Machine-Description Driven Compilers for EPIC Processors

B. Ramakrishna Rau, Vinod Kathail, Shail Aditya
Compiler and Architecture Research
HPL-98-40
September, 1998

E-mail: [rau,kathail,aditya]@hpl.hp.com

retargetable compilers, table-driven compilers, machine description, processor description, instruction-level parallelism, EPIC processors, VLIW processors, EPIC compilers, VLIW compilers, code generation, scheduling, register allocation

In the past, due to the restricted gate count available on an inexpensive chip, embedded DSPs have had limited parallelism, few registers and irregular, incomplete interconnectivity. More recently, with increasing levels of integration, embedded VLIW processors have started to appear. Such processors typically have higher levels of instruction-level parallelism, more registers, and a relatively regular interconnect between the registers and the functional units. The central challenges faced by a code generator for an EPIC (Explicitly Parallel Instruction Computing) or VLIW processor are quite different from those for the earlier DSPs and, consequently, so is the structure of a code generator that is designed to be easily retargetable.

In this report, we explain the nature of the challenges faced by an EPIC or VLIW compiler and present a strategy for performing code generation in an incremental fashion that is best suited to generating high-quality code efficiently. We also describe the Operation Binding Lattice, a formal model for incrementally binding the opcodes and register assignments in an EPIC code generator. As we show, this reflects the phase structure of the EPIC code generator. It also defines the structure of the machine-description database, which is queried by the code generator for the information that it needs about the target processor. Lastly, we discuss general features of our implementation of these ideas and techniques in Elcor, our EPIC compiler research infrastructure.

Internal Accession Date Only

An abbreviated version of the report will appear in an issue of *Design Automation for Embedded Systems*, Kluwer Academic Publishers, 1999. Presented at the *3rd International Workshop on Code Generation for Embedded Processors*, Witten, Germany, March 4-6, 1998.

1 Introduction

In the past, due to the restricted gate count available on an inexpensive chip, embedded digital signal processors (DSPs) have had limited parallelism, few registers and irregular, incomplete interconnectivity [1-3]. As a result, compilers for such processors have focused primarily on the task of code selection, i.e., choosing a sequence of instructions that minimizes the number of instructions executed without requiring more registers than the number present. A large amount of work has been done in this area [4-6]. These DSPs are in many ways reminiscent of the early "attached processors", such as Floating Point Systems' AP-120B and FPS-164 [7] for which, due to their irregular and idiosyncratic architecture, it was very difficult to write high quality compilers. Both sets of processors share the property that they were typically designed to support some particular computation, such as convolution or (in the case of the AP-120B) the FFT, and the data paths reflected this. Typically, such machines were horizontally microprogrammed, and their programmability came as an afterthought; the read-only control store was replaced by a writeable one. In effect, these processors were like special-purpose ASICs with a thin veneer of programmability. However, in one important respect these machines were already taking the first step towards VLIW in that the operations that they executed were not micro-operations but, rather, operations such as floating-point adds and multiplies.

The first-generation of VLIW processors[1] were motivated by the specific goal of cleaning up the architecture of the attached processor sufficiently to make it possible to write good compilers [8, 9]. Such processors typically had higher levels of instruction-level parallelism (ILP), more registers, and a relatively regular interconnect between the registers and the functional units. Furthermore, the operations were RISC-like (in that their sources and destinations were registers), not "micro-operations" (which merely source or sink their operands from or to buses). Recently, with increasing levels of integration, DSPs have begun to appear that have a VLIW architecture [10, 11]. This current generation of embedded VLIW processors reflects the state of the art of VLIW in the mini-supercomputer space a decade ago [12, 13].

[1] Note that our use of the term VLIW processor is specifically intended to differentiate it from the array processors of the past and the DSPs of today. Even though these might have certain VLIW attributes, such as the ability to issue multiple operations in one instruction, we view them as being half way between VLIW processors and horizontally microprogrammed processors.

In the meantime, VLIW has continued to evolve into an increasingly general-purpose architecture, providing high levels of ILP and incorporating a number of advanced features [14, 15]. This evolved style of VLIW is termed EPIC (Explicitly Parallel Instruction Computing). Our current research is based on the belief that the EPIC style of architecture will show up in embedded DSPs a few years down the road. In the rest of this report, we shall use the term EPIC to include VLIW as well.

For EPIC processors, since the primary focus is on achieving high levels of ILP, the most important compiler task is to achieve as short a schedule as possible. The scheduler and register allocator are, therefore, the key modules. These two topics have received a great deal of attention over the years in many research communities, resulting in a vast body of literature. In this report, we are concerned not with the scheduling and register allocation algorithms but with the information that EPIC compilers need about the processor for which they are performing these functions. The identification of this information makes it possible to write "table-driven" EPIC compilers which have no detailed assumptions regarding the processor built into the code.

Instead, such a compiler makes queries to a *machine-description database (mdes)* which provides the information needed by the compiler about the processor. Such a compiler can be retargeted to different EPIC processors by changing the contents of this database, despite the fact that the processors vary widely, including in the number of functional units, their pipeline structure, their latencies, the set of opcodes that each functional unit can execute, the number of register files, the number of registers per register file, their accessibility from the various functional units and the busing structure between the register files and the functional units.

An mdes-driven compiler is of particular value in the context of developing a capability to automatically synthesize custom EPIC ASIPs (Application-Specific Instruction-Set Processors), where one of the obstacles, both to the evaluation of candidate designs as well as to the use of the selected one, is that of automatically generating a high-quality compiler for the synthesized processor.

In this report, we discuss the mdes-driven aspects of compilers for EPIC processors. It is important to stress that our goal is not to be able to build an mdes-driven compiler that can target any arbitrary processor. The target space is limited to a stylized class of EPIC processors for which we know how to generate good code using systematic rather than ad hoc techniques. Clearly, if one does not even understand how to write a compiler for a

single target processor using systematic, formal techniques, the goal of automatically generating a compiler for any processor in the permissible space will be impossible to achieve. Also, our focus in this report is restricted to the mdes required by the scheduler and register allocator, since these are the most important modules of an EPIC compiler. Specifically, we shall not address other phases of a compiler such as code selection, partitioning across multiple clusters and optimizations concerned with the memory hierarchy.

The outline of the report is as follows. In Section 2, we describe our space of target EPIC processors. To a large extent, the set of issues that we address in achieving the mdes-driven capability of an EPIC code generator are complementary to the ones that are of primary importance in writing retargetable code generators for contemporary DSPs. In Section 3, we explain the additional issues in EPIC code generation and develop the key concepts of the expanded computation graph, full-connectedness, opcode and register option sets, and access-equivalent operation sets. Section 4 develops the concepts of binding hierarchies for opcodes and registers as well as the Operation Binding Lattice (OBL). It also articulates our model of phase ordered EPIC code generation. In Section 5, we describe all the relevant information about the target machine that has to be stored in the mdes, and in Section 6 we explain how it is used by the EPIC code generator modules. Section 7 provides a brief overview of the implementation of the mdes in Elcor, our EPIC compiler research infrastructure. Section 8 reviews the antecedents of this work, as well as other related work. Section 9 summarizes the contributions of this report.

2 The space of target EPIC processors

EPIC processors evolved in response to a different set of emphases than did contemporary DSPs. The first one was that they provide relatively high levels of ILP. The second one was that they lend themselves to the implementation of efficient, high-quality compilers. The space of processors was consciously restricted to promote the latter goal. As a result, the typical structure of an EPIC processor is different in a number of important ways which, in turn, leads to a different set of code generation challenges. We wish to highlight these points of difference.

2.1 Key features of an EPIC processor

Figure 1 shows an EPIC processor, which we shall use as a running example, throughout this report, to illustrate various points. We first list those features that are the basic

requirements for achieving high levels of instruction-level parallelism while ensuring that it is possible to write an efficient, high quality code generator.

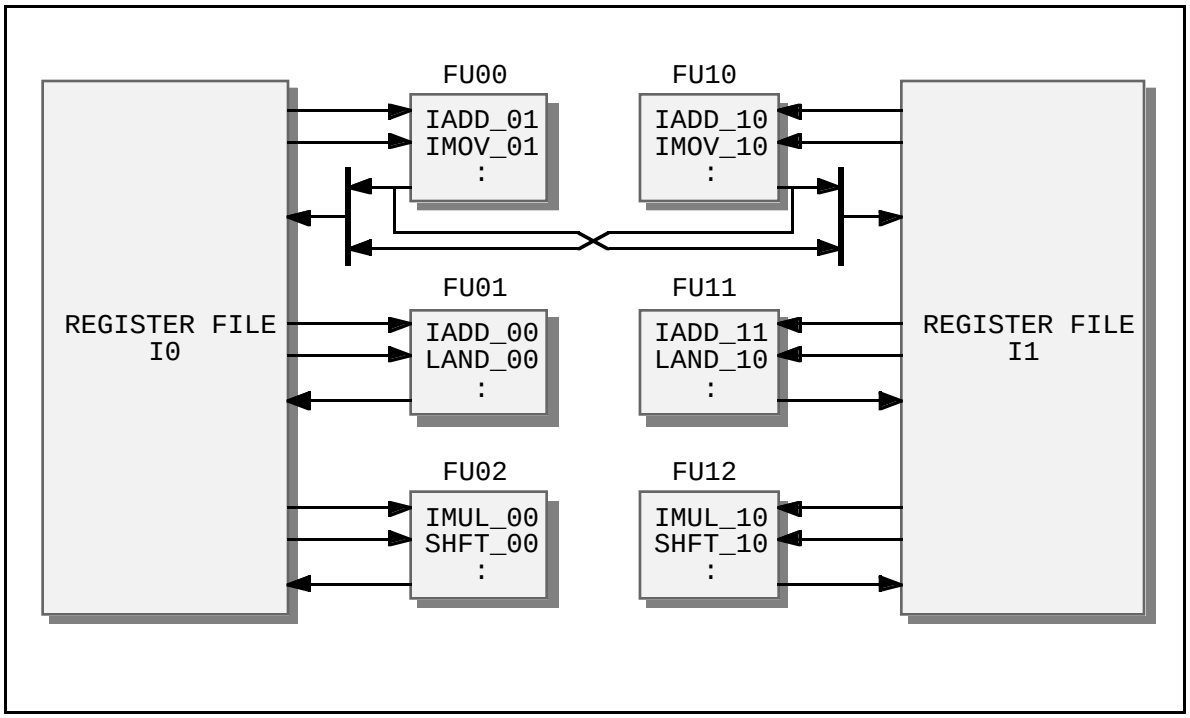

Figure 1: A sample EPIC processor.

- **Multiple pipelined functional units**. Since the EPIC architecture was developed to exploit ILP, the most basic characteristic of our space of target EPIC processors is the presence of multiple functional units (FUs) that can execute multiple operations in parallel. Often, the parallelism is such as to allow multiple identical operations to be issued simultaneously. In Figure 1, integer add operations can be performed on four of the FUs, whereas integer multiply operations can be performed on the other two FUs. The machine has been designed in such a way that all six FUs can initiate operations simultaneously.

  In real machines, there are always at least a few opcodes that take more than one cycle to generate their results. Also, depending on the relative importance of such an opcode, its execution may be not pipelined at all, be fully pipelined (i.e., a new operation can be issued every cycle), or be partially pipelined (e.g., a divide may take 24 cycles to complete, but a new divide can be started every 10 cycles).

- **Register-register operations**. A fundamental requirement for facilitating the task of writing a scheduler is that it be possible to write all results, produced during the same cycle, to a register file. In certain processors, a result is transient--it is necessary to consume it directly off the result bus either because the result bus is not connected to a register file, or because the register file does not have enough write bandwidth to accomodate this result as well as the other results that are simultaneously produced. All of the operations that consume this result have to be scheduled at precisely the cycle on which this result is produced. Often, this is impossible due to resource conflicts between those operations. It is difficult to write a high quality, efficient scheduler for such a processor. Consequently, in an EPIC processor, all of the operations, other than loads and stores, are such that they operate from register to register. (Optionally, there can be register bypass logic that provides the additional capability to pass the result directly from a result bus to a source bus.)
- **Register files.** Rather than a heterogeneous collection of registers with differing special capabilities and interconnectivity, the registers in an EPIC processor are organized into one or more register files, each of which consists of a number of similar registers.
- **Large numbers of registers per register file.** Rather than taking the view that most of the operands of the operations executed reside in memory (and providing the requisite memory bandwidth), the view taken is that most operands will be in registers. The expectation is that a variable will primarily be referenced in a register, with, perhaps, an initial load from memory or a final store to memory. Consequently, a relatively large numbers of registers are provided, compared to most contemporary DSPs. The code generator takes advantage of this and is designed with the view that register spill is infrequent, not the norm[2]. This also reduces the memory bandwidth requirement, which is particularly valuable at high levels of ILP.

Competently architected EPIC processors have a number of additional features that we also include within our space of target architectures.

- **Multiple register files.** From the compiler's viewpoint, the most convenient arrangement is for all similar registers to be in the same register file, e.g., all the 16-bit registers in one file and all the 32-bit registers in another one. But in a highly parallel EPIC processor, this implies very highly ported register files, which tend to be slow

[2] This assumption is central to our use of the phase ordering described in Section 4.

and expensive. Instead, such highly ported register files are split into two or more register files, with a subset of the FUs having access to each register file. In our example processor, the integer registers have been divided into two register files, I0 and I1.

- **Heterogeneous functional units.** When designing a processor that can issue multiple integer operations per cycle, it is unusual for all of the integer or floating-point FUs to be replicas of one another. In Figure 1, the processor has been designed to be able to perform an IADD on every one of four FUs. But on two of them, it can perform an IMOV (copy) operation, while on the other two it can perform an LAND (logical-AND) operation. This leads to a set of heterogeneous FUs which, in the extreme case, are all dissimilar.

- **Shared register ports.** It is often impractical to provide every FU with dedicated register ports for every opcode that it can execute. Opcodes that are provided for completeness, but which are infrequently used, use register ports that they share with other FUs. FU00 and FU10 each need a write port into both register files to permit data to be copied from one register file to the other. Instead of dedicated ports, they share a single write port per register file. Thus, while F00 is writing the destination register of an IMOV_01 into I1, FU10 cannot be completing an operation at the same time. Correct code generation must honor this constraint.

- **Shared instruction fields.** Likewise, bits in the instruction word are a valuable commodity, and must often be shared. The instruction format that permits the specification of a long literal may do so by using the instruction bits that would normally have specified, for instance, the third operand of a multiply-add operation. This precludes the multiply-add opcode, but not the two-input multiply or add opcode, from being issued in the same cycle as the opcode that specifies a long literal.

- **Interruption model.** The schedule constructed by the compiler for a program can be disrupted by events, such as interrupts, page faults or arithmetic exceptions, which require that the execution of the program be suspended and then be resumed after the interrupt or exception handler has been executed. We shall refer to such events, generically, as *interruptions*. We shall consider two hardware strategies for dealing with interruptions. The first one halts the issue of any further instructions, takes a snapshot of the processor state for the executing program, including the state of execution of the pipelined operations, and then invokes the interruption handler. After the handler has executed, the program's processor state is restored, and program execution is resumed. The net effect, from the viewpoint of the program, is that its

execution was frozen in place during the execution of the interruption handler. We refer to this as the *freeze model*.

