operation in the order specified by the operation format. The operation format identifies whether a register or a literal is specified for predicates and data sources.

```
int disassemble_multi_noop();
bool disassemble_EOP();
```

The queries determine the template-wide properties regarding the number of multi-noop cycles encoded within the template and whether or not the EOP bit is set.

9.2.4 Miscellaneous queries

The following queries are used to obtain miscellaneous information about the instruction format design of the given architecture.

```
int get_packet_size();
int get_quantum_size();
int get_max_inst_size();
```

The first function is used to obtain the size of an instruction packet in bits that identifies the start of the next instruction when the EOP bit is set in the current instruction. The second function returns the quantum size used during instruction format design. The third function returns the size of the maximum sized instruction template in bits which is the size of the bitvector passed to the template manager to disassemble.

9.3 mQS data structures and mechanisms

In order to support the above queries, several architecture-dependent data structures are maintained within the mQS by the template manager. Below, we describe these data structures together with the various mechanisms that the template manager uses in order to implement the above mQS interface.

9.3.1 Template selection support

The template manager needs to find all instruction templates that can encode the given set of architectural operations characterized by their opcode and operation format combinations. The number of such combinations is, however, extremely large. This is because there may be a large number of architectural opcodes supported by an architecture when counting all the variations for data widths, resource reservations, latencies and compare conditions. Furthermore, each opcode may support many different operation formats including various kinds of addressing modes and

```
procedure FindValidTemplates(List<opcode, ioformat>, operationList)
  TemplateSet templates = AllTemplates ;
  for (each (opcode, ioformat) in operationList) do
    OpGroupSet opgroups = OpcodeMap.value(opcode) ;
    TemplateSet opTemplates = emptySet ;
    for (each opgroup in opgroups) do
      if (Match(opgroup.ioformat, ioformat)) then
        opTemplates = opTemplates ∪ opgroup.templates ;
      endif
    endfor
    templates = templates ∩ opTemplates ;
  endfor
  return templates ;
```

Figure 20: Pseudo-code to find the set of valid templates that may encode a given set of operations.

register files for each of its operands. Finally, an instruction is characterized by a set of such opcodes and operation formats which is a cross product of all these possibilities. Therefore, rather than enumerating all possible sets of operations and the templates that may encode them, the template manager uses some auxiliary data structures and a matching algorithm to find the valid templates.

The template manager keeps a hash table called the **opcode map** which maps each architectural opcode to the set of opgroups (kept as a bitvector) in which that opcode may appear. Different opgroups in this set may implement either disjoint or overlapping operation formats for the given opcode. An opcode may end up belonging to multiple opgroups when a new combination of opcodes or certain operation format combinations of existing opcodes are separated into a new opgroup for template customization.

Another data structure kept by the template manager is the **opgroup table** that lists all opgroups and the various operation formats they support. The size of this table is proportional to the number of opgroups supported by the architecture which is usually even smaller than the number of architectural opcodes. Each opgroup in the opgroup table also keeps a set of templates (also kept as a bitvector) in which it can be encoded. This set is directly derived from the instruction format design process by identifying the opgroups that are present in a template.

Figure 20 shows the pseudo-code for the process of identifying all valid templates that can encode a given instruction within the template manager. This process is initiated when the assembler provides the opcode and operation format pairs characterizing the current instruction to the template manager using the function `set_operation_tuple`. The first step is to map each supplied opcode from this pair to the set of opgroups it belongs using the opcode map. The next step is to filter this set for valid opgroups that can implement the current opcode and operation format pair. This is done by matching the supplied operation format with the ones allowed for that opcode within each opgroup. If more than one opgroup matches, then each is considered to be a candidate for encoding that opcode and operation format pair. A union of the set of templates that may encode a candidate opgroup gives the set of all templates that may encode the given operation. Finally, the intersection of such sets across all operations gives the set of valid templates that may encode all the operations of the current instruction.

Any one of the templates found in the above process may be selected by the assembler to encode the given instruction. Therefore, various properties of the templates need to be recorded in order for the assembler to identify the best template amongst the available choices. These properties are also kept within the template manager in a **template table**. Such properties include the size of the template in bits, the number of multi-noops cycles that can be encoded within it, and the number of unused bits within the template due to encoded no-ops as well as affinity allocation of fields.

9.3.2 Template assembly and disassembly support

The assembly and disassembly processes require complete knowledge about the bit positions of all instruction fields and their encodings within each template that is selected. The hierarchical syntax of the instruction format is completely captured in the IF-tree data structure as shown in Figure 6. After bit-allocation, the exact bit positions allocated to each instruction field are annotated back into the IF-tree. The various selector fields (e.g. template selector field, opgroup selector field, opcode, etc.) also keep the exact encoding of each branch of the sub-tree that they select. All this information is supplied to the mQS by the instruction format design process.

The template manager uses the IF-tree not only to identify the instruction syntax and bit positions of various fields, but also to keep track of the partial state of assembling or disassembling an instruction. This can be easily accomplished by keeping a pointer at each level of the IF-tree to identify the sub-tree under which an instruction field is being assembled. This permits a hierarchical, iterative interface to be built to direct the template manager through the various steps of assembly and disassembly.

Template assembly. The process of assembly proceeds as follows. First, the template manager allocates a bitvector wide enough to encode the template chosen by the assembler and sets the template selector field bits to indicate that choice. The choice of the template specifies the set of operation groups encoded by it and the bit positions occupied by them by virtue of identifying the template sub-tree within the IF-tree. Next, the assembler directs the template manager to encode each operation of the current instruction one-by-one in the context of this sub-tree using the interface shown above.

The encoding of an operation is initiated by the function `assemble_op` which identifies the operation being assembled and sets the opgroup, opcode and the operation format selector field bits. For this task, the template manager consults the opcode map which provides the set of all opgroups that may encode the given opcode. From this set, the opgroup belonging to the current template is identified by iterating over the children sub-trees of the current template. The operation format corresponding to the opcode supplied at the time of template selection is used to identify the operation format selection bits. Subsequent calls to assemble the various operands using the interface functions shown above obtain their bit positions and encodings from the selected opgroup and IO-set sub-tree of the IF-tree. Finally, the EOP bit and the multi-noop cycles are filled in as needed and the finished string of bits representing the instruction are returned to the assembler by using the function `get_instruction`.

Template disassembly. The process of disassembly proceeds in the reverse manner of assembly. First, the disassembler supplies a string of bits from the instruction stream of size equal to the maximum size instruction using the function `set_instruction`. The instruction to be disassembled is positioned left-justified in this string. The template manager decodes the template select field and the EOP bit to identify the template and its size. The disassembler uses this information to determine the start of the next instruction as follows. If the EOP bit is set (function `disassemble_EOP` returns true) then the disassembler skips to the next packet boundary, otherwise it queries the size of the current template (function `get_template_size`) to determine the start of the next instruction.

Next, the disassembler identifies the various operations encoded within the current instruction using the function `get_operation_tuple` which returns a list of opcodes and their operation formats encoded within the template. This information is determined by the template manager by identifying the bit positions of the various selector fields and decoding their value using the IF-tree. The disassembler then uses this list to disassemble each operation one-by-one by specifying the opcode to the template manager using the function `disassemble_op` and then querying its operands in the order specified by the operation format previously identified for that operation. Each such query returns the actual register number or literal value encoded within the operation. The template manager disregards any extra bits in the supplied string that are not decoded as part of the current instruction.

9.3.3 Miscellaneous data structures

Aside from the above data, the template manager may keep some general information about the instruction format design such as the size of the instruction quantum and the size of the instruction packet fetched from the instruction cache which may be used to decide whether to align an instruction to the packet boundary or not. Another piece of information is the size of the longest template which is the number of bits that the disassembler must provide to the template manager to decode.

10 External uses of the instruction format

10.1 External file format

In order to decouple the assembler from instruction format design, the information contained within the IF-tree is output to an external file at the end of the design process. This file is then read in by the mQS as shown pictorially in Figure 19. The file format uses the same database language HMDES Version 2 [9] as used for specifying the archspec.

As an example, the external specification of the IF-tree for the processor specified in Appendix A is shown in Appendix C. At present, we only emit the levels of the IF-tree that help in template selection and code size estimation. This includes from the top of the IF-tree–instruction, instruction templates, operation slots, opgroups and their opcodes. In addition, miscellaneous control fields and select fields corresponding to the ANDOR-nodes are also emitted. Each template, slot or opgroup identifies the number of bits it takes to encode it. This is used to estimate the code size of the assembled program as well as the benefit of adding new combinations of opgroups as custom templates within the processor.

10.2 Syntax-directed instruction decoding table generation

The IF-tree data structure directly defines the decoding structure of an instruction and may be used to automatically generate the instruction decode tables for a given datapath. These tables directly express the relationship between the bits of the instruction register and the bits needed to be supplied to the various control ports of the datapath.

In the PICO system, these decode tables are generated in the form of a PLA specification. However, the actual control hardware may be implemented using a PLA, a ROM, or direct random logic using existing control logic optimization and synthesis tools.

PICO generates the following two kinds of decode tables:

Template decode table – The control logic in this table identifies the EOP bit and the width of the current template by decoding the template selection field. This information is used by the instruction pipeline control unit to identify the start of the next instruction in sequence.

Functional unit decode table – One decode table is generated per functional unit which represents the control logic responsible for identifying the operation to be issued to the given functional unit within the current instruction. There can be at most one such operation in the current instruction. If there is no operation in any operation slot of the current instruction that is geared towards the given functional unit, then this logic issues a no-op to the unit.

For a recognized operation on a unit, this logic decodes the IO-set select fields in the instruction and generates control information for the operand multiplexors and demultiplexors at the inputs and outputs of that functional unit, respectively. This logic is also responsible for decoding any immediate literals encoded in the operation slot that are to be used by the functional unit.

As an example, the template decode table and shift unit decode table for the processor specified in Appendix A are shown in Appendix D.

10.3 Architecture manual

Another module of the PICO system, the report generator, uses the current design of the processor to generate an architectural report. This report documents, among other things, the instruction format design including all the allowable instruction templates, their various operation slot and opgroup combinations, the various instruction fields contained within each template and the bit positions allocated to them. This information may be used for assembly-level programming and code-generation.

As an example, the instruction format section of the architecture report of the processor specified in Appendix A is shown in Appendix B. The first sub-section describes the number and overall structure of each of the instruction templates in terms of the various opgroups. There are 8 templates in this example: the first (T0) is the minimal template which covers the full parallelism of the machine as prescribed by the archspec. The remaining templates (T1-T7) are custom templates based on the compilation of the jpeg application for this processor. Note that each operation slot in the minimal template (T0) consists of a set of opgroups (super group) while that in the custom templates point to a single opgroup.

The second sub-section identifies the unique operation formats applicable to each of the opgroups. A table of operand choices and bit requirements is generated corresponding to each operation format. Choices for an operand represent either connections to a register file or immediate literals and are preceded by selector field encodings.

The final sub-section identifies the exact bit allocation of fields within each opgroup for each template that it occurs in. For opgroups occurring under multiple templates, one may see the effect of affinity allocation which attempts to assign the same, even if discontiguous, bit positions to the various fields of the opgroups under those templates.

11 Conclusion

PICO-VLIW is an architecture synthesis system for automatically designing the architecture and microarchitecture of VLIW and EPIC processors. It has been operational as a research prototype since late 1997. Starting from an abstract specification of the instruction-set architecture, PICO-VLIW automatically generates

- the concrete instruction-set architecture for the processor, including the opcode repertoire and instruction format,
- the detailed microarchitecture consisting of the execution and instruction unit datapaths along with a specification of the control tables for the latter, and
- a machine description (including the instruction format) for use by our retargetable compiler, assembler and simulator.

Our focus in this report has been on one aspect of PICO-VLIW, which is its ability to design variable-width, multi-template instruction formats that minimize code size. By using such formats, we are able to accommodate the widest instructions where necessary, while employing compact, restricted instructions for much of the application program where the amount of parallelism is insufficient.

In this report, we described the various steps involved in the instruction format design process including the data structures and the algorithms used by PICO-VLIW during each step. The design process is driven by an abstract ISA specification, the archspec, and a description of the processor datapath. This is in contrast to traditional, manual design flows, in which the concrete ISA is the input specification from which the processor datapath is derived. PICO-VLIW uses the archspec to automatically select a set of minimal instruction templates that are sufficient to exploit the full ILP of the processor, along with a set of application-specific templates customized to the needs of a given application. The system also generates the exact bit layout of the instruction templates, optimizing them for reduced size as well as reduced controlpath complexity. Since the instruction format is designed with the hardware rather than a human programmer in mind, it has the unusual property that the instruction fields of an operation and, for that matter, the bits of an instruction field need not be contiguous.