The second strategy halts the issue of any further instructions, but permits those that have already been issued to go to completion. Once the pipelines have drained, the interruption handler is executed, after which instruction issue takes up where it left off. We refer to this as the *drain model*[3]. Although the drain model is considerably less complex to implement from a hardware viewpoint, it requires that specific measures be taken during scheduling and code generation to ensure correctness. These measures, which are discussed in Section 6, affect the information needed in the mdes regarding execution latencies.

In the case of pipelined branch operations, i.e., branches which have a latency of more than one cycle between the initiation of the branch operation and the initiation of the target operation, we always use the freeze model regardless of the model used for the rest of the operations. Since the semantics of a branch are to alter the flow of control and start issuing a new stream of instructions, the drain model, which would permit the branch to continue to completion, but which would nevertheless prevent any further instructions being issued, is meaningless.

In addition to those mentioned above, EPIC processors can have a number of other features such as predication, control speculation, data speculation, rotating registers and programmatic cache management [14, 15] which, individually and collectively, can have a great impact on the quality of the schedule. Since these features do not introduce any additional issues that are pertinent to the subject of this report, we shall not discuss them further.

2.2 The compiler's view of a processor

Conceptually, the compiler is playing the role of a human assembly language programmer. As such, it is not interested in the structural description of the hardware. More precisely, it is only interested in the hardware structure indirectly, to the extent that this is reflected in the architectural description of the machine as might be found in the Architecture Manual. Specifically, rather than needing the equivalent of Figure 1 in the form of an HDL description, it only needs to know, for each opcode, which registers can be accessed as

[3] In previous works by the authors, the freeze and drain models have been termed the EQ model and the LEQ model, respectively. We believe that way in which we conceptualized these issues in the past was imprecise, and that our current terminology better reflects our current way of thinking about exception handling.

each of the source and destination operands[4]. Table 1 lists some of the opcodes and the set of registers[5] to which each of the source or destination operands of those opcodes can be bound.

Table 1: The input-output behavior of selected opcodes

Semantics	Opcode	Source 1	Source 2	Destination
Integer Add	IADD_00	{I0}	{I0}	{I0}
	IADD_01	{I0}	{I0}	{I0,I1}
	IADD_10	{I1}	{I1}	{I0,I1}
	IADD_11	{I1}	{I1}	{I1}
Integer Multiply	IMUL_00	{I0}	{I0}	{I0}
	IMUL_10	{I1}	{I1}	{I1}
Copy	IMOV_01	{I0}	-	{I0,I1}
	IMOV_10	{I1}	-	{I0,I1}

An *operation* consists of an opcode and a register tuple—one register each per operand. An important point to note is that the choice of register for one operand can sometimes restrict the choice for another operand. Imagine that in our example processor FU00, in addition to the two read ports from I0, also possessed its own read port from I1 which could be used as the input port for either of the inputs of FU00. In this case, IADD_01 could source a register in either I0 or I1 as its left source and it could source a register in either I0 or I1 as its left source. But it cannot simultaneously source two registers in I1 since it has only one read port from it. Thus the accessibility of registers by an opcode must be specified in terms of which register tuples are legal. It is insufficient to merely specify which registers are accessible on an operand by operand basis. Consequently, the first row for IADD_00 in Table 1 should be interpreted as stating that every register tuple in the Cartesian product $\{I0\} \times \{I0\} \times \{I0\}$ is legal.

[4] Additionally, for an EPIC processor, it also needs to know the relevant latencies and resource usage of each of these operations, since it is the responsibility of the compiler to ensure a correct schedule. We shall return to this issue later on.

[5] As a notational convenience we shall use {I0} and {I1} to refer to the set of all the registers in register files I0 and I1, respectively.

Note that Table 1 does not contain any explicit information about the structure of the hardware. Instead, for each opcode, it specifies which sets of registers can be the sources or destination. The compiler does not need to know about FUs, register files and interconnects; all it needs to know is which opcodes can access which registers. This opcode- and register-centric view will be reflected throughout the report.

3 Issues in EPIC code generation

Due to the nature of our space of target EPIC processors, performing traditional code selection for an EPIC processor in a machine-description driven fashion is no different than for a CISC or RISC processor [16-20], depending on whether or not the EPIC processor has complex opcodes such as loads, stores or branches with complex addressing modes. In contrast, retargetable code selection for contemporary DSPs is quite different, because they fall outside of this space on one or more counts[6] [21-24, 5, 25]. Since there is little about traditional code selection that is unique to EPIC, we shall not dwell upon it. However, there are other EPIC-related code selection and code generation steps that are somewhat unique. These are the topic of this section.

3.1 Imperatives in an EPIC code generator

The driving issue in EPIC code generation is that, in general, each type of operation can be executed on more than one FU and that in each of these options the operation may execute out of a different subset of the registers. The code generator is faced with a multiplicity of choices which have to be made in such a way as to minimize the length of the resulting schedule.

Our discussion of the EPIC code generation process makes extensive use of the notion of a computation graph which we define as follows:

> **Definition:** A *computation graph* is a directed, bipartite graph whose vertices are either operators or variables. The operators specify the action to be performed upon the variables to which they are connected. The variables serve as the operands of the operators. The operator, along with its operands, constitutes an operation.

[6] There are two main reasons why the traditional DSP falls outside of our space of processors. One is that every operation may not be register-to-register; certain results may have to be consumed right off the result bus, if the available ILP is to be achieved. The second reason is that whereas it may have many "register files", each of them typically has just one or two registers. These two factors have major implications for code selection.

In order to constrast the difference between conventional code generation and EPIC code generation, consider the code fragment in Figure 2, which is intended to be compiled to the machine described in Table 1. On the left side of Figure 2, we have the computation graph for the machine-independent code. The vertices are either operators (circles) or variables (squares). At this stage of compilation, the operators are associated with machine-independent *semantic opcodes* for a virtual machine which specify only the semantics of the operator (e.g. an integer add) but not how it is to be implemented. The variables are associated with machine-independent *virtual registers* which specify the data type of the variable, but nothing about which physical register in the machine is to be used to hold this variable. Conventional code selection rewrites this computation taking into account the types of opcodes that exist in the actual machine, e.g., CISC-style compound opcodes, and the types of registers present. (In the case of our example machine, this is a null step.) At this point, for a sequential processor, code selection is complete and register allocation and scheduling would take place.

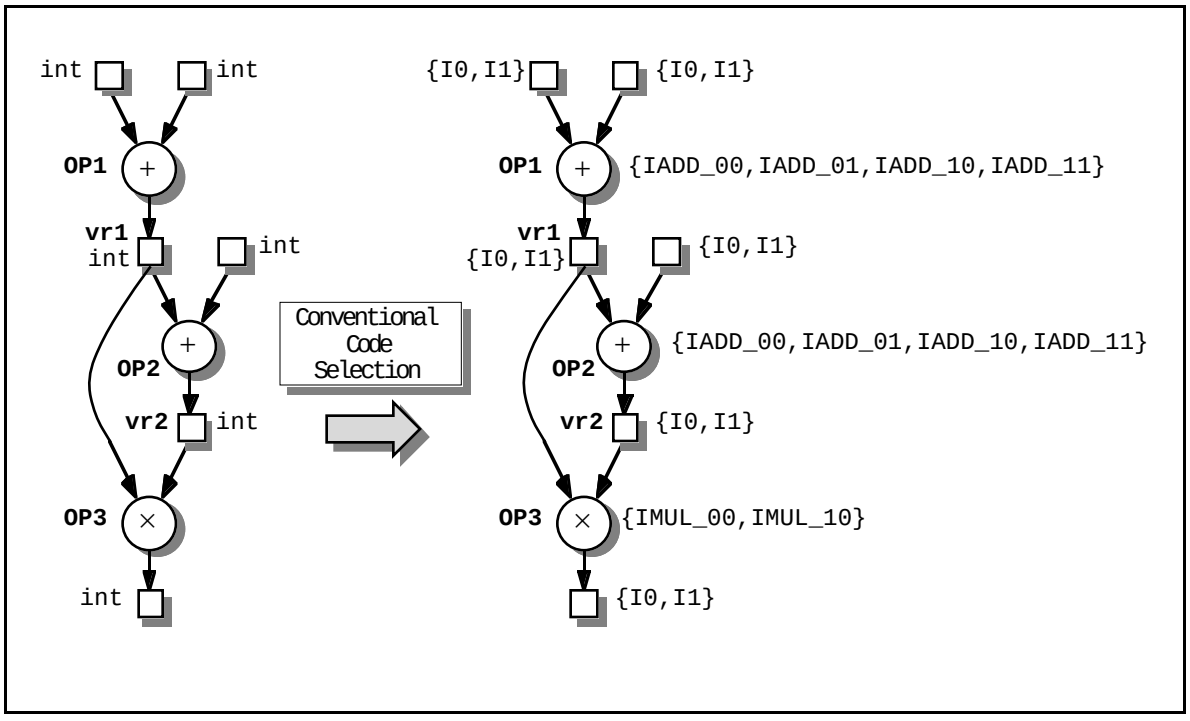

Figure 2: A fragment of a computation graph before and after conventional code selection.

In the case of EPIC code generation, the conventional code selection step has merely narrowed the options in Figure 2 down to those listed against OP1, OP2, and OP3 on the

right side of Figure 2. The four opcode options listed against OP1 and OP2 represent all of the ways in which an integer add can be performed on this machine. We refer to them jointly as the generic IADD_G opcode set. In general, a *generic opcode set* is the set of all the opcodes available for implementing an action having the same semantics. Also, in Figure 2, the options listed for vr1 and vr2 are the union of all the registers in I0 and I1. We refer to them jointly as a generic register set. In general, a *generic register set* consists of all of the registers that can hold a variable of a particular data type. In Figure 2, the opcodes and the operands are both far from being unambiguously specified.

For an EPIC compiler, the primary measure of code quality is the schedule length. The entire code generator is designed with this in mind and, over the years, a number of complex, sophisticated scheduling algorithms have been developed [26-31]. All of these scheduling algorithms benefit from having as many options as possible available to choose from while deciding when and where to schedule each operation. It is desirable, therefore, to avoid making premature code selection decisions. Rather than bind OP1 to use the IADD_00 opcode, one would rather leave it up to the scheduler to choose which of the four integer add opcodes is best used. This allows the scheduler to use whichever FU happens to be free, with a view to minimizing the schedule length. Doing so constitutes a form of delayed code selection; only after the scheduler is done, do we know which precise opcode has been selected.

On the other hand, we do wish to constrain the options available to the scheduler to those that the scheduler can handle efficiently. Consider again the example in Figure 2. If the scheduler is given full freedom in selecting which multiply and add opcodes to use, it might end up selecting, for OP1, OP2 and OP3, the opcodes in either Figure 3a or Figure 3b. Referring to Table 1 we see that the choices made in Figure 3a are structurally valid. It is possible to find a register file, I1, which can be the destination of IADD_01 and the source for IADD_11 and IMUL_10. In contrast, the choices made in Figure 3b are jointly incorrect. IADD_00 can only write to register file I0, whereas IMUL_10 can only read from register files I1. Thus there is no register which can be allocated to vr1 such that it is accessible by both opcodes. One solution is for the scheduler, in the course of scheduling, to insert an IMOV_01 operator which copies the result of OP1 from vr1 to a new variable, vr3, which is then read by OP3 (Figure 3c). Now, vr1, v2 and vr3 can be allocated to I0, I1 and I1, respectively, and be accessible to all the operators that source or sink them.

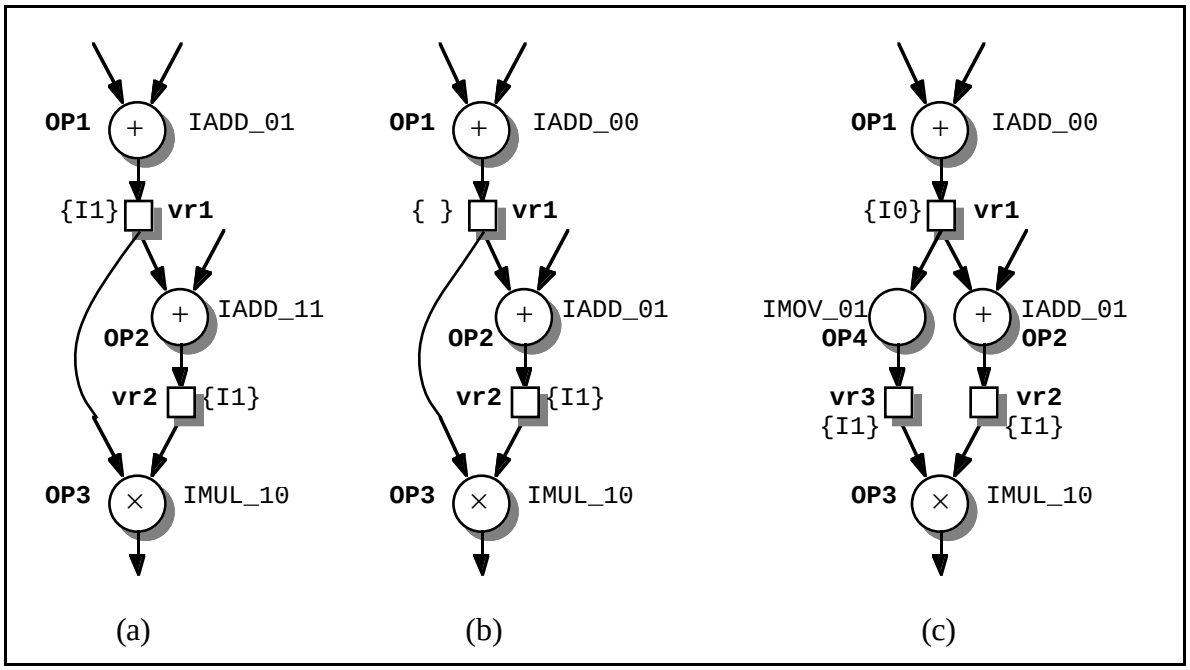

Figure 3: (a) A valid choice of opcodes. (b) An invalid choice of opcodes since vr1 cannot be assigned to either register file I0 or I1 and be accessible to both OP1 and OP3. (c) The insertion of a copy operator to make the choice of opcodes valid once again.

Unfortunately, there is a new problem with this solution. All of the scheduling algorithms mentioned above perform analyses, of one sort or other, on the computation graph prior to scheduling it. Common examples are the annotation of vertices in the computation graph with the length of the longest path to a terminal vertex in the graph or, in the case of loops, the identification of the critical recurrence cycles or the number of operations of each type (according to some pre-specified grouping) [29, 28, 31]. The scheduling heuristics are guided by the results of these analyses, which would be invalidated if the computation graph is modified. Each time such a copy operator is inserted, the analyses and scheduling would have to be repeated--a level of inefficiency that is hard to tolerate.

If these scheduling algorithms are to be used, then in the interests of efficiency, it is important for the scheduler to avoid changing the computation graph during scheduling by inserting or deleting operators, or by altering the structure of the computation graph. In other words, it is inappropriate to give the scheduler complete freedom in choosing the opcode. Some amount of binding is needed prior to scheduling.

It might well be the case that if best use is to be made of the available FUs, the insertion of a certain number of copy operators is essential to successfully spreading the computation

out over the entire machine. Since three of the FUs in our example machine, operate primarily out of register file I0, and the other three out of I1, the computation should be partitioned so that approximately half of the IADDs and IMULs execute on each trio of FUs. Whenever there is a flow dependence between these two partitions, IMOV operators are needed to copy the result from one register file to the other. Obtaining a good schedule demands that we do this. However, efficiency demands that the copy operators be inserted, once and for all, into the computation graph prior to scheduling, and with no further copy insertion thereafter. This leads to an additional step, *partitioning*, which precedes scheduling, decides how best to spread the computation over the available FUs, and inserts the appropriate copy operators.