The class of instruction formats generated by PICO-VLIW incorporate a variety of techniques to contain code size even for extremely wide-issue and deeply-pipelined processors. Custom templates reduce the code size for a given processor width, while the multi-noop capability reduces code size by an amount proportional to the operation latency. In contrast, affinity allocation attempts to trade-off some of the reduction in code size for reduced controlpath complexity arising from custom templates. Likewise, judicious use of the EOP bit during assembly reduces the run-time stall penalty for fetching branch target instructions with minor increase in code size. The effectiveness of all these techniques has been measured in a recent study [17]. Using the above techniques, the study reports the code size expansion relative to an abstract, sequential CISC processor to be between 1.5x and 2.3x for a 4-issue VLIW processor and between 1.6x and 2.3x for a 12-issue processor even with 3x the normal latencies. This increase is comparable to that for a

RISC processor.

In this report, we also described the structure of a machine-description driven assembler for EPIC and VLIW processors. Such an assembler is an essential part of a system for exploring the space of processors and finding the good designs. The assembler is written with no in-built assumptions regarding the instruction format of the processor. Instead, it uses a finite and well-defined set of queries to access the services of the mdes Query System, which is an active database that holds all of the necessary information regarding the processor. Consequently, such an assembler concerns itself only with the policies and heuristics for generating compact code (in addition, of course, to the other, conventional tasks of an assembler).

We would like to thank Mike Schlansker for suggesting the use of multiple templates as a means of compressing out no-ops from a canonical instruction format. We would also like to thank Scott Mahlke for helping in the design and a preliminary implementation of the mQS interface for the assembler.

References

[1] Shail Aditya, Vinod Kathail, and B. Ramakrishna Rau. Elcor's machine description system: Version 3.0. Technical Report HPL-98-128, Hewlett-Packard Laboratories, October 1998.

[2] Shail Aditya and B. Ramakrishna Rau. Automatic architectural synthesis and compiler retargeting for VLIW and EPIC processors. Technical Report HPL-1999-93, Hewlett-Packard Laboratories, 1999.

[3] G. R. Beck, W. L. Yen, and T. L. Anderson. The Cydra 5 mini-supercomputer: architecture and implementation. *Journal of Supercomputing*, 7(1/2):143–180, May 1993.

[4] 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.

[5] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge, MA, 1990.

[6] Henk Corporaal and Reinoud Lamberts. TTA Processor Synthesis. In *First Annual Conf. of ASCI*, Heijen, The Netherlands, May 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, P. Russo, and S. Devadas. A Methodology for Accurate Performance Evaluation in Architecture Exploration. In *Design Automation Conference*, New Orleans, LA, June 1999.

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

[12] 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.

[13] Robert J. McEliece. *The Theory of Information and Coding*. Encyclopedia of Mathematics and its Applications. Cambridge University Press, Cambridge, 1984.

[14] B. Ramakrishna Rau Michael S. Schlansker. EPIC: An architecture for instruction-level parallel processors. Technical Report HPL-1999-111, Hewlett-Packard Laboratories, January 2000.

[15] B. R. Rau. Cydra 5 Directed Dataflow architecture. In *Proceedings of Compcon Spring 88*, pages 106–113, San Francisco, California, February 29–March 4, 1988.

[16] B. Ramakrishna Rau, Vinod Kathail, and Shail Aditya. Machine-description driven compilers for EPIC and VLIW processors. *Design Automation for Embedded Systems*, 4:71–118, 1999.

[17] B. Ramakrishna Rau Shail Aditya, Scott A. Mahlke. Code size minimization and retargetable assembly for custom EPIC and VLIW instruction formats. *ACM Transactions on Design Automation of Electronic Systems, special issue on SCOPES '99*, 5(4), October 2000. To appear.

[18] Vinod Kathail Shail Aditya, B. Ramakrishna Rau. ”automatic architecture synthesis of VLIW and EPIC processors”. In *Proceedings of the 12th International Symposium on System Synthesis, San Jose, California*, pages 107–113, November 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 Instruction Format Design

B.1 Instruction Issue Templates

EOP$< 3 >$ T0$< 0 \cdots 2 >$

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

EOP$< 3 >$ T1$< 0 \cdots 2 >$

VOG_o1c1i0	VOG_o52c3i0

EOP$< 3 >$ T2$< 0 \cdots 2 >$

VOG_o1c1i0	VOG_o60c3i0

EOP$< 3 >$ T3$< 0 \cdots 2 >$

VOG_o1c1i0

EOP$< 3 >$ T4$< 0 \cdots 2 >$

VOG_o21c1i0

EOP$< 3 >$ T5$< 0 \cdots 2 >$

VOG_o6c1i0

EOP$< 3 >$ T6$< 0 \cdots 2 >$ $\boxed{\text{VOG\_o18c1i0}}$

EOP$< 3 >$ T7$< 0 \cdots 2 >$ $\boxed{\text{VOG\_o2c1i0}}$

B.2 Operation 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 01 10	gpr cr s	00 01 10	gpr cr s	0 1	gpr cr

Opset: EOS_intarith2_intdiv

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

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

B.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	$< 4, 5 >$	$< 6 \cdots 13 >< 18 \cdots 22 >< 24 \cdots 29 >< 58 \cdots 70 >$	$< 14 \cdots 17 >$

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

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

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

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

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

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

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

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

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

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

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

Template	OPCODE	SRC1	DEST1
T0	$< 4 >$	$< 6 \cdots 11 >$	$< 14 \cdots 19 >$

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

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

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

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

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

Template	OPCODE	SRC1	DEST1
T0	$< 74 >$	$< 76 \cdots 81 >$	$< 82 \cdots 87 >$

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

Template	SRC1	DEST1
T0	$< 76 \cdots 81 >$	$< 74, 75 >< 82 \cdots 85 >$

VOG_o34c2i0 : (Opset: EOS_convif_S_float)

IO descriptor: pr ? gpr : fpr

Template	SRC1	DEST1
T0	$< 74 \cdots 79 >$	$< 82 \cdots 87 >$

VOG_o31c2i0 : (Opset: EOS_floatarith2_D_float)

IO descriptor: pr ? fpr , fpr : fpr

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

VOG_o27c2i0 : (Opset: EOS_floatarith2_S_floatdiv)

IO descriptor: pr ? fpr , fpr : fpr

Template	SRC1	SRC2	DEST1
T0	$< 76 \cdots 81 >$	$< 88 \cdots 93 >$	$< 82 \cdots 87 >$

VOG_o28c2i0 : (Opset: EOS_floatarith2_S_floatmpy)

IO descriptor: pr ? fpr , fpr : fpr

Template	SRC1	SRC2	DEST1
T0	$< 76 \cdots 81 >$	$< 88 \cdots 93 >$	$< 82 \cdots 87 >$

VOG_o56c3i0 : (Opset: EOS_floatload_load1)

IO descriptor: pr ? gpr : fpr

Template	OPCODE	SRC1	DEST1
T0	$< 42 >$	$< 34 \cdots 39 >$	$< 43 \cdots 47 >< 97 >$

VOG_o62c3i0 : (Opset: EOS_floatstore_store)

IO descriptor: pr ? gpr , fpr :

Template	OPCODE	SRC1	SRC2
T0	$< 42 >$	$< 34 \cdots 39 >$	$< 43 \cdots 47 >< 97 >$

VOG_o52c3i0 : (Opset: EOS_intload_load1)

IO descriptor: pr ? gpr : gpr cr btr

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

VOG_o60c3i0 : (Opset: EOS_intstore_store)

IO descriptor: pr ? gpr , gpr cr btr s :

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

VOG_o1c1i1 : (Opset: EOS_intarith2_int)

IO descriptor: pr ? gpr cr s , gpr cr s : gpr cr

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

VOG_o2c1i1 : (Opset: EOS_intarith2_intshift)