An EPIC compiler typically contains a number of optimization and transformation steps that modify the computation graph. Most of these occur before partitioning. This includes a long list of traditional optimizations as well as ILP-related optimizations such as tail duplication, loop unrolling, expression re-association and critical path length reduction [30, 32, 33]. Others, most notably spill code insertion, occur after scheduling and register allocation but before post-pass scheduling (see Section 4). Our position is only that the modification of the computation graph not occur during scheduling. Scheduling is already the most time-consuming phase, and this would make it much worse. No product compiler for a significantly parallel ILP processor does this, as far as we are aware. However, if compile-time is not an issue (a luxury that we have never faced) one could entertain modifying the computation graph during scheduling.

3.2 A property of correctly generated code

If the code that results from code generation is correct, it must satisfy a number of properties. There is one property that is particularly germane to our discussion. Assume that specific opcodes and registers have been selected for the operators and variables, respectively, in the computation graph. Then, for all the operators that either source or sink a particular variable, their selected opcodes must all have access to the register that has been selected for that variable. If not, the "connectedness" of the computation graph has been violated. We shall formalize this property shortly, but we first need to define a few terms.

> **Definitions:** The *expanded computation graph*, corresponding to a computation graph, is a directed, bipartite graph in which for each vertex in the computation graph, there is a corresponding set of vertices in the expanded computation graph. Each such vertex set represents the generic opcode set or the generic register set

corresponding to an operator vertex or a variable vertex, respectively, in the computation graph. Two generic sets are *adjacent* in the expanded computation graph if the corresponding vertices in the computation graph are connected by an edge. Every pair of adjacent generic sets in the expanded computation graph has a set of edges between them, one edge between each opcode and each of the registers that it can access as a particular operand. The direction of the edge is the same as that between the corresponding vertices in the computation graph.

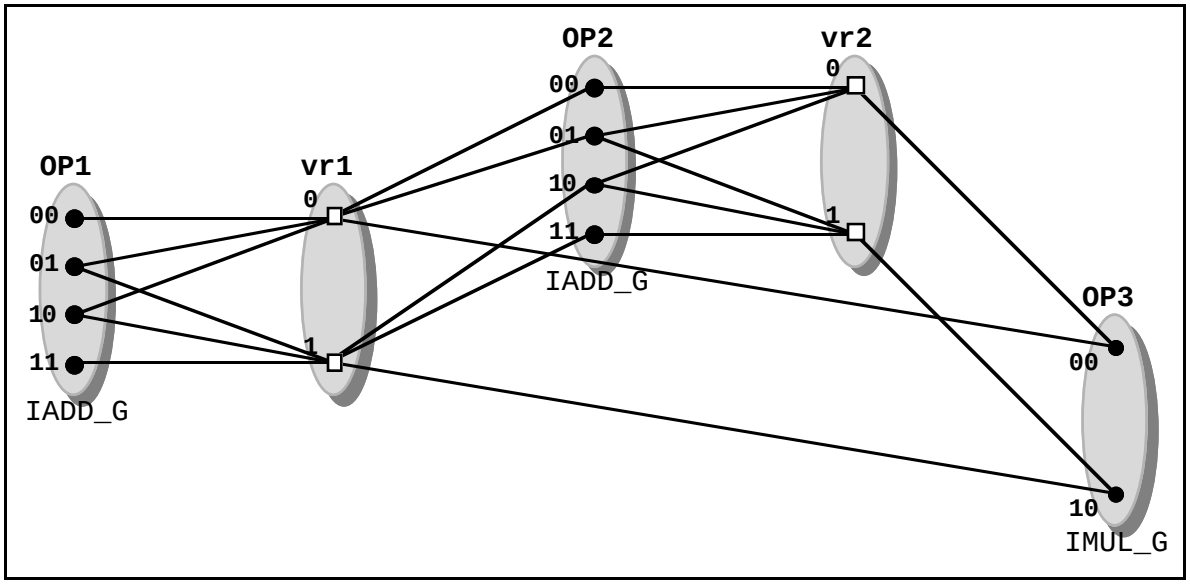

Figure 4: The expanded computation graph for the computation graph in Figure 2, considering only two of the variables, vr1 and vr2. The four opcodes corresponding to an IADD_G are IADD_00, IADD_01, IADD_10 and IADD_11, the two opcodes for IMUL_G are IMUL_00 and IMUL_10, and the two sets of registers for the variables are I0 and I1 (in each case, reading from top to bottom).

The space of all correct binding decisions is best illustrated using an expanded computation graph. For each operator or variable in Figure 2, Figure 4 contains its generic opcode set or generic register set, respectively. Each such set is represented by a gray ellipse containing black circles (opcodes) or hollow squares (registers). The flow of data along the edges is from left to right. For the sake of convenience, we take two liberties in Figure 4. First, as in Figure 3, we ignore all but two of the variables (vr1 and vr2). Second, in the two generic register sets, we lump together all of the registers which are part of the same register file and represent them by a single hollow rectangle. An edge to a register file is to be viewed as graphical shorthand for an edge to every register in that register file. For each

opcode in the expanded computation graph, the set of registers that it can access, and to which it has an edge is specified by Table 1.

The code generator's task is to select exactly one opcode or register from each generic set. Correctness requires that the following property exist after code generation has taken place.

> **Connectedness property:** For every opcode option selected by the code generator, all of the selected register options in the adjacent generic register sets must each be connected by an edge to the opcode option, and they must jointly constitute a valid register tuple for that opcode.

For the tiny example in Figure 2, one could exhaustively enumerate all of the correct bindings for the three operators and the two variables by analyzing Figure 4. Every one of these would represent code that possesses the connectedness property.

Table 2: The set of correct bindings for the code fragment in Figure 2, obtained by analyzing Figure 4.

OP1	vr1	OP2	vr2	OP3
IADD_00	I0	IADD_00	I0	IMUL_00
IADD_01	I0	IADD_00	I0	IMUL_00
IADD_10	I0	IADD_00	I0	IMUL_00
IADD_00	I0	IADD_01	I0	IMUL_00
IADD_01	I0	IADD_01	I0	IMUL_00
IADD_10	I0	IADD_01	I0	IMUL_00
IADD_01	I1	IADD_10	I1	IMUL_10
IADD_10	I1	IADD_10	I1	IMUL_10
IADD_11	I1	IADD_10	I1	IMUL_10
IADD_01	I1	IADD_11	I1	IMUL_10
IADD_10	I1	IADD_11	I1	IMUL_10
IADD_11	I1	IADD_11	I1	IMUL_10

3.3 Constraining the scheduler's and register allocator's options

As the example in Figure 3b demonstrates, one cannot arbitrarily choose the opcode or register for an operator or variable, respectively, from their respective generic sets without running the risk of getting incorrect code. We need to constrain the choice of opcodes that

are available to the scheduler for each operator and the choice of registers that are available to the register allocator for each variable so that the resulting bindings will always possess the connectedness property. We refer to these constrained sets of choices as option sets, and define them as follows.

Definition: An *opcode option set* is a subset of the generic opcode set associated with an operator in a computation graph, with the intent that the opcode selected for this operator be a member of this subset.

Definition: A *register option set* is a subset of the generic register set associated with a variable in a computation graph, with the intent that the register selected for this variable be a member of this subset.

We wish to assign option sets to the operators and variables of a computation graph, prior to scheduling and register allocation, in such a way that the resulting code possesses the connectedness property

- regardless of which opcode the scheduler independently selects for each operator from its opcode option set,
- regardless of which register the register allocator independently selects for each variable from its register option set, and
- without the insertion of any copy operators during or after scheduling and register allocation,
- while keeping the option sets as large as possible.

Thus we preserve the efficiency of the scheduler and register allocator while retaining as much flexibility as possible for the two of them. With respect to our example code fragment, we would like to start off with the largest number of options for each operator and variable such that regardless of which option is independently selected for each one, the result would be one of the correct bindings listed in Table 2.

The ability to independently choose the option for each operator and variable and yet be certain that the resulting code will possess the connectedness property requires that the option sets associated with the vertices of the computation graph possess the following property.

Full-connectedness property: For every pair of adjacent vertices in the computation graph, the corresponding pair of option sets must be fully connected in

the expanded computation graph, i.e., every option in one option set is connected to every option in the other option set. Furthermore, for every opcode in an opcode option set, every register tuple in the Cartesian product of the adjacent register option sets must be legal for that opcode.

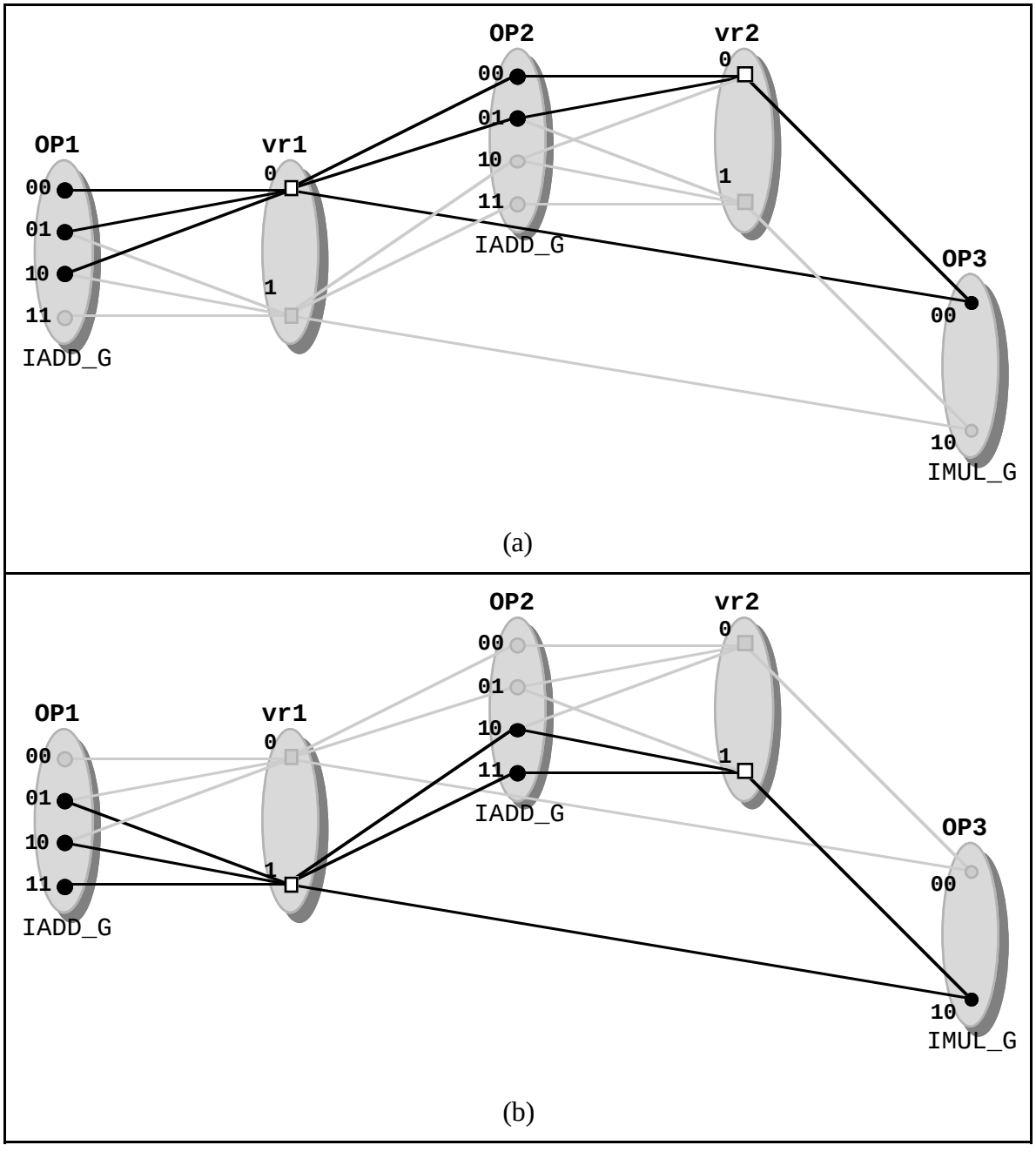

Figure 5: The two maximal, fully-connected sub-graphs of the expanded computation graph in Figure 4.

If this is the case, then regardless of which option is selected out of each option set, the resulting code will necessarily possess the connectedness property. The desire to maximize the number of options available to the scheduler and register allocator means that we want these fully connected option sets to be maximal, i.e., it should not be possible to add an option to any of the option sets while still maintaining full-connectedness.

Figure 5 displays the two maximal, fully-connected sub-graphs of the expanded computation graph in Figure 4. In each case, OP1 has three options, OP2 has two options, and OP3, vr1 and vr2 have one option each. Note that adjacent option sets are completely connected and that no option set can be augmented without violating the property of full-connectedness. In this example the edges of the two sub-graphs are disjoint, but this need not necessarily be the case.

The full-connectedness property as stated above can be understood as a specification of the problem of finding all possible, maximal, fully-connected sub-graphs from the expanded computation graph. Each such sub-graph identifies a choice of maximal option sets for all the vertices of the given computation graph. Clearly, the naive algorithm of first listing all possible combinations, as in Table 2, and then collapsing them into maximal sets would be exponential. We do not take any algorithmic or heuristic position on how this is best done.

Table 3: The two 5-tuples of maximal, fully-connected option sets for the vertices of the computation graph in Figure 2.

OP1	vr1	OP2	vr2	OP3
{IADD_00,IADD_01,IADD_10}	{I0}	{IADD_00,IADD_01}	{I0}	{IMUL_00}
{IADD_01,IADD_10,IADD_11}	{I1}	{IADD_10,IADD_11}	{I1}	{IMUL_10}

Table 3 lists the two maximal, fully-connected sub-graphs in the form of 5-tuples of maximal option sets. In general, a given correct choice of options could be part of more than one of these tuples of maximal option sets. Also, due to the full connectivity between adjacent option sets, options can be removed from one or more option sets and, as long as no option set becomes empty, the resulting binding is still guaranteed to possess the connectedness property. Thus, for the five vertices in our example computation graph fragment, the two tuples in Table 3 completely specify all possible choices of option sets which jointly possess the full-connectedness property. Naturally, to maximize the degrees

of freedom for the code generator, the preferred choice of option sets is one of the two maximal sets in Table 3. After selecting one of these, scheduling and register allocation can proceed.

Although these are the only 5-tuples of maximal option sets for the computation graph as shown in Figure 2, different tuples are obtained by modifying the computation graph, for instance, by inserting a copy operator as in Figure 3c, and these might well be more attractive. In keeping with our philosophy, this copy insertion must happen before invoking the scheduler. After copy insertion, the maximal option set tuples would have to be computed as described above, followed by scheduling and register allocation.

3.4 Access-equivalent option sets

In practice, the number of operator and variable vertices in the computation graph can be very large. All of these vertices must be decorated with option sets which jointly possess the full-connectedness property. Ideally, these option sets should be maximal as well. For large computation graphs, and when the processor has a rather irregular connectivity between FUs and register files, computing a tuple of maximal option sets is expensive; there can be very many of them, and even defining a local metric for selecting the best one—one that is correlated with eventually getting a good schedule—can be difficult. Although the concept of a tuple of maximal option sets is useful in understanding what we would like to achieve, it has little value as part of a constructive procedure that an EPIC compiler can use. The compiler needs a practical method of constructing a tuple of fully-connected option sets, and the machine description database must contain supporting information that can be computed off-line rather than during code generation.