IO descriptor: pr ? gpr cr s , gpr cr s : gpr cr

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

C External Representation of Instruction Format

```
$include "VLIW_family.hmdes2"

SECTION IF_Opgroups
{
VOG_o79c4i0 (ops(
BRCT.0
BRCF.0
));
VOG_o80c4i0 (ops(
BRL.1
));
VOG_o78c4i0 (ops(
BRU.2
RTS.2
));
VOG_o22c1i0 (ops(
MOVELB.3
MOVELBX.3
MOVELBS.3
));
VOG_o38c2i0 (ops(
CONVSD.4
CONVDS.4
));
VOG_o37c2i0 (ops(
CONVDW.5
));
VOG_o34c2i0 (ops(
CONVWS.6
));
VOG_o31c2i0 (ops(
FADD_D.7
FMAX_D.7
FMIN_D.7
FSUB_D.7
));
VOG_o27c2i0 (ops(
FDIV_S.8
));
VOG_o28c2i0 (ops(
FMPY_S.9
));
VOG_o56c3i0 (ops(
FL_S_C1_C1.10
FL_D_C1_C1.10
));
VOG_o62c3i0 (ops(
```

```
FS_S_C1.11
FS_D_C1.11
));
VOG_o1c1i0 (ops(
ADD_W.12
ADDL_W.12
SH1ADDL_W.12
SH2ADDL_W.12
SH3ADDL_W.12
AND_W.12
ANDCM_W.12
NAND_W.12
NOR_W.12
OR_W.12
ORCM_W.12
SUB_W.12
SUBL_W.12
XOR_W.12
XORCM_W.12
MIN_W.12
MAX_W.12
));
VOG_o1c1i1 (ops(
ADD_W.13
ADDL_W.13
SH1ADDL_W.13
SH2ADDL_W.13
SH3ADDL_W.13
AND_W.13
ANDCM_W.13
NAND_W.13
NOR_W.13
OR_W.13
ORCM_W.13
SUB_W.13
SUBL_W.13
XOR_W.13
XORCM_W.13
MIN_W.13
MAX_W.13
));
VOG_o3c1i0 (ops(
DIV_W.14
DIVL_W.14
REM_W.14
REML_W.14
));
VOG_o4c1i0 (ops(
MPY_W.15
MPYL_W.15
));
VOG_o2c1i0 (ops(
SHL_W.16
SHR_W.16
SHLA_W.16
SHRA_W.16
));
VOG_o2c1i1 (ops(
SHL_W.17
SHR_W.17
SHLA_W.17
SHRA_W.17
));
VOG_o18c1i0 (ops(
```

```
CMPP_W_FALSE_UN_UN.18
CMPP_W_FALSE_UN_UC.18
CMPP_W_FALSE_UC_UN.18
CMPP_W_FALSE_UC_UC.18
CMPP_W_TRUE_UN_UN.18
CMPP_W_TRUE_UN_UC.18
CMPP_W_TRUE_UC_UN.18
CMPP_W_TRUE_UC_UC.18
CMPP_W_EQ_UN_UN.18
CMPP_W_EQ_UN_UC.18
CMPP_W_EQ_UC_UN.18
CMPP_W_EQ_UC_UC.18
CMPP_W_NEQ_UN_UN.18
CMPP_W_NEQ_UN_UC.18
CMPP_W_NEQ_UC_UN.18
CMPP_W_NEQ_UC_UC.18
CMPP_W_LT_UN_UN.18
CMPP_W_LT_UN_UC.18
CMPP_W_LT_UC_UN.18
CMPP_W_LT_UC_UC.18
CMPP_W_LLT_UN_UN.18
CMPP_W_LLT_UN_UC.18
CMPP_W_LLT_UC_UN.18
CMPP_W_LLT_UC_UC.18
CMPP_W_LEQ_UN_UN.18
CMPP_W_LEQ_UN_UC.18
CMPP_W_LEQ_UC_UN.18
CMPP_W_LEQ_UC_UC.18
CMPP_W_LLEQ_UN_UN.18
CMPP_W_LLEQ_UN_UC.18
CMPP_W_LLEQ_UC_UN.18
CMPP_W_LLEQ_UC_UC.18
CMPP_W_GT_UN_UN.18
CMPP_W_GT_UN_UC.18
CMPP_W_GT_UC_UN.18
CMPP_W_GT_UC_UC.18
CMPP_W_LGT_UN_UN.18
CMPP_W_LGT_UN_UC.18
CMPP_W_LGT_UC_UN.18
CMPP_W_LGT_UC_UC.18
CMPP_W_GEQ_UN_UN.18
CMPP_W_GEQ_UN_UC.18
CMPP_W_GEQ_UC_UN.18
CMPP_W_GEQ_UC_UC.18
CMPP_W_LGEQ_UN_UN.18
CMPP_W_LGEQ_UN_UC.18
CMPP_W_LGEQ_UC_UN.18
CMPP_W_LGEQ_UC_UC.18
));
VOG_o52c3i0 (ops(
L_B_C1_C1.19
L_H_C1_C1.19
L_W_C1_C1.19
));
VOG_o5c1i0 (ops(
EXTS_B.20
EXTS_H.20
));
VOG_o60c3i0 (ops(
S_B_C1.21
S_H_C1.21
S_W_C1.21
));
VOG_o6c1i0 (ops(
```

```
MOVE.22
));
VOG_o21c1i0 (ops(
PBRR.23
PBRA.23
));
}
SECTION Instruction_Format
{
template_id(size(3) type(IF_LEAF) kind(IF_CTRL_PORT));
eop_0 (size(1) type(IF_LEAF) kind(IF_CTRL_PORT));
multinop_0 (size(29) type(IF_LEAF) kind(IF_CTRL_PORT));
opgselect_0_0 (size(2) type(IF_LEAF) kind(IF_CTRL_PORT));
opg_0_0_0 (size(11) type(IF_LEAF) kind(IF_OP_GROUP) info(VOG_o79c4i0));
opg_0_0_1 (size(8) type(IF_LEAF) kind(IF_OP_GROUP) info(VOG_o80c4i0));
opg_0_0_2 (size(5) type(IF_LEAF) kind(IF_OP_GROUP) info(VOG_o78c4i0));
slot_0_0 (size(13) type(IF_ANDOR) kind(IF_TMPL_ELEM) orsel(opgselect_0_0) or(
opg_0_0_0
opg_0_0_1
opg_0_0_2
));
opgselect_0_1 (size(4) type(IF_LEAF) kind(IF_CTRL_PORT));
opg_0_1_0 (size(38) type(IF_LEAF) kind(IF_OP_GROUP) info(VOG_o22c1i0));
opg_0_1_1 (size(28) type(IF_LEAF) kind(IF_OP_GROUP) info(VOG_o1c1i0));
opg_0_1_2 (size(25) type(IF_LEAF) kind(IF_OP_GROUP) info(VOG_o3c1i0));
opg_0_1_3 (size(24) type(IF_LEAF) kind(IF_OP_GROUP) info(VOG_o4c1i0));
opg_0_1_4 (size(25) type(IF_LEAF) kind(IF_OP_GROUP) info(VOG_o2c1i0));
opg_0_1_5 (size(26) type(IF_LEAF) kind(IF_OP_GROUP) info(VOG_o18c1i0));
opg_0_1_6 (size(13) type(IF_LEAF) kind(IF_OP_GROUP) info(VOG_o5c1i0));
opg_0_1_7 (size(16) type(IF_LEAF) kind(IF_OP_GROUP) info(VOG_o6c1i0));
opg_0_1_8 (size(19) type(IF_LEAF) kind(IF_OP_GROUP) info(VOG_o21c1i0));
slot_0_1 (size(42) type(IF_ANDOR) kind(IF_TMPL_ELEM) orsel(opgselect_0_1) or(
opg_0_1_0
opg_0_1_1
opg_0_1_2
opg_0_1_3
opg_0_1_4
opg_0_1_5
opg_0_1_6
opg_0_1_7
opg_0_1_8
));
opgselect_0_2 (size(3) type(IF_LEAF) kind(IF_CTRL_PORT));
opg_0_2_0 (size(13) type(IF_LEAF) kind(IF_OP_GROUP) info(VOG_o38c2i0));
opg_0_2_1 (size(12) type(IF_LEAF) kind(IF_OP_GROUP) info(VOG_o37c2i0));
opg_0_2_2 (size(12) type(IF_LEAF) kind(IF_OP_GROUP) info(VOG_o34c2i0));
opg_0_2_3 (size(20) type(IF_LEAF) kind(IF_OP_GROUP) info(VOG_o31c2i0));
opg_0_2_4 (size(18) type(IF_LEAF) kind(IF_OP_GROUP) info(VOG_o27c2i0));
opg_0_2_5 (size(18) type(IF_LEAF) kind(IF_OP_GROUP) info(VOG_o28c2i0));
slot_0_2 (size(23) type(IF_ANDOR) kind(IF_TMPL_ELEM) orsel(opgselect_0_2) or(