We do so by focusing on smaller, modular portions of the computation graph. Instead of considering the computation graph as a whole, we consider overlapping sub-graphs, each of which correspond to a single operation: one operator vertex and the variable vertices adjacent to it. Corresponding to each of these sub-graphs is an expanded computation graph, which we term a *generic operation set*, i.e., a generic opcode set for the operator and, for each of its operands, the appropriate generic register set.

Figure 6 shows the expanded computation graphs for IADD_G and IMUL_G. Note that each of these graphs is completely specified by Table 1. Each generic operation set can be analyzed at the time that the mdes is constructed and the set of tuples of maximal option sets

can be constructed for each generic operation set, individually. These are listed in Table 4 for IADD_G, IMUL_G and IMOV_G.

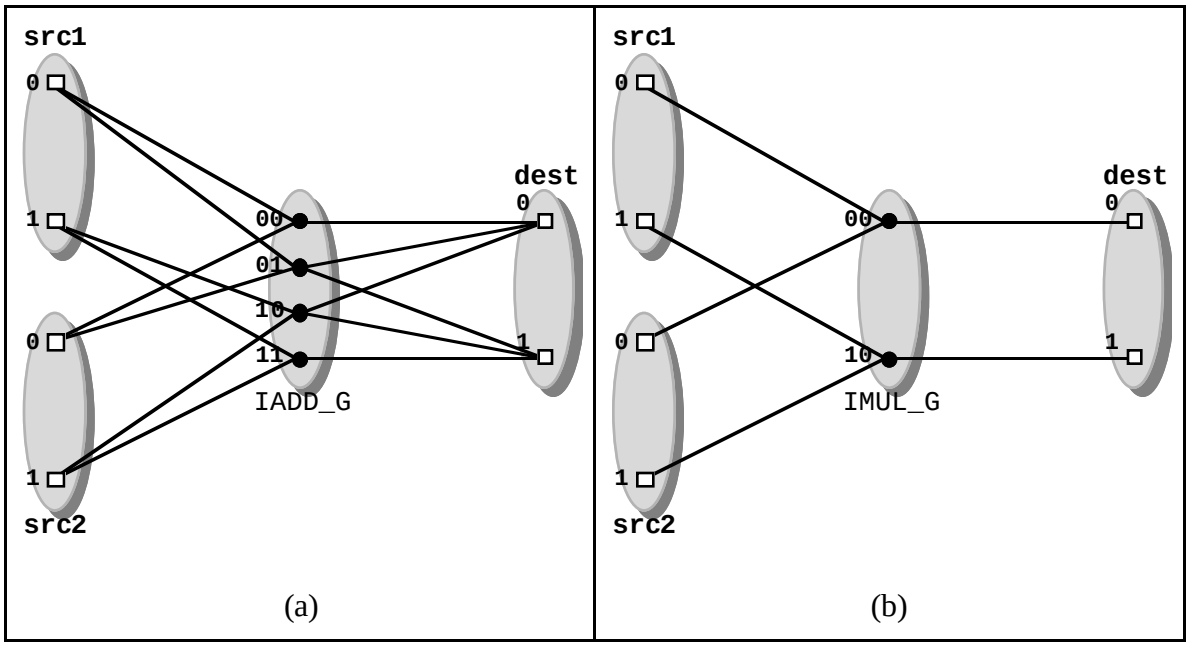

Figure 6: The expanded computation graphs for the (a) the IADD_G operation and (b) the IMUL_G operation.

Table 4: Access-equivalent operation sets, i.e., maximal mutually access-equivalent sets of opcode option sets and register option sets. ({I0} and {I1} each represent a set of registers.)

Semantics	Opcode	Source 1	Source 2	Destination
Integer Add	{IADD_00,IADD_01}	{I0}	{I0}	{I0}
	{IADD_01}	{I0}	{I0}	{I0,I1}
	{IADD_10}	{I1}	{I1}	{I0,I1}
	{IADD_10,IADD_11}	{I1}	{I1}	{I1}
Integer Multiply	{IMUL_00}	{I0}	{I0}	{I0}
	{IMUL_10}	{I1}	{I1}	{I1}
Copy	{IMOV_01}	{I0}	-	{I0,I1}
	{IMOV_10}	{I1}	-	{I0,I1}

Each tuple of maximal option sets is termed an *access-equivalent operation set*. Each one represents a set of legal, completely bound operations. These sets are not necessarily disjoint. Each of the constructed option sets is termed an *access-equivalent opcode set* or an *access-equivalent register set* (as the case may be). For any given access-equivalent operation set, every register in the access-equivalent register set corresponding to a particular operand is equally accessible by any opcode in the access-equivalent opcode set. Conversely, for every opcode in the access-equivalent opcode set, that opcode can access any register in the access-equivalent register set corresponding to a particular operand, and every register tuple in the Cartesian product of these access-equivalent register sets is legal with respect to that opcode. These option sets are mutually access-equivalent. Again, note that one can subset one or more of the options sets in an access-equivalent operation set to any non-empty subset of the maximal set, without loosing the property of mutual access-equivalence.

The same algorithm, that might have been used to derive the fully-connected, maximal sub-graphs of the expanded computation graph of a program, can be applied to the expanded computation graphs of each individual generic operation set to yield the access-equivalent operation sets. This problem is substantially easier since each such modular expanded computation graph is much smaller in size and even exponential enumeration and grouping may not be impractical.

The significance of Table 4 is that these access-equivalent operation sets are a property of the machine alone and, unlike Table 3, not of the computation graph as well. They can be computed once and for all from Table 1 while constructing the mdes. Thereafter, for any given computation graph, they can be used to annotate each operator and variable with an access-equivalent option set, using some form of constraint-satisfaction process. The constraint that must be satisfied is that each operation in the computation graph has to be annotated with an access-equivalent operation set in such a way that, for every variable, the intersection of the access-equivalent register sets imposed upon it by all of the operations that access it (i.e., its access-equivalent register option set), must be non-empty. If this constraint is satisfied, the full-connectedness property holds. This annotation and partial binding step is termed the *pre-pass operation binding* phase.

Consider once again the example in Figure 2. OP3 has a choice of two access-equivalent opcode sets, {IMUL_00} and {IMUL_10}, each of which contains only a single option. Let us assume that {IMUL_10} is selected. This restricts the options sets for both vr1 and vr2 to {I1}. This, in turn, restricts the possible option sets for OP2 to be either

{IADD_10,IADD_11} or {IADD_10}. The former is clearly preferable since it contains more options. Since vr1 has the option set {I1}, OP1 can be bound to either {IADD_10,IADD_11} or to {IADD_01}. Perhaps for the purposes of load balancing across the FUs, the partitioning algorithm picks the latter. Now that every vertex in the computation graph has been decorated with an option set, scheduling can be performed and might result in the code shown in Figure 3a.

In general, there are many correct annotations for the computation graph and, for a processor with an arbitrary topology, the task of obtaining a good annotation could still be very complex. Consequently, there is a strong incentive to design processors with a relatively simple structure. Typically, each opcode can only access a single register file, except for copy opcodes that source one register file and sink a different one. Our example processor is not far removed from this design style. For such processors, once the desired copy operators have been inserted by the partitioning algorithm, the process of assigning access-equivalent option sets is trivial. The real problem lies in deciding where to insert copy operators so that the computation is equitably distributed over the FUs of the machine [34, 29, 35, 36].

One way of characterizing traditional DSPs in the context of our framework is that by the time pre-pass operation binding has been performed, the opcode and register option sets have dwindled down to such an extent that there is typically just one option per option set. The scheduling and register allocation that follow are largely trivialized. All the scheduler can do is "compaction", i.e., selecting the time at which each operation is performed within the constraints imposed by the lack of opcode options and the anti- and output dependences introduced by the register binding. Thus, for DSPs, the pre-pass operation binding phase is the crucial one, whereas for the EPIC processor the most important part of code generation is yet to come. It has been consciously designed in such a way as to ensure plenty of options at this stage of binding. Because of the large number of register options for each operand, the scheduler has the luxury of acting as if there is an unlimited number of register options available, with register spill being handled as an exceptional circumstance.

An issue that attracts a lot of attention in the DSP literature is that of making good use of address generators [6, 5]. To our way of thinking, an address generator is an adder, one of whose inputs (the array address) is from the same register file to which it writes its result (which is also sent to the memory as the address for the load or store). The other input (the address increment) is from a different register file. None of this causes any great difficulty with our model of compilation. Nor does it cause any problems with the style of machine

description that we shall propose. What does constitute a problem is that each of these register files typically has just a single register. If not for this, making use of address generation hardware would not constitute any particular problem. In fact, the Cydra 5 mini-supercomputer had what might be viewed as an address generator, except that it was designed in such as way, with an adequate number of registers in the address register file, as to lie within our space of EPIC processors [37].

The access-equivalent operation sets, as defined in this section are applicable across all phase orderings that might be imposed upon the rest of the code generator; they are unbiased with respect to the phase order. For any given phase ordering, the definition of access-equivalent operation sets may change (as we shall see in the next section). But, in every case, the resulting access-equivalent operation sets must, necessarily, be a subset of those defined in this section (since these sets are maximal).

4. A model of EPIC code generation

The last section discussed the major challenges of EPIC code generation and argued that, for reasons of efficiency, the code generation process for an EPIC processor must use a phased approach in mapping the source program's operations to the processor's architectural operations. We concluded that a pre-pass operation binding phase, which binds operations to access-equivalent operation sets, was a practical necessity. However, the detailed phase ordering used to accomplish scheduling and register allocation was left unspecified.

For any EPIC processor with even a reasonable amount of parallelism, register allocation is never performed before scheduling. If it were, schedule-unaware register allocation would impose crippling and unnecessary restrictions upon the scheduler in the form of anti- and output dependences caused by using the same register for an unfortunate choice of variables. This leaves one with two choices. Either scheduling can be performed before register allocation, or the two can be performed simultaneously. Each one has its advantages and drawbacks.

In this section, we look at how the incremental binding process and even the definition of an access-equivalent operation set are affected by the specific choice of phase ordering for scheduling and the register allocation. In particular, we consider the example of the phase

ordering used by Elcor, our research compiler for EPIC processors. Its phase order is:

1. code selection
2. pre-pass operation binding (including partitioning, if needed)
3. scheduling
4. register allocation and spill code insertion
5. post-pass scheduling
6. code emission

Each phase successively refines and narrows down the options available for either the opcodes, the registers, or both, finally yielding architectural operations that can be executed by the processor. Optimization phases may be inserted at various points in this sequence but, since they do not affect the level of binding, we ignore them. This section introduces a hierarchy of operation refinements between the semantic and architectural layers that forms the basis for organizing the machine description database for the use of the code generator.

Table 5: Access-equivalent operation sets in the context of our phase ordering. ({I0} and {I1} each represent a set of registers.)

Semantics	Opcode	Source 1	Source 2	Destination
Integer Add	{IADD_00,IADD_01}	{I0}	{I0}	{I0}
	{IADD_01}	{I0}	{I0}	{I1}
	{IADD_10}	{I1}	{I1}	{I1}
	{IADD_10,IADD_11}	{I1}	{I1}	{I1}
Integer Multiply	{IMUL_00}	{I0}	{I0}	{I0}
	{IMUL_10}	{I1}	{I1}	{I1}
Copy	{IMOV_01}	{I0}	-	{I0}
	{IMOV_01}	{I0}	-	{I1}
	{IMOV_10}	{I1}	-	{I1}
	{IMOV_10}	{I1}	-	{I0}

Our focus, in the rest of this report, is on steps 3 through 5. Each of these steps imposes a pre-condition upon the state of binding that affects our definitions of access-equivalence.

Just prior to register allocation, the scheduler has bound the opcode that is to be used by each operation, as well as when it is scheduled, thereby specifying which functional unit and resources will be used, and when. The operands, however, have only been bound to access-equivalent register sets. It is necessary that the choices made by the register allocator not invalidate the schedule in any manner.

This imposes a new constraint upon the definition of access-equivalent register sets, which tends to shrink them relative to those defined in the previous section. This, in turn, changes the definition of the access-equivalent operation sets that are acceptable just prior to scheduling. Table 5 shows all of the (maximal) access-equivalent operation sets in the context of our phase ordering. The rest of this section explains the reasoning behind this re-definition of access-equivalent operation sets as well as the process of incremental binding that results from our phase ordering. For a different phase ordering, the methodology presented below could be used to derive the corresponding access-equivalent operation sets.

4.1 Code generation: semantic operations to architectural operations

Semantic operations and architectural operations form the two end points of the EPIC code generation process. In this section, we describe those properties of these two types of operations that are relevant to our discussion in the subsequent sections.

Semantic operations. These are the operations in the input to the code generation process, and they correspond to actions performed by a pre-defined virtual machine. The opcodes at this stage of compilation are the semantic opcodes provided by the pre-defined virtual machine. An operand is either a constant (such as 3 or 4.1) or a *virtual register* (VR). The following two attributes of a VR are relevant to the discussion of the code generation process.

1. Data type, *e.g.*, int, long, float, double. The data type determines the number of bits in the container that is needed to store the value.
2. Simple VR or an element of an *expanded virtual register* (EVR) [38]. Although EVRs are not part of any language, some of the machine-independent loop-transformations preceding the code generation may introduce them into the code. EVRs either can be mapped to rotating registers [13, 15] or can be converted to simple VRs by code transformation (*e.g.*, loop-unrolling) [39].

Architectural operations. These are the operations in the output of the code generation process, and they represent commands performed by the target machine. The opcodes at this stage are architectural opcodes available in the target machine. To simplify the exposition, we treat any addressing modes supported by the target machine as part of the opcode rather than part of an operand. An operand is an *architectural register* which is either a literal or a register in the target machine. For uniformity, we model a literal as the contents of a read-only "literal register". Thus an architectural register is either a real machine register or a literal register. Architectural registers are characterized by the following two types of attributes.

- **Storage attributes:** There are five storage attributes associated with an architectural register.
  - The bit width of the register. This is the number of bits in the container.
  - The presence or absence of a speculative tag bit. Architectures that support compile-time speculative code motion (control, data or both) provide an additional speculative tag bit in each register that is intended to participate in the speculative execution. This is provided for the correct handling of architecturally-visible exceptions [40, 41, 15].
  - Whether this is part of a static or rotating architectural register file. This determines whether the register specifier used for this register is an absolute register address or an offset from the rotating register base [15].
  - Whether the register is read/write (the normal case), read-only (a literal) or write-only (the proverbial "bit bucket", which is often implemented as GPR 0 in a number of architectures).
  - For a read-only register, the value of the literal.

  Note that the presence or absence of the speculative tag bit and the static/rotating classification make sense only for "normal" architectural registers and not for literals or the bit bucket.
- **Accessibility from opcodes:** This consists of the set of architectural opcodes that can use the given architectural register as an operand, and for each opcode, the operand positions in which that register can appear as an operand. This is the same as specifying the connectivity between functional units and register files in a structural description of the target machine.

4.2 The register binding hierarchy

The process of binding generic operations to architectural operations is based on two refinement hierarchies, one for opcodes and one for registers. We describe the additional register layers in this section and do the same for opcodes in the next section.

The three additional layers that an EPIC compiler needs, to translate VRs to architectural registers, are described below. Note that the order in which they appear is not the order in which they are used to successively refine virtual registers.

Compiler-registers. A *compiler-register* is either a single architectural register or a set of architectural registers, with a fixed spatial relationship, that are viewed as a single entity by the compiler. Although, in principle, an architecture could view any arbitrary set of registers as a single entity, we believe that such a view is too general and not very useful. Thus all architectural registers that are part of a compiler-register must be identical with respect to all of their characterizing attributes, *i.e.*, storage attributes and the accessibility from opcodes. A compiler-register is characterized by the same types of attributes as is an architectural register. The width of a compiler-register is a function of the width of constituent architectural registers. The other attributes simply carry over.

It is important to note that an architectural register can be part of more than one compiler-register. For example, some architectures view an even-odd pair of single-precision floating-point registers as a double-precision register. In that case, a single-precision architectural register is both a single-precision compiler-register and an element of a double-precision compiler-register.

Compiler-registers also provide a convenient abstraction for compiler and OS conventions such as the procedure calling convention. For example, most phases of the compiler can work with compiler-registers such as the stack pointer, the frame pointer, and parameter passing registers without worrying about the specific architectural registers reserved for these purposes; only the register allocator need be aware of the exact correspondence.

Note that the translation from compiler-registers to architectural registers is primarily a book-keeping step as there are no decisions to be made.

Generic register sets. A generic register set is a maximal set of compiler-registers that have the same storage attributes.

Generic register sets provide the first layer in translating operands in the semantic layer, *i.e.*, VRs and constants, to architectural registers. They provide a useful abstraction, since they focus on the properties that are relevant to the abstract computation model and ignore details such as the physical connectivity of the target machine. To map a VR to a generic register set, the data type of a VR is used to select all the generic register sets with the appropriate width. If loop unrolling is precluded, an element of an EVR can be mapped only to those generic register sets that have rotating registers. This restricts the set of candidate generic register sets. A simple VR, on the other hand, can be mapped to any one of the selected generic register sets. The default position is that a VR can map to a register whether or not it has the speculative tag bit (i.e., the VR does not need it). Subsequent code transformations may determine that the VR needs a generic register set with speculative tag bit, which would then restrict the candidate generic register sets further. If at the end of this process there are multiple candidate generic register sets, the VR can be mapped to any one of them using an appropriate heuristic. Constants are mapped to generic register sets corresponding to the literals provided by the target machine.

Access-equivalent register sets. In order to help us redefine an access-equivalent register set, we first define a few other terms. An *alternative* is a triple consisting of a compiler-opcode (as defined in Section 4.3), a latency descriptor and a reservation table[7], that are jointly valid for the target processor. A *register set tuple (RS-tuple)* is a tuple of register sets, such that each register set is a subset of a single generic register set (*i.e.*, all the registers have the same storage attributes). An *access-equivalent RS-tuple* corresponding to a given alternative is a maximal RS-tuple, where each register set corresponds to one of the operands of the compiler-opcode, and every register tuple in the Cartesian product of the register sets is jointly valid with that alternative, taking into account both the connectivity constraints of the processor as well as the instruction format constraints. Each register set in an access-equivalent RS-tuple is an *access-equivalent register set.*

For every choice of register tuple in the access-equivalent RS-tuple, along with the compiler-opcode of the alternative, the resulting operation has the same latency descriptor and the same reservation table, since all the register tuples are accessible with the same

[7] Latency descriptors and reservation tables are discussed in Section 5. For now, it suffices to say that a latency descriptor provides all the information needed by the scheduler regarding an operation's latencies, and a reservation table describes its resource usage over time.

alternative[8]. Consequently, each access-equivalent register set contains registers that are interchangeable with respect to that opcode after scheduling has taken place; any register can be used in place of any other without any impact on the correctness of a scheduled piece of code. Furthermore, since all the register tuples implied by an access-equivalent RS-tuple are architecturally valid, the compiler-register for each operand can be independently selected by the register allocator.

Purely to help in explaining this definition of access-equivalent register sets, we outline the following conceptual procedure to find all of them. (This is not necessarily a practical approach.) We first consider the set of all legal compiler-register tuples, L, for a given alternative. Next, we find all maximal compiler-register set tuples, T, such that each of the compiler-register sets in T is a subset of a single generic register set, and such that the set of register tuples that results from forming the Cartesian product of the compiler-register sets in T, is a subset of L[9]. Each T is an access-equivalent RS-tuple, and each of the compiler-register sets in T is an access-equivalent register set for the given alternative. Finally, we repeat the entire process for every possible alternative in the target processor.

Since all compiler-registers in an access-equivalent set have identical storage attributes, we can also associate these attributes with the set itself. The hardware notion of a register file is an example of an access-equivalent register set.

Access-equivalent register sets provide the second layer in translating VRs to architectural registers. The refinement of an access-equivalent set to a particular compiler-register in the set is done during the register allocation phase.

We refer back to Table 4 to see how our phase ordering affects the definition of access-equivalent register sets for our example processor. Consider the second access-equivalent operation set for integer add. Different resources (register write ports) are used depending

---

[8] Note that in the event that a particular RS-tuple happens to be the access-equivalent RS-tuple with respect to two compiler-opcodes, the two compiler-opcodes can have different latencies or resource usage when accessing this access-equivalent RS-tuples. For example, assume there are two functional units FU1 and FU2, which can each perform an add using opcodes ADD1 and ADD2, respectively, using the same register file. Assume that FU1 uses a different set of shared buses to access the register file than does FU2. Now, ADD1 and ADD2 both have the property that the choice of register that they access does not affect the resources (buses) used. Therefore, the registers in the register file constitute an access-equivalent register set with respect to both opcodes. Nevertheless, ADD1 uses different resources when accessing this access-equivalent register set than does ADD2.

[9] Clearly, the Cartesian product cannot be allowed to be a superset of L, since this would mean that we are admitting spurious, illegal register tuples. If the Cartesian product is equal to L, then a single access-equivalent RS-tuple suffices to specify L. On the other hand, assume that every Cartesian product, which is not a superset of L, is a proper subset of L. Since, by definition, each of the register sets that form the Cartesian products is maximal, we shall need multiple access-equivalent RS-tuples to cover L.

on whether the destination is I0 or I1. So, this corresponds to two different reservation tables and two distinct alternatives. Consequently, the access-equivalent RS-tuple has to be split into two. The first one is identical to that for the first access-equivalent operation set in Table 4. (The resulting access-equivalent operation set is subsumed by the first one, which has a larger opcode set.) The second one is <{I0}, {I0}, {I1}>. Thus {I0, I1} is no longer an access-equivalent register set. The access-equivalent RS-tuples and register sets, in the context of our phase ordering, are shown in Table 5.

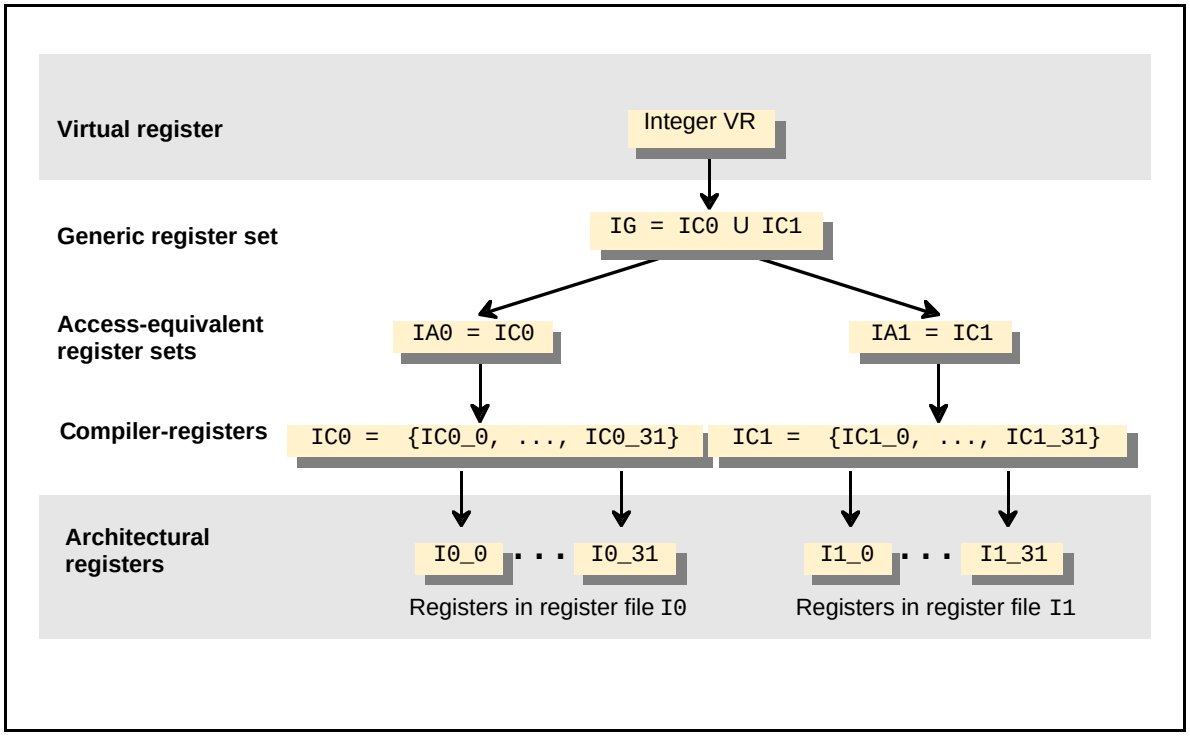

Figure 7: The register binding hierarchy for the integer registers in the example machine introduced in Section 2. Virtual register and architectural register (shown shaded) are not part of the binding hierarchy. They are included to show implementation relations; that is, virtual registers are implemented using generic register sets, and compiler-registers are implemented using architectural registers.

Figure 7 shows the binding hierarchy for integer registers in the context of our phase ordering. At the lowest level are architectural registers in the two register files I0 and I1; at the highest level is a VR with integer data type. These two levels are not part of the refinement hierarchy; they are used to show "implementation relationships". Architectural registers are used to implement compiler-registers, and in this case, they have a one-to-one relationship with compiler-registers. Since all these compiler-registers have the same

storage attributes, they are grouped into one generic register set denoted by IG in the figure. This example shows a trivial implementation relation between the integer VR and the generic register set, but that need not always be the case. As shown in Table 5, with our refined definitions of access-equivalent register sets, there are now only two distinct architectural register sets, I0 and I1, that are accessed by various alternatives corresponding to the integer add, multiply and move opcodes. The architectural register sets, I0 and I1, expressed in terms of compiler-registers form the two access-equivalent register sets, IA0 and IA1, respectively.

4.3 The opcode binding hierarchy

The binding process for opcodes parallels that for registers. The three additional opcode layers that an EPIC compiler needs are described below. As in the case of registers, the order in which they appear is not the order in which they are used to successively refine semantic opcodes.

Compiler-opcodes. In some cases, architectural opcodes don't provide the right abstraction that the compiler, especially the scheduler, needs. As a consequence, either the scheduler has less freedom during scheduling, or it has to use a complex scheduling algorithm. Consider, for example, a target machine that doesn't provide an explicit register-to-register copy and in which the copying of a register is done, instead, by either adding 0 or multiplying by 1. We must be able to describe that a copy operation can be implemented in one of these two ways so that the scheduler is free to choose either one based on the availability of resources. In addition, compiler algorithms, such as copy-propagation, would prefer to deal with the abstract notion of a register-to-register copy rather than all possible ways in which it can be coded for the target machine. Thus, we need a capability to describe versions of architectural opcodes in which some of the operands have been pre-bound to literals.

Another example is a compiler-opcode whose semantics specify a 64-bit add on a machine that only has 16-bit adds. The 64-bit add can be implemented as four 16-bit adds with the carry-out of one add feeding the carry-in of the next. Since there is only a single carry bit per adder, the four 16-bit adds must be constrained to execute on the same adder in successive cycles (assuming a single cycle add latency).

As a final example, consider a target machine which provides register-to-register add and multiply operations. In addition, suppose that it provides a fast way of doing a multiply-

add by “chaining” the two units, without going through a register, provided that the add is scheduled in the cycle after the multiply. Such constraints, which are necessary to handle transient data, increase the complexity of the scheduling algorithm and are one of the reasons why it is hard to schedule for DSPs. However, if such constraints are rare, a simple way to handle them is to represent the two operations, with a fixed relative schedule, by one multiply-add compiler-opcode.

Thus, we introduce an abstraction over architectural opcodes, called compiler-opcodes. A compiler-opcode is implemented by one or more architectural opcodes as specified by a *bundle-macro*, which is described in detail in Section 5.2. As with registers, the translation from compiler-opcodes to architectural opcodes is essentially a book-keeping step; there are no decisions to be made. The compiler intermediate representation with compiler-opcodes is translated to architectural opcodes using the associated bundle-macro definition. This step constitutes a form of delayed code selection.

Generic opcode sets. A generic opcode set is the maximal set of compiler-opcodes that implement the same function, *e.g.*, integer add. In other words, the compiler-opcodes in such a set are interchangeable as far as their semantics are concerned, and the function being implemented can be materialized using any one of the compiler-opcodes.

These sets provide the first layer in translating semantic opcodes to architectural opcodes and provide a useful abstraction. During the code selection phase, they focus on the semantics of the operations available on the target machine and hide details such as the connectivity of functional units to register files.

Access-equivalent opcode sets. These sets are defined using the notion of access-equivalent register sets introduced in the last section. An access-equivalent opcode set is the maximal set of compiler-opcodes that are part of the same generic opcode set (*i.e.*, implement the same function) and for each of which there is an alternative that yields the same access-equivalent RS-tuple. Note that this definition permits each compiler-opcode to access the access-equivalent RS-tuple with a different latency descriptor or reservation table, but it does require that the RS-tuple be equally accessible by every compiler-opcode in the set. Therefore, these compiler-opcodes can be interchanged without having to insert or delete any copy operations.

Conceptually, to find all access-equivalent opcode sets, we consider all possible access-equivalent RS-tuples for the target processor that were discovered during the process of identifying the access-equivalent register sets. Each RS-tuple has one or more alternatives

with respect to which it is an access-equivalent RS-tuple, and each such alternative has an associated compiler-opcode. We partition these compiler-opcodes, grouping together those that are a subset of the same generic opcode set. Each of these partitions constitutes an access-equivalent opcode set.

Access-equivalent register sets, RS-tuples and opcode sets, as defined in this and the previous sections, provide maximal freedom to the scheduler while satisfying both requirements: that the computation graph need not change during scheduling, and that the choices available to the register allocator after scheduling are schedule-neutral and can be made considering each operand individually.