opg_0_2_0
opg_0_2_1
opg_0_2_2
opg_0_2_3
opg_0_2_4
opg_0_2_5
));
opgselect_0_3 (size(3) type(IF_LEAF) kind(IF_CTRL_PORT));
opg_0_3_0 (size(13) type(IF_LEAF) kind(IF_OP_GROUP) info(VOG_o56c3i0));
opg_0_3_1 (size(13) type(IF_LEAF) kind(IF_OP_GROUP) info(VOG_o62c3i0));
opg_0_3_2 (size(16) type(IF_LEAF) kind(IF_OP_GROUP) info(VOG_o52c3i0));
opg_0_3_3 (size(16) type(IF_LEAF) kind(IF_OP_GROUP) info(VOG_o60c3i0));
slot_0_3 (size(19) type(IF_ANDOR) kind(IF_TMPL_ELEM) orsel(opgselect_0_3) or(
opg_0_3_0
```

```
opg_0_3_1
opg_0_3_2
opg_0_3_3
));
opgselect_0_4 (size(2) type(IF_LEAF) kind(IF_CTRL_PORT));
opg_0_4_0 (size(28) type(IF_LEAF) kind(IF_OP_GROUP) info(VOG_o1c1i1));
opg_0_4_1 (size(27) type(IF_LEAF) kind(IF_OP_GROUP) info(VOG_o2c1i1));
slot_0_4 (size(30) type(IF_ANDOR) kind(IF_TMPL_ELEM) orsel(opgselect_0_4) or(
opg_0_4_0
opg_0_4_1
));
template_0 (size(160) type(IF_AND) kind(IF_TEMPLATE) and(
eop_0
multinop_0
slot_0_0
slot_0_1
slot_0_2
slot_0_3
slot_0_4
));
eop_1 (size(1) type(IF_LEAF) kind(IF_CTRL_PORT));
multinop_1 (size(14) type(IF_LEAF) kind(IF_CTRL_PORT));
opgselect_1_0 (size(1) type(IF_LEAF) kind(IF_CTRL_PORT));
opg_1_0_0 (size(28) type(IF_LEAF) kind(IF_OP_GROUP) info(VOG_o1c1i0));
slot_1_0 (size(29) type(IF_ANDOR) kind(IF_TMPL_ELEM) orsel(opgselect_1_0) or(
opg_1_0_0
));
opgselect_1_1 (size(1) type(IF_LEAF) kind(IF_CTRL_PORT));
opg_1_1_0 (size(16) type(IF_LEAF) kind(IF_OP_GROUP) info(VOG_o52c3i0));
slot_1_1 (size(17) type(IF_ANDOR) kind(IF_TMPL_ELEM) orsel(opgselect_1_1) or(
opg_1_1_0
));
template_1 (size(64) type(IF_AND) kind(IF_TEMPLATE) and(
eop_1
multinop_1
slot_1_0
slot_1_1
));
eop_2 (size(1) type(IF_LEAF) kind(IF_CTRL_PORT));
multinop_2 (size(14) type(IF_LEAF) kind(IF_CTRL_PORT));
opgselect_2_0 (size(1) type(IF_LEAF) kind(IF_CTRL_PORT));
opg_2_0_0 (size(28) type(IF_LEAF) kind(IF_OP_GROUP) info(VOG_o1c1i0));
slot_2_0 (size(29) type(IF_ANDOR) kind(IF_TMPL_ELEM) orsel(opgselect_2_0) or(
opg_2_0_0
));
opgselect_2_1 (size(1) type(IF_LEAF) kind(IF_CTRL_PORT));
opg_2_1_0 (size(16) type(IF_LEAF) kind(IF_OP_GROUP) info(VOG_o60c3i0));
slot_2_1 (size(17) type(IF_ANDOR) kind(IF_TMPL_ELEM) orsel(opgselect_2_1) or(
opg_2_1_0
));
template_2 (size(64) type(IF_AND) kind(IF_TEMPLATE) and(
eop_2
multinop_2
slot_2_0
slot_2_1
));
eop_3 (size(1) type(IF_LEAF) kind(IF_CTRL_PORT));
multinop_3 (size(31) type(IF_LEAF) kind(IF_CTRL_PORT));
opgselect_3_0 (size(1) type(IF_LEAF) kind(IF_CTRL_PORT));
opg_3_0_0 (size(28) type(IF_LEAF) kind(IF_OP_GROUP) info(VOG_o1c1i0));
slot_3_0 (size(29) type(IF_ANDOR) kind(IF_TMPL_ELEM) orsel(opgselect_3_0) or(
opg_3_0_0
));
template_3 (size(64) type(IF_AND) kind(IF_TEMPLATE) and(
```

```
eop_3
multinop_3
slot_3_0
));
eop_4 (size(1) type(IF_LEAF) kind(IF_CTRL_PORT));
multinop_4 (size(8) type(IF_LEAF) kind(IF_CTRL_PORT));
opgselect_4_0 (size(1) type(IF_LEAF) kind(IF_CTRL_PORT));
opg_4_0_0 (size(19) type(IF_LEAF) kind(IF_OP_GROUP) info(VOG_o21c1i0));
slot_4_0 (size(20) type(IF_ANDOR) kind(IF_TMPL_ELEM) orsel(opgselect_4_0) or(
opg_4_0_0
));
template_4 (size(32) type(IF_AND) kind(IF_TEMPLATE) and(
eop_4
multinop_4
slot_4_0
));
eop_5 (size(1) type(IF_LEAF) kind(IF_CTRL_PORT));
multinop_5 (size(11) type(IF_LEAF) kind(IF_CTRL_PORT));
opgselect_5_0 (size(1) type(IF_LEAF) kind(IF_CTRL_PORT));
opg_5_0_0 (size(16) type(IF_LEAF) kind(IF_OP_GROUP) info(VOG_o6c1i0));
slot_5_0 (size(17) type(IF_ANDOR) kind(IF_TMPL_ELEM) orsel(opgselect_5_0) or(
opg_5_0_0
));
template_5 (size(32) type(IF_AND) kind(IF_TEMPLATE) and(
eop_5
multinop_5
slot_5_0
));
eop_6 (size(1) type(IF_LEAF) kind(IF_CTRL_PORT));
multinop_6 (size(1) type(IF_LEAF) kind(IF_CTRL_PORT));
opgselect_6_0 (size(1) type(IF_LEAF) kind(IF_CTRL_PORT));
opg_6_0_0 (size(26) type(IF_LEAF) kind(IF_OP_GROUP) info(VOG_o18c1i0));
slot_6_0 (size(27) type(IF_ANDOR) kind(IF_TMPL_ELEM) orsel(opgselect_6_0) or(
opg_6_0_0
));
template_6 (size(32) type(IF_AND) kind(IF_TEMPLATE) and(
eop_6
multinop_6
slot_6_0
));
eop_7 (size(1) type(IF_LEAF) kind(IF_CTRL_PORT));
multinop_7 (size(2) type(IF_LEAF) kind(IF_CTRL_PORT));
opgselect_7_0 (size(1) type(IF_LEAF) kind(IF_CTRL_PORT));
opg_7_0_0 (size(25) type(IF_LEAF) kind(IF_OP_GROUP) info(VOG_o2c1i0));
slot_7_0 (size(26) type(IF_ANDOR) kind(IF_TMPL_ELEM) orsel(opgselect_7_0) or(
opg_7_0_0
));
template_7 (size(32) type(IF_AND) kind(IF_TEMPLATE) and(
eop_7
multinop_7
slot_7_0
));
instruction(size(160) type(IF_ANDOR) kind(IF_INSTN)
  orsel(template_id) andor(
template_0
template_1
template_2
template_3
template_4
template_5
template_6
template_7
));
}
```