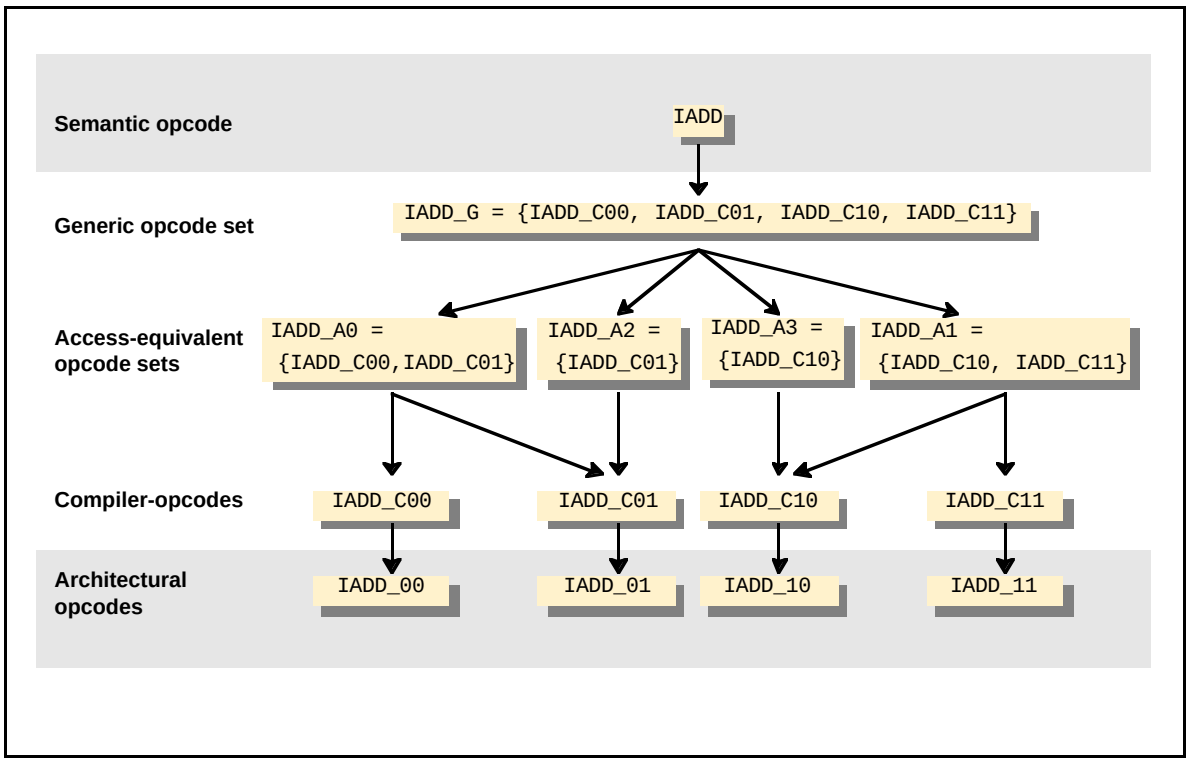

Figure 8: The opcode binding hierarchy for integer add operations in the example machine introduced in Section 2. Semantic opcode and architectural opcode (shown shaded) are not part of the binding hierarchy. They are included to show implementation relations; that is, semantic opcodes are implemented using generic opcode sets, and compiler-opcodes are implemented using architectural opcodes.

The effect of our phase ordering was to split some of the access-equivalent register sets in Table 4. As pointed out earlier, subsetting one or more access-equivalent register sets of an access-equivalent operation set yields another (non-maximal) access-equivalent operation

set with the same access-equivalent opcode set. Consequently, our phase ordering does not alter the definition of the access-equivalent opcode sets (Table 5).

Access-equivalent opcode sets form the second layer in the opcode binding hierarchy. Generic opcode sets, assigned to each operator node of the computation graph, are refined to access-equivalent opcode sets by the pre-pass operation binding phase before scheduling in order to provide maximal freedom to the scheduler while still ensuring the interchangeability of opcodes. The scheduler then chooses an alternative corresponding to a compiler-opcode from the access-equivalent opcode set based on the availability of resources.

Figure 8 shows the opcode binding hierarchy corresponding to the four architectural opcodes that perform integer add in our example processor, in the context of our phase ordering. At the lowest level are architectural opcodes IADD_00, IADD_01, IADD_10 and IADD_11; at the highest level is the semantic opcode denoted by IADD. As in the case of the register hierarchy, these two levels, shown shaded in the figure, are not part of the refinement hierarchy; instead, they are used to show "implementation relationships". Architectural opcodes are used to implement compiler-opcodes and, in this case, have a one-to-one relationship with compiler-opcodes. Since all these compiler-opcodes perform the same semantic operation, they are grouped in one generic opcode set, denoted by IADD_G in the figure. The generic opcode set is used to implement the semantic opcode IADD. This example shows a trivial relationship between the semantic opcode and the generic opcode set, but that is not always the case. There are four access-equivalent opcode sets, denoted by IADD_A0, IADD_A1, IADD_A2 and IADD_A3 in the figure. These are taken from Table 5 which lists all access-equivalent opcode sets for integer add in terms of architectural opcodes; the figure shows them in terms of the compiler-opcodes.

4.4 The operation binding lattice (OBL)

Since an operation is composed of an opcode and one or more operands, we can describe all possible ways of binding semantic operations to architectural operations as a binding matrix obtained by taking the cross-product of the steps used in binding opcodes with steps used in binding registers. Figure 9 shows the operation binding matrix. The columns correspond to the various levels in the opcode binding hierarchy described in Section 4.3. The first four columns correspond to the levels in the opcode hierarchy within the compiler, the last column represents the architectural opcodes in the target machine. The rows in the matrix correspond to the levels in the register binding hierarchy described in Section 4.2.

The first four rows correspond to the levels in the register binding hierarchy used within the compiler, the last row represents the architectural registers in the target machine.

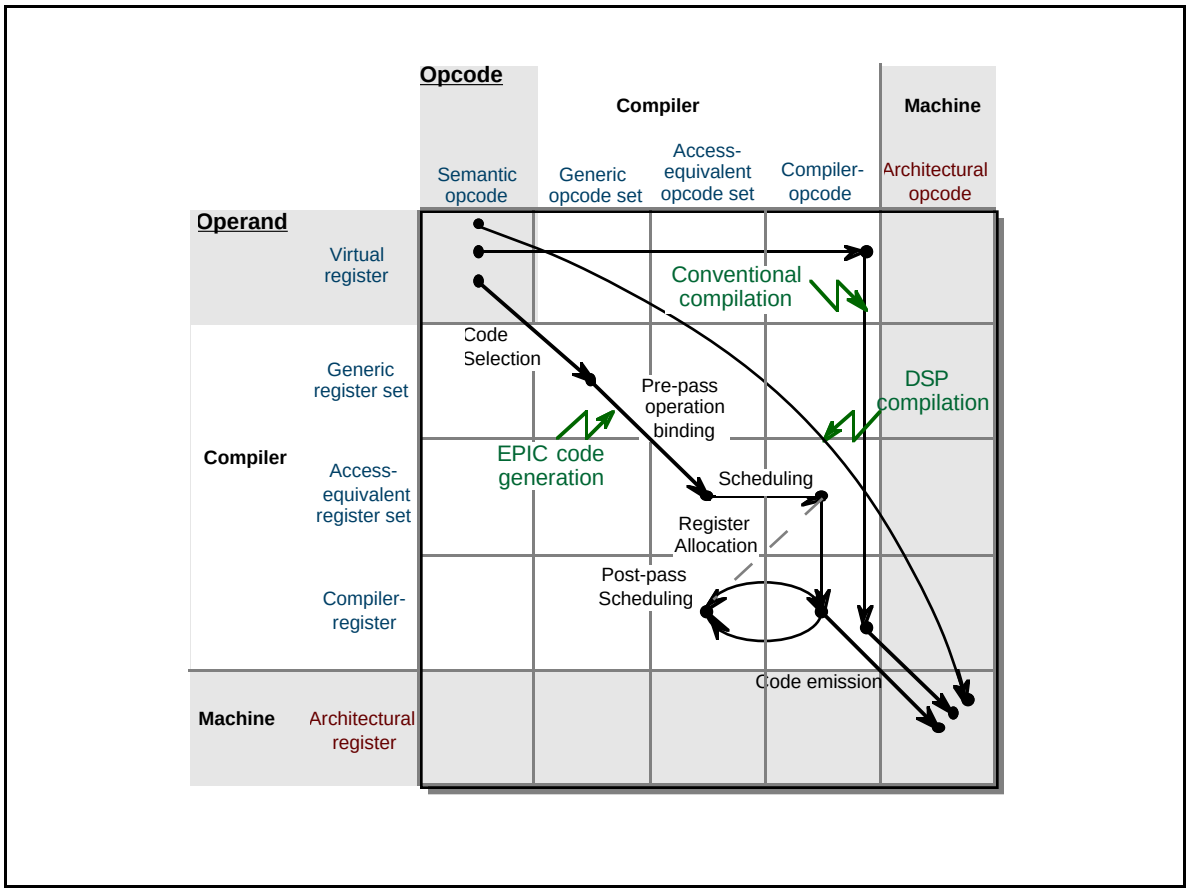

Figure 9: The operation binding matrix.

A left to right walk of a row in the matrix represents the steps used in binding semantic opcodes to architectural opcodes but keeping the operand binding fixed. Similarly, a top to bottom walk of a column represents the steps used in binding virtual registers to architectural registers but keeping the opcode binding fixed. Some compiler modules may refine the binding of both opcodes and operands simultaneously; this corresponds to moving diagonally in the matrix. Note that, in general, a binding decision made for either one (opcode or register) may restrict the choices available for the other. To simplify the exposition, our discussion as well as the matrix in Figure 9 assumes that all operands in a program follow the same binding path. Although this is typically the case, exceptions may

exist. For example, some compilers may bind constants to literal registers during code selection, and virtual registers to real architectural registers in the register allocation phase.

The top-left corner and the bottom-right corner in the matrix correspond to the two ends of the binding spectrum, and any path that connects these two end points defines a phase ordering that can be used to structure the code generation process. For most DSPs, it is hard to define intermediate layers of generic and access-equivalent sets (both opcode and operand) because of their highly specialized opcode repertoire and irregular connectivity. Thus, compilers for DSPs typically translate semantic operations to architectural operations in one step, *i.e.*, follow the path labeled "DSP compilation", and that is one of the reasons they are so hard to implement.

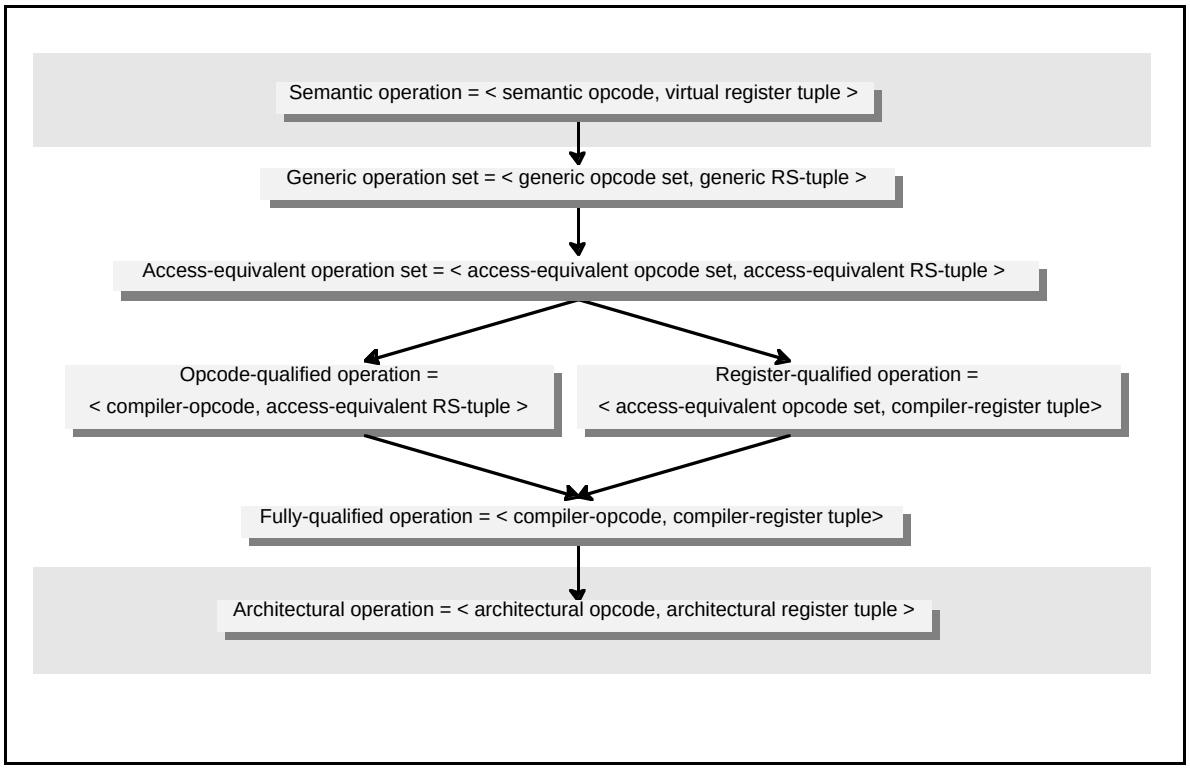

Figure 10: Operation binding lattice for our preferred phase ordering. Semantic and architectural operations are not part of the binding lattice. They are included to show implementation relationships; that is, semantic operations are implemented using generic operation sets, and fully-qualified operations are implemented using architectural operations.

Compilers for traditional RISC architectures follow the path labeled "conventional compilation". Semantic opcodes are mapped to the compiler abstraction of architectural

opcodes during the code selection phase. Scheduling of operations simply spreads them apart in time to honor latency constraints. Then the register allocation phase maps virtual registers to the compiler abstraction of architectural registers. Finally, the code emission phase converts the code to assembly code for the target machine.

In contrast to DSPs and traditional RISC architectures, the EPIC code generation process involves many more steps. In Figure 9, the path labeled “EPIC code generation” shows our preferred phase ordering for EPIC code generation. The preferred phase ordering induces a lattice of operation bindings, which we call the *operation binding lattice*, as shown in Figure 10. Note that semantic and architectural operations are shown in the figure, but they are not part of the lattice. They are used to show “implementation relationships”; semantic operations are implemented by generic operation sets and architectural operations implement fully-qualified operations. We briefly describe the various phases as they step through the operation binding lattice.

Code selection. The code selection phase maps semantic operations to generic operation sets, i.e., it maps semantic opcodes and virtual registers to generic opcode sets and generic register sets, respectively. Note that the mapping from semantic opcodes to generic opcodes is not, in general, one-to-one. A further point to note is that a phase, such as the code selection phase, may have internal sub-phases that correspond to moving horizontally or vertically in the binding matrix.

Pre-pass operation binding. If, at this point, the generic operation sets contain multiple access-equivalent operation sets, then the operations need to be further bound down to a single access-equivalent operation set. This is done by the pre-pass operation binding phase. A variety of heuristics, which are beyond the scope of this discussion, may be employed in so doing. In the case of a multi-cluster processor, one of these is a partitioning algorithm which attempts to distribute the given computation over the clusters. This may introduce copy operations which move data between register files in order to get a balanced distribution. Since the access-equivalent opcode and register sets are closely inter-related, the pre-pass operation binding phase partially binds both opcodes and registers simultaneously.

Scheduling. The scheduling phase is one of the main phases of an EPIC code generator. For each operation, the scheduler decides the time at which the operation is to be initiated. It also determines which compiler-opcode is to be used as well as the reservation table and latency descriptor that are used by the operation, i.e., it picks a specific alternative. In the

case of statically scheduled EPIC machines, the scheduling phase refines access-equivalent operation sets to *opcode-qualified operation sets, i.e.,* operations in which the possible alternatives have been narrowed down to a particular one, as a consequence of which the opcode options have been narrowed down to a single compiler-opcode, but with the register options having been narrowed down only to an access-equivalent RS-tuple[10].

Register allocation. The register allocation phase assigns a specific compiler-register to each of the variables in the computation graph by selecting one of the compiler-registers from the corresponding access-equivalent register set. This yields *fully-qualified operations,* i.e., a specific alternative and a specific compiler-register tuple.

The register allocation phase may introduce additional code to spill variables to memory. The spill code is fully-bound as far as the registers are concerned, but it has not been scheduled. Thus, after this phase, the program contains two types of operations. Firstly, it contains operations that have been narrowed down to fully-qualified operations. Secondly, it contains spill operations whose operands are fully bound to compiler-register tuples, but whose opcodes are still at the level of access-equivalent opcode sets. We call such operations *register-qualified operation sets.*

Post-pass scheduling. A second pass of scheduling, called post-pass scheduling, is necessary to schedule the spill code introduced by the register allocator. This phase has a choice with the fully-qualified operations: it can either keep the opcode bindings selected by the earlier scheduling phase or it can start afresh by reverting all compiler-opcodes back to their original access-equivalent opcode sets. Since the latter strategy gives more freedom to the scheduler in accommodating spill code and yields better schedules, we prefer it. Post-pass scheduling deals with code containing variables that are fully bound to compiler-registers. It is greatly constrained, therefore, by a host of anti- and output dependences. However, since the register assignments were made subsequent to the main scheduling phase, they are fully informed by, and are sensitive to achieving, a good schedule.

---

[10] In the case of statically scheduled EPIC machines, the binding by the scheduler, of an operation to an opcode-qualified operation set, fully specifies the latency parameters and resource usage. This is a necessary property since the register allocator has to be able to choose any register in the set without affecting the correctness of the schedule created by the scheduler. Thus, all choices have to be identical with respect to their latency parameters and resource usage. It is also important to note that this doesn't apply to dynamically scheduled machines or machines with dynamic arbitration for resources in which, regardless of what the scheduler decides, the hardware may cause an operation to use any one of many resources based on the resource availability at run-time, each one, possibly, with a different latency. However, since the hardware is responsible for preserving the semantic correctness of the program, this is not a problem.

Code emission. The final phase is the code-emission phase. This phase converts fully-qualified operations to architectural operations. This is a book-keeping step and no decisions are made by this phase.

5 Machine description database contents

Based on the model of EPIC code generation and the phase ordering laid out in the previous section, we now discuss the type of information that must be provided in a machine-description database for use by the scheduling and register allocation phases of EPIC code generation.

5.1 Operation, opcode, and register descriptors

As explained in Section 4, the EPIC code generator is structured around the process of binding a semantic operation to an architectural operation, each consisting of an opcode and some number of registers operands. The opcode and the registers may each be specified at various levels of binding, from the minimally bound generic opcode or register sets, through the more tightly bound access-equivalent opcode and register sets, to the completely bound compiler-opcodes or compiler-registers.

Corresponding to each opcode set at each level in the OBL, the mdes contains an *opcode descriptor* that specifies the compiler-opcodes contained in that set. It also specifies an opcode attribute descriptor (see Section 5.2) which records various semantic attributes of the opcode. In the case of a compiler-opcode, the opcode descriptor also specifies a bundle-macro descriptor (see Section 5.2) which, in turn, specifies the implementation of the compiler-opcode using architectural opcodes.

Likewise, for each register set at each level in the OBL, the mdes contains a *register descriptor* that specifies the compiler-registers contained in that set as well as a storage attribute descriptor (see Section 5.3) that specifies their storage attributes. For a compiler-register, the register descriptor also specifies a register-package descriptor (see Section 5.3) which, in turn, specifies the set of architectural registers that comprise a compiler-register.

As shown in Section 4.4, operations are specified at various levels of binding. In order of increasing degrees of binding, we have generic operation sets, access-equivalent operation sets, register-qualified operation sets or opcode-qualified operation sets, and the fully-qualified operations. Only certain combinations of opcode sets and RS-tuples yield legal operation sets. Each legal combination is defined by an *operation descriptor* which specifies

an opcode descriptor and as many register descriptors as there are register sets in the RS-tuple. Additionally, for opcode-qualified operation sets and fully-qualified operations, the operation descriptor also specifies a latency descriptor and a reservation table (see Section 5.4).

This constitutes the key information that we wish to emphasize in this report. Although there may be other information that needs to be attached to an operation, opcode or register set, it is not relevant to the topics that we wish to discuss here. We now describe the various items of information that are associated with opcode, register or operation descriptors.

5.2 Information associated with opcodes and opcode sets

Bundle-macro descriptors. A compiler-opcode is implemented by one or more architectural opcodes as specified by a bundle-macro descriptor. A *bundle-macro* represents a small computation graph which contains no control flow. The operators in this graph have a fixed issue time relative to one another. This computation graph may contain internal variables that may only be accessed by the operators of the bundle macro. Thus, their lifetime is entirely contained within the duration of execution of the bundle-macro and has a pre-determined length. Internal variables either represent storage resources that are not subject to register allocation or represent architecturally-invisible, transitory storage resources. Some operators may have one or more source operands that are bound to a literal. Source operands that are bound to neither an internal variable nor a literal constitute the source operands of the compiler-opcode of which this bundle-macro is the implementation. Those destination operands that are not bound to an internal variable constitute the destination operands of the compiler-opcode. A bundle-macro specifies the following information:

- A list of architectural opcodes.
- The issue time of each opcode, relative to the issue time of the earliest one.
- A list of internal variables.
- The binding of each operator's source operands to either an internal variable, a literal or a formal input argument of the bundle-macro.
- The binding of each operator's destination operands to an internal variable, a write-only register or a formal output argument of the bundle-macro.

In the simplest and most frequent case, a bundle-macro consists of a single architectural opcode, none of whose operands have been bound to either a literal or an internal variable. In some other cases, it consists of a single architectural opcode, one of whose source operands has been bound to a literal.

Opcode attribute descriptors. Every generic opcode set, access-equivalent opcode set, compiler-opcode and architectural opcode has an associated opcode attribute descriptor which specifies various properties of the opcode or the opcode set which are used during compilation. These include the following:

- The number of source and destination operands.
- If the target machine supports speculative execution, then whether or not the opcode (set) has a speculative version.
- If the target machine supports predicated execution, then whether or not the opcode (set) has a guarding predicate input.
- Opcode classification into certain abstract classes (such as integer, floating-point, memory, branch) for compiler use.
- Certain semantic properties useful during compilation; examples include whether or not the opcode represents an associative operation or a commutative operation.

5.3 Information associated with registers and register sets

Register-package descriptors. A compiler-register consists of a *register package,* i.e., one or more architectural registers, as specified by the register-package descriptor. The register-package descriptor specifies the following information:

- A list of architectural registers.
- The mapping from the bits of each architectural register to the bits of the compiler-register.

Most often, a compiler-register consists of a single architectural register. This includes the case of a compiler-register such as the stack pointer which might be implemented using (for instance) register 31 in the integer register file. A common, non-trivial use of a register-package is in the case of a double-precision compiler-register that is implemented as an even-odd pair of single-precision architectural registers.

Storage attribute descriptors. Every generic register set, access-equivalent register set, compiler-register and architectural register has an associated storage attribute descriptor which specifies five attributes which were discussed earlier and are listed here for completeness:

- The bit width of the register.
- The presence or absence of a speculative tag bit.
- Whether this is part of a static or rotating register file.
- Whether the register is read/write (the normal case), read-only (a literal) or write-only (the "bit bucket").
- For a read-only register, the value of the literal.

By definition, every compiler-register in a generic or access-equivalent register set possesses the same set of storage attributes.

5.4 Information associated with operations and operation sets

Latency descriptors. Each opcode-qualified operation set or fully-qualified operation has, associated with it, a set of latencies that are collectively provided in a *latency descriptor*. All latencies are expressed as the length of the time interval, measured in units of cycles, from the time of initiation of the operation, and are integer valued.

The following two latency values are provided for each register source operand:

- The *earliest read latency* (T_{er}) which is the earliest time at which a particular register source operand could possibly be read, relative to the initiation time of the operation.
- The *latest read latency* (T_{lr}) which is the latest time at which a particular register source operand could possibly be read, relative to the initiation time of the operation.

The following two latency values are provided for each register destination operand:

- The *earliest write latency* (T_{ew}) which is the earliest time at which a particular register destination operand could possibly be written, relative to the initiation time of the operation.
- The *latest write latency* (T_{lw}) which is the latest time at which a particular register destination operand could possibly be written, relative to the initiation time of the operation.

The following latency value is provided for each operation:

- The *operation latency* (T_{op}) which is the time, relative to the initiation time of the operation, at which it completes, i.e., the earliest point in time, ignoring interruptions, after which it will not cause any change of state or make use of any machine resources that are visible to the compiler.

The following two latency values are provided for each load or store operation:

- The *earliest memory serialization latency* (T_{em}) which is the earliest possible time, relative to the initiation time of the memory operation, at which it could reach the point in the memory pipeline beyond which all operations are guaranteed to be processed in FIFO order.
- The *latest memory serialization latency* (T_{lm}) which is the latest possible time, relative to the initiation time of the memory operation, at which it could reach the point in the memory pipeline beyond which all operations are guaranteed to be processed in FIFO order.

The following latency value is provided globally for all branch operations and is, therefore, not part of any latency descriptor:

- The *branch latency* (T_{br}) which is the time, relative to the initiation time of a branch operation, at which the target of the branch is initiated.

These latencies are discussed at length below.

Since the mdes is the compiler's view of the processor, the latencies for an operation are specified in the abstract context of the scheduler's virtual time. In the scheduler's virtual time, one instruction (possibly consisting of multiple operations) is started per cycle. An abstract clock separates pairs of instructions that are consecutive in time. In this logical world view, one instruction is initiated at the instant of each abstract clock and results are written into destination registers at the instant of an abstract clock, i.e., processor state changes at these instants. Source registers are viewed as being read "just after" the abstract clock specified by the read latency (relative to the initiation time of the operation). This is only a logical view which must be extracted from an examination of the timing relationships of the processor.

Thus if an operation that writes a register and another one that reads that same register are scheduled such that the scheduled time of the read event is the same as (or later than) the

scheduled time of the write event, then the result computed by the first one will serve as the operand for the second. If this is not desired, the operations should be scheduled such that the read event is scheduled to occur at least once cycle before the write event. Generally, multiple write events that attempt to change the same item of processor state at the same time, result in an undefined state. Accordingly, such write events must be separated in time by some minimum amount which is usually, but not necessarily, one cycle. In this report we concern ourselves only with processors for which this minimum requisite interval is one cycle. Likewise, we restrict ourselves to processors in which a register read can be initiated every cycle.

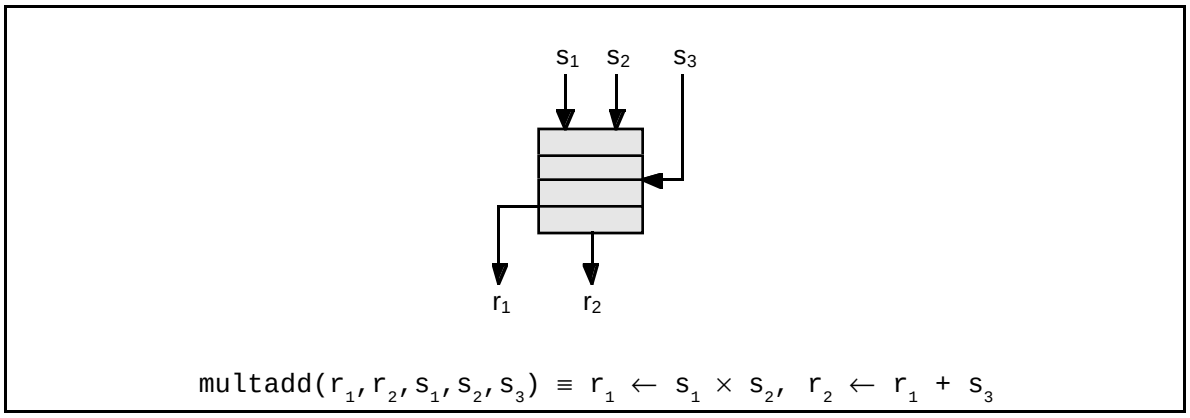

Figure 11: A temporal representation of a pipelined 3-input, 2-output floating-point multiply-add operation.

Figure 11 shows the example of a pipelined 3-input, 2-output floating-point multiply-add operation. The first two source operands are read sometime during the first cycle after the initiation of the operation. So, their read latency is 0. The third source operand, the one required by the add, is read sometime between two and three cycles after the operation is initiated and, therefore, its read latency is 2. The result of the multiply is written at the end of 3 cycles, while that of the add is written at the end of 4 cycles.

Interruptions--or more specifically interruption handlers that re-execute an operation--complicate the picture. Examples are the page fault handler that re-executes the load or store after processing the page fault, or a floating-point exception handler that attempts to repair the exception. In such cases, the source operand registers must be retained unmodified, until the latest time at which such an exception can be flagged, so that they can be read again, effectively with a latency equivalent to the exception reporting latency. No other operation should be scheduled in such a way that it can end up modifying these source

registers prior to this point in time. After that, the source operand registers can be deallocated and reused (at least as far as this operation is concerned). Likewise, an exception handler may also cause a result to be written either early or late relative to the nominal write latency. It is also possible that there is some non-determinacy even in the nominal latencies. For instance, a heavily pipelined floating-point add may normally take two cycles to write its result, but might take an extra cycle to do so if the result needs to be normalized.

This leads to a situation in which, for every input or output operand, one must specify an earliest and a latest read or write latency, respectively. In all cases, when specifying the above latencies, it is the responsibility of the creator of the machine description to account for all the diverse factors that can affect the latencies with which the inputs may be read and the outputs written.

Table 6: Part of the latency descriptor for the 5-operand multiply-add operation in Figure 11.

	Source read latency		**Destination write latency**	
	Earliest (T_{er})	**Latest (T_{lr})**	**Earliest (T_{ew})**	**Latest (T_{lw})**
s_1	0	2	-	-
s_2	0	2	-	-
s_3	2	2	-	-
r_1	-	-	2	3
r_2	-	-	2	4

Once again, consider the multiply-add example of Figure 11. Let us assume that either the multiply or the add may cause an exception at a latency of 2 cycles. Consequently, the source operands may be read by the exception recovery code with an effective latency of 2 cycles. Thus, the latest read latencies for all three source operands is 2 cycles. Assuming that the exception handler computes and writes both results to their respective destinations before returning control, the effective latency with which the destination operands may be written by the exception recovery code is 2 cycles. Consequently, the earliest write latencies for both destination operands must be set to 2 cycles. Table 6 shows part of the latency descriptor for this operation.

The latencies involving the register operands of loads, stores and branches are defined as above. Because of cache faults, memory bank conflicts, etc., the behavior of loads and stores is unpredictable. Therefore, the expected latency with which the result will be written to the destination register, is used as the write latency of a load operation. To reduce the variance between the expected and the actual load latencies, HPL-PD provides different load opcodes that correspond to the level in the cache hierarchy at which the requested datum is expected to be found. The latency of a read from a particular level is used as the latency for operations associated with the load opcode corresponding to that level.

Also, because of the aforementioned unpredictability of memory operations, no attempt is made to model precisely the latency with which the memory location is accessed by a load or store. Instead, the assumption is made that as long as memory operations to the same memory location get to a particular point in the memory system's pipeline in a particular order, they will be processed in that same order. So, in terms of preserving the desired order of memory accesses, the latency that we are concerned with is that of getting to the serialization point in the memory pipeline. We define the latency to the serialization point to be the *memory serialization latency*. In the presence of page faults, one must allow for the possibility that the operation will be re-executed after the page fault has been handled, which leads to some ambiguity in the effective memory serialization latency. To address this, two latencies are associated with each memory operation: the earliest memory serialization latency, T_{em}, and the latest memory serialization latency, T_{lm}.

For fully-qualified operations, the above latency values are derived by directly examining their mapping to architectural opcodes. When a compiler-opcode is implemented by a non-trivial bundle-macro, the computation of these latencies is a bit more intricate, but still unambiguous. (A point to bear in mind here is that the relative schedule times of all the architectural opcodes in the bundle-macro are known.)

For generic and access-equivalent operation sets, the latency values are derived by determining, for each latency, either the smallest or the largest corresponding latency over all fully-qualified operations for that operation set. All four latencies are calculated so that they are optimistic in the sense that they minimize inter-operation delays and, hence, the schedule length. Consequently, T_{lw} and T_{lr} are computed as the *min* of the corresponding latencies over all fully-qualified operations, whereas T_{er} and T_{ew} are computed as the *max* of the corresponding latencies. Note that the set of latencies computed in this manner may not be achievable, i.e., there may be no single fully-qualified operation that possesses this set of latencies.

Resources and reservation tables. Each target machine is characterized in the mdes by a set of *machine resources*. A machine resource is any aspect of the target architecture for which over-subscription is possible if not explicitly managed by the compiler. Such over-subscription may either lead to hardware conflicts and incorrect, undefined results, or at the very least result in inefficient execution due to numerous pipeline stalls caused by the arbitration logic. Therefore, it is necessary to model the resource requirements of each opcode-qualified operation set or fully-qualified operation and present it to the compiler in a compact form. The scheduler uses this information to pick conflict-free operation alternatives for each operation of the computation graph from its pre-assigned access-equivalent operation set, making sure that no resource is used simultaneously by more than one operation. The three kinds of resources, which are modeled in the mdes, are described below.

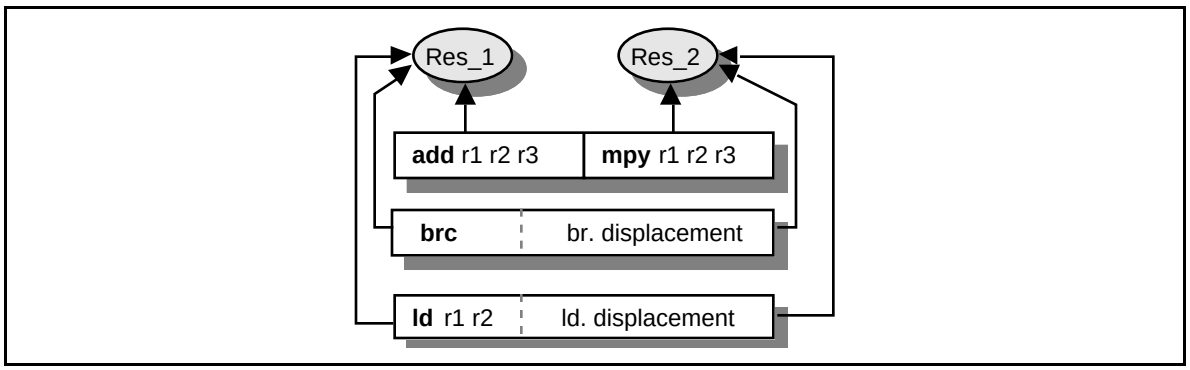

Figure 12: Instruction format conflicts may be modeled using abstract resources. The instruction format may dictate that either a branch with displacement may be issued, or a load with displacement may be issued or an add and a multiply may be issued together. This can be modeled by two abstract resources **Res_1** and **Res_2** as shown. The branch and the load operations each use both resources while the add only uses **Res_1** and the multiply only uses **Res_2**.

Hardware resources are hardware entities that would be occupied or used during the execution of architectural opcodes identified by an opcode-qualified operation set or a fully-qualified operation. This includes integer and floating point ALUs, pipeline stages, register file ports, input and result buses, etc. Note that only those aspects of the target architecture that have the possibility of sharing and resource conflict need to be modeled as independent resources. For instance, in the case of a non-blocking, pipelined ALU, all the pipeline stages need not be modeled separately, modeling the first stage subsumes the possible

resource conflicts at other stages. Similarly, only those register file ports and buses need to be modeled that are shared by two or more functional units.

Abstract resources are conceptual entities that are used to model operation conflicts or sharing constraints that do not directly correspond to any hardware resource. This includes combinations of operations that are not allowed to be issued together due to instruction format conflicts such as sharing of an instruction field. In such cases, we create an abstract resource that is used at the same time (e.g. issue time) by the various operations that have a conflict as shown in Figure 12. This prevents the scheduler from scheduling such operations simultaneously.

Sometimes several identical resources, that may be used interchangeably by an operation, are present in the architecture. These could be either hardware resources, e.g. any two input buses or any result bus that connects the given functional unit to the appropriate register file ports, or abstract resources, e.g. any issue slot of an instruction for issuing the given operation. It is useful to represent such resources as *counted resources*, i.e. the scheduler is merely required to obey the constraint that the total number of such resources used in a given cycle does not exceed the number available, but the specific resource assigned to an operation is immaterial or may even be determined dynamically.

In addition to defining the set of all machine resources for the purpose of the compiler, the mdes also records how each opcode-qualified operation set or fully-qualified operation uses these resources during specific cycles relative to its initiation time. Collectively, such a table of resource usages is termed a *reservation table* [42]. As an example, we show the pictorial representation of the reservation tables for the Add, Multiply, and Load operations for a hypothetical machine in Figures 13a, 13b and 13c respectively. The tables use hardware resources (ALU, MULT, ResultBus) to model resource conflicts in the datapath and abstract resources (Res_1, Res_2) to model instruction format constraints. Had they been present, counted resources would be marked with the number of resources needed by that operation at that cycle rather than a simple checkmark.

Reservation tables can be an economical way of representing the pairwise constraints between alternatives. If N alternatives all make one use each of a resource in their respective reservation tables, N(N+1)/2 pairwise constraints have been specified, implicitly. For instance, by virtue of the fact that all three reservation tables in Figure 13 make use of the ResultBus resource, we have implicitly expressed the three inter-operation constraints that an Add and a Load may not be issued simultaneously, and that neither one

may be issued one cycle after a Multiply as well as the three constraints that only one each of the three operations can be issued in any given cycle. If desired, these constraints can be made explicit by computing the *forbidden initiation intervals* between each pair of reservation tables [42].

In a machine with predicated or conditional execution [13, 15], an operation may use certain resources only conditionally (if the operation's predicate operand, which serves to guard the operation, is `true`). Such resource usages must be marked in the reservation table as being conditional so that the scheduler can make the same resource available to another operation under a mutually exclusive predicate.

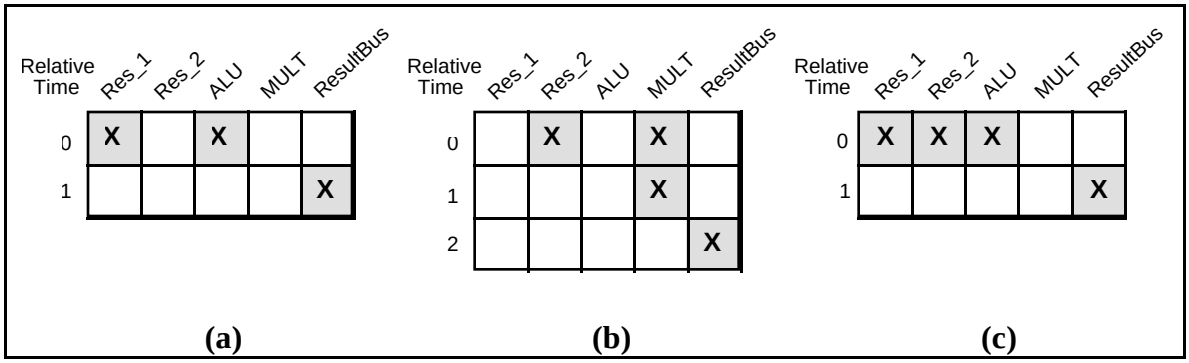

Figure 13: Reservation tables. (a) For an Add operation, which uses Res_1 and ALU at cycle 0 and ResultBus at cycle 1. (b) For a Multiply (Mpy) operation, which is only partially pipelined as it uses MULT resource for two consecutive cycles. (c) For a load operation, which uses the ALU to do address calculation.

When the compiler-opcode corresponding to the opcode-qualified operation set or fully-qualified operation is implemented with a non-trivial bundle-macro, the reservation table for the operation is computed by taking the union of the resource usages of the component architectural operations. In principle, reservation tables, too, could be conservatively approximated for access-equivalent and generic operation sets, although the authors are not yet aware of any useful way of doing so.

6 Usage of the mdes by the EPIC code generator modules

We now look at how the various modules of an EPIC code generator, with a special emphasis on scheduling and register allocation, make use of the above information in the process of translating semantic operations to architectural operations. Our focus here is not

on the specific algorithms and heuristics used but, rather, on what each phase is supposed to accomplish, with a view to identifying what information each one needs from the mdes.

6.1 Code selection

The code selector constitutes the first phase of an EPIC code generator. This is usually the most important phase for DSP compilers where decisions are simultaneously taken regarding collapsing multiple RISC operations into CISC operations, selecting operand addressing modes, assigning source and destination registers, and establishing connectivity between the functional units and the operand registers. A good deal of research has been performed on the topic of creating retargetable DSP compilers [22, 24, 21, 4, 23, 5].

Engineering a retargetable code selector for an EPIC processor is much closer in spirit to that for a conventional processor [16, 17, 43, 44, 18, 19]. The primary task of the code selector is to bind the abstract semantics of the virtual machine that is targeted by the program's intermediate form to the machine-level semantics of the target machine. To the extent that the architectural opcode repertoire contains CISC-like opcodes, the code selector must rewrite the incoming computation graph, consisting of RISC-like semantic operators, variables and literals, into the outgoing computation graph, consisting of the generic opcode sets and generic register sets of the target EPIC. The mapping from virtual registers to generic register sets depends upon the data type of each variable (e.g. 32-bit integers vs. 64-bit floats) which determines the width of the compiler-register needed, whether the data needs to be in the rotating register file or static register file, whether a parameter-passing register or a caller-save or callee-save register is to be used, etc. These are matched with the storage attributes that are associated with each generic register set.

Typically, a retargetable code selector uses a database consisting of pairs of patterns, each of which is a computation graph. In each pair, the first pattern consists of semantic operators and semantic variables, whereas the second one consists of generic opcode sets and generic register sets. The code selector first covers the program's computation graph with the semantic patterns. Using cost models and heuristics, a set of semantic patterns is chosen which best tiles the program's computation graph. Then, each semantic pattern is replaced by the corresponding pattern that consists of generic opcode and register sets.

6.2 Pre-pass operation binding

The pre-pass operation binding phase refines the generic operation sets to access-equivalent operation sets. The goal of the pre-pass operation binding phase is to re-structure the

program's computation graph in such a way as to jointly meet three objectives. Firstly, the computation graph must be distributed over the resources of the processor in such a way as to make full use of them, thereby reducing the schedule length. In general, this requires the insertion of copy operators. Secondly, the copy operators should be inserted into the graph in such a way that the critical path length does not increase, thereby causing the schedule length to increase. Thirdly, this phase must attempt to assign the largest possible access-equivalent opcode sets and register sets to the vertices of the computation graph. When selecting an access-equivalent operation set, there may be a tradeoff between selecting a large access-equivalent opcode set, which benefits the scheduler, and selecting a large access-equivalent register sets, which benefits the register allocator.

The pre-pass operation binding phase employs various heuristics to achieve its ends. When the target machine is organized into multiple clusters of similar functional units and register files, this phase is very similar in spirit to the code and data partitioning phases of parallelizing MIMD compilers that attempt to distribute work and data among processors while inserting the minimum amount of inter-processor communication. However, in the EPIC context, there is a distinct bias towards achieving good utilization of the available execution units and reducing the overall schedule length, and not in merely reducing the number of inter-cluster move operations. For example, this phase in the Multiflow compiler performed a trial schedule attempting to minimize the data transfer latency [29]. It performed a bottom-up greedy (BUG) assignment of generic operation sets to functional units and register files attempting to achieve the minimal critical path for the overall program. The resulting schedule was discarded and only the access-equivalent opcode and register set bindings were retained.

The details of the information that the machine-description database must include depend to a large extent upon the specific heuristics employed by the pre-pass operation binding algorithm. However, the need for certain information is clear. The database must contain the available choices of access-equivalent opcode, register and operation sets for each generic opcode, register or operation set, respectively. Since the heuristics employed by this phase have a strong flavor of scheduling, it is reasonable to assume that all of the information required by the scheduler is needed during this phase as well. Lastly, in order to be able to insert copy operators properly, the database must make available the set of valid copy operations for moving data from one access-equivalent register set to another.

6.3 Scheduling

The scheduler is the module which uses the most detailed information about the target architecture, its machine resources, its opcode-qualified operation sets, their reservation tables and latencies. The primary task of the scheduler is to select, for each operator, a compiler-opcode from that operator's access-equivalent opcode set and to assign a schedule time to it, subject to the constraints of data dependence and resource availability, while minimizing the schedule length by making full use of the machine resources.

Dependence graph construction. The actual scheduling step is preceded by a preparation step that builds the *data dependence graph* of the region to be scheduled. The vertices of this graph are operations which have been annotated with access-equivalent operation sets, connected by dependence edges representing constraints on scheduling. There are three kinds of dependence edges that result from a pair of operations accessing a register in common: *flow dependence*, *anti-dependence*, and *output dependence* edges. Flow analysis of the region to be scheduled determines the kind of edges that need to be inserted between operations that either define or use the same or overlapping (aliased) registers, either virtual or physical.

Often, the scheduler requires the insertion of a different kind of edge between two operations to specify an ordering between the two that is unrelated to the register operands that are read or written in common by those two operations. These are called sync edges or sync arcs. Sync arcs between a pair of memory operations (loads or stores) specify the ordering that must be maintained between those operations in order to honor the flow, anti- and output dependences that exist between them by virtue of their (possibly) accessing the same memory location. Such sync arcs are termed *Mem edges*. In comparison to the dependences that exist between operations because of their use of a common register, where the presence or absence of the dependence is unambiguous, memory dependences can be ambiguous since the addresses of the referenced memory locations are not always known at compile-time. So, instead of using the regular flow, anti- or output dependences, Mem edges are used.

Sync arcs also exist between a branch and the operations before and after it. These are termed *Control1 edges*. Control1 edges from a branch to the non-speculative operations after it (including other branches) represent control dependences which ensure that those operations will not be issued prior to the branch taking. This is because they depend upon the branch going a particular way and should not, therefore, be issued if the branch goes

the other way. The Control1 edges to a branch from the operations before it ensure that those operations are issued, perhaps even completed, before the branch takes.

Thus far, we have ignored the possibility that the code may be predicated. If two operations which access the same register are predicated, the possibility arises that their respective predicates are mutually exclusive, i.e., they cannot both be true simultaneously. In such cases, no dependence edge need be drawn between the two operations. Since they cannot both execute during any given traversal of that region, there cannot possibly be any interference caused