D Instruction Decode Tables

D.1 Template decode table

```
.design stdout_Decode_PLA
.type fr
.inputnames i3
.outputnames o0 o1 o2 o3



# ROOT : Node 0: IF_ANDOR (STEERING Node 1) IF_NC_INSTN
# TEMPLATE STEER i0 i1 i2  : Node 1: IF_LEAF (ifld 8) IF_NC_CTRL_PORT IF_STEER      WIDTH(3)

# TEMPLATE 0 : Node 2: IF_AND   IF_NC_TEMPLATE

# TEMPLATE WIDTH : 5*32=160
# OUTPUT o0 o1 o2
.field i0 i1 i2 o0 o1 o2
000 101

# END OF PACKET BIT

# INPUT  i3  : Node 3: IF_LEAF (ifld 6) IF_NC_CTRL_PORT IF_STEER      WIDTH(1)
# OUTPUT o0  : sequencer:sequencer_eop
.field i0 i1 i2 i3 o0
0000 0
0001 1

# TEMPLATE 1 : Node 295: IF_AND   IF_NC_TEMPLATE

# TEMPLATE WIDTH : 2*32=64
# OUTPUT o0 o1 o2
.field i0 i1 i2 o0 o1 o2
001 010

# END OF PACKET BIT

# INPUT  i3  : Node 296: IF_LEAF (ifld 0) IF_NC_CTRL_PORT IF_STEER      WIDTH(1)
# OUTPUT o0  : sequencer:sequencer_eop
.field i0 i1 i2 i3 o0
0010 0
0011 1

# TEMPLATE 2 : Node 330: IF_AND   IF_NC_TEMPLATE

# TEMPLATE WIDTH : 2*32=64
# OUTPUT o0 o1 o2
.field i0 i1 i2 o0 o1 o2
010 010

# END OF PACKET BIT

# INPUT  i3  : Node 331: IF_LEAF (ifld 3) IF_NC_CTRL_PORT IF_STEER      WIDTH(1)
# OUTPUT o0  : sequencer:sequencer_eop
.field i0 i1 i2 i3 o0
0100 0
```

```
0101 1

# TEMPLATE 3 : Node 366: IF_AND   IF_NC_TEMPLATE

# TEMPLATE WIDTH : 2*32=64
# OUTPUT o0 o1 o2
.field i0 i1 i2 o0 o1 o2
011 010

# END OF PACKET BIT

# INPUT  i3  : Node 367: IF_LEAF (ifld 1) IF_NC_CTRL_PORT IF_STEER      WIDTH(1)
# OUTPUT o0  : sequencer:sequencer_eop
.field i0 i1 i2 i3 o0
0110 0
0111 1

# TEMPLATE 4 : Node 389: IF_AND   IF_NC_TEMPLATE

# TEMPLATE WIDTH : 1*32=32
# OUTPUT o0 o1 o2
.field i0 i1 i2 o0 o1 o2
100 001

# END OF PACKET BIT

# INPUT  i3  : Node 390: IF_LEAF (ifld 7) IF_NC_CTRL_PORT IF_STEER      WIDTH(1)
# OUTPUT o0  : sequencer:sequencer_eop
.field i0 i1 i2 i3 o0
1000 0
1001 1

# TEMPLATE 5 : Node 405: IF_AND   IF_NC_TEMPLATE

# TEMPLATE WIDTH : 1*32=32
# OUTPUT o0 o1 o2
.field i0 i1 i2 o0 o1 o2
101 001

# END OF PACKET BIT

# INPUT  i3  : Node 406: IF_LEAF (ifld 2) IF_NC_CTRL_PORT IF_STEER      WIDTH(1)
# OUTPUT o0  : sequencer:sequencer_eop
.field i0 i1 i2 i3 o0
1010 0
1011 1

# TEMPLATE 6 : Node 425: IF_AND   IF_NC_TEMPLATE

# TEMPLATE WIDTH : 1*32=32
# OUTPUT o0 o1 o2
.field i0 i1 i2 o0 o1 o2
110 001

# END OF PACKET BIT

# INPUT  i3  : Node 426: IF_LEAF (ifld 4) IF_NC_CTRL_PORT IF_STEER      WIDTH(1)
# OUTPUT o0  : sequencer:sequencer_eop
.field i0 i1 i2 i3 o0
1100 0
1101 1

# TEMPLATE 7 : Node 445: IF_AND   IF_NC_TEMPLATE
```

```
# TEMPLATE WIDTH : 1*32=32
# OUTPUT o0 o1 o2
.field i0 i1 i2 o0 o1 o2
111 001

# END OF PACKET BIT

# INPUT  i3  : Node 446: IF_LEAF (ifld 5) IF_NC_CTRL_PORT IF_STEER      WIDTH(1)
# OUTPUT o0  : sequencer:sequencer_eop
.field i0 i1 i2 i3 o0
1110 0
1111 1
```

D.2 Shift unit decode table

```
.design stdout_PD_i_shift_9_decode_PLA
.type fr
.inputnames i0 i1 i2 i22 i23 i24 i25 i26 i27 i28 i29 i54 i55 i56 i57
.outputnames o0 o1 o2 o3 o4 o5 o6 o7



# ROOT : Node 0: IF_ANDOR (STEERING Node 1) IF_NC_INSTN
# TEMPLATE STEER i0 i1 i2  : Node 1: IF_LEAF (ifld 8) IF_NC_CTRL_PORT IF_STEER      WIDTH(3)

# TEMPLATE 0 : Node 2: IF_AND   IF_NC_TEMPLATE

# OPGROUP SET  : Node 24: IF_ANDOR (STEERING Node 25) IF_NC_TMPL_ELEM
# OPGROUP STEER i54 i55 i56 i57  : Node 25: IF_LEAF (ifld 22) IF_NC_CTRL_PORT IF_STEER      WIDTH(4)

# VLIW_OP_GROUP  : Node 88: IF_ANDOR (STEERING Node 89) IF_NC_OP_GROUP  VOG_o2c1i0(EOS_intarith2_intshift)

# INPUT  i22 i24  : Node 91: IF_LEAF (ifld 164) IF_NC_CTRL_PORT FU_OPCODE     WIDTH(2)
# OUTPUT o0 o1 o2  : PD_i_shift_9:PD_i_shift_9_op
.field i0 i1 i2 i54 i55 i56 i57 i22 i24 o0 o1 o2
# OPCODE SHL_W.16 : FUCODE 0
000010100 000
# OPCODE SHR_W.16 : FUCODE 1
000010101 001
# OPCODE SHLA_W.16 : FUCODE 2
000010110 010
# OPCODE SHRA_W.16 : FUCODE 3
000010111 011

# IOFORMAT STEER  : Node 89: IF_LEAF (ifld 162) IF_NC_CTRL_PORT IF_STEER      WIDTH(0)

# IO Descriptor: pr ? gpr cr s , gpr cr s  : gpr cr

# INPUT IOSET PORT  0 : Node 92: IF_ANDOR (STEERING Node 93) IF_NC_IO_SET

# INPUT  i25 i26  : Node 93: IF_LEAF (ifld 166) IF_NC_CTRL_PORT MUX_SEL       WIDTH(2)
# OUTPUT o3 o4  : mux_3_1_w32_64:mux_3_1_w32_64_sel
.field i0 i1 i2 i54 i55 i56 i57 i25 i26 o3 o4
000010100 00
000010101 01
000010110 10

# INPUT IOSET PORT  1 : Node 97: IF_ANDOR (STEERING Node 98) IF_NC_IO_SET

# INPUT  i27 i28  : Node 98: IF_LEAF (ifld 169) IF_NC_CTRL_PORT MUX_SEL       WIDTH(2)
```

```
# OUTPUT o5 o6  : mux_3_1_w32_66:mux_3_1_w32_66_sel
.field i0 i1 i2 i54 i55 i56 i57 i27 i28 o5 o6
000010100 00
000010101 01
000010110 10

# OUTPUT IOSET PORT 0 : Node 102: IF_ANDOR (STEERING Node 103) IF_NC_IO_SET

# INPUT  i29  : Node 103: IF_LEAF (ifld 174) IF_NC_CTRL_PORT DEMUX_SEL     WIDTH(1)
# OUTPUT o7  : demux_1_2_w1_68:demux_1_2_w1_68_sel
.field i0 i1 i2 i54 i55 i56 i57 i29 o7
00001010 0
00001011 1

# TEMPLATE 1 : Node 295: IF_AND   IF_NC_TEMPLATE
# OUTPUT o0 o1 o2  : PD_i_shift_9:PD_i_shift_9_op
.field i0 i1 i2 o0 o1 o2
# OPCODE NO_OP : FUCODE 4
001 100

# TEMPLATE 2 : Node 330: IF_AND   IF_NC_TEMPLATE
# OUTPUT o0 o1 o2  : PD_i_shift_9:PD_i_shift_9_op
.field i0 i1 i2 o0 o1 o2
# OPCODE NO_OP : FUCODE 4
010 100

# TEMPLATE 3 : Node 366: IF_AND   IF_NC_TEMPLATE
# OUTPUT o0 o1 o2  : PD_i_shift_9:PD_i_shift_9_op
.field i0 i1 i2 o0 o1 o2
# OPCODE NO_OP : FUCODE 4
011 100

# TEMPLATE 4 : Node 389: IF_AND   IF_NC_TEMPLATE
# OUTPUT o0 o1 o2  : PD_i_shift_9:PD_i_shift_9_op
.field i0 i1 i2 o0 o1 o2
# OPCODE NO_OP : FUCODE 4
100 100

# TEMPLATE 5 : Node 405: IF_AND   IF_NC_TEMPLATE
# OUTPUT o0 o1 o2  : PD_i_shift_9:PD_i_shift_9_op
.field i0 i1 i2 o0 o1 o2
# OPCODE NO_OP : FUCODE 4
101 100

# TEMPLATE 6 : Node 425: IF_AND   IF_NC_TEMPLATE
# OUTPUT o0 o1 o2  : PD_i_shift_9:PD_i_shift_9_op
.field i0 i1 i2 o0 o1 o2
# OPCODE NO_OP : FUCODE 4
110 100

# TEMPLATE 7 : Node 445: IF_AND   IF_NC_TEMPLATE

# OPGROUP SET  : Node 448: IF_ANDOR (STEERING Node 449) IF_NC_TMPL_ELEM
# OPGROUP STEER i29  : Node 449: IF_LEAF (ifld 316) IF_NC_CTRL_PORT IF_STEER      WIDTH(1)

# VLIW_OP_GROUP  : Node 450: IF_ANDOR (STEERING Node 451) IF_NC_OP_GROUP  VOG_o2c1i0(EOS_intarith2_intshift)

# INPUT  i22 i23  : Node 453: IF_LEAF (ifld 163) IF_NC_CTRL_PORT FU_OPCODE     WIDTH(2)
# OUTPUT o0 o1 o2  : PD_i_shift_9:PD_i_shift_9_op
.field i0 i1 i2 i29 i22 i23 o0 o1 o2
# OPCODE SHL_W.16 : FUCODE 0
111100 000
# OPCODE SHR_W.16 : FUCODE 1
111101 001
```

```
# OPCODE SHLA_W.16 : FUCODE 2
111110 010
# OPCODE SHRA_W.16 : FUCODE 3
111111 011

# IOFORMAT STEER  : Node 451: IF_LEAF (ifld 317) IF_NC_CTRL_PORT IF_STEER      WIDTH(0)

# IO Descriptor: pr ? gpr cr s , gpr cr s  : gpr cr

# INPUT IOSET PORT  0 : Node 454: IF_ANDOR (STEERING Node 455) IF_NC_IO_SET

# INPUT  i24 i25  : Node 455: IF_LEAF (ifld 165) IF_NC_CTRL_PORT MUX_SEL       WIDTH(2)
# OUTPUT o3 o4  : mux_3_1_w32_64:mux_3_1_w32_64_sel
.field i0 i1 i2 i29 i24 i25 o3 o4
111100 00
111101 01
111110 10

# INPUT IOSET PORT  1 : Node 459: IF_ANDOR (STEERING Node 460) IF_NC_IO_SET

# INPUT  i26 i27  : Node 460: IF_LEAF (ifld 170) IF_NC_CTRL_PORT MUX_SEL       WIDTH(2)
# OUTPUT o5 o6  : mux_3_1_w32_66:mux_3_1_w32_66_sel
.field i0 i1 i2 i29 i26 i27 o5 o6
111100 00
111101 01
111110 10

# OUTPUT IOSET PORT 0 : Node 464: IF_ANDOR (STEERING Node 465) IF_NC_IO_SET

# INPUT  i28  : Node 465: IF_LEAF (ifld 173) IF_NC_CTRL_PORT DEMUX_SEL     WIDTH(1)
# OUTPUT o7  : demux_1_2_w1_68:demux_1_2_w1_68_sel
.field i0 i1 i2 i29 i28 o7
11110 0
11111 1
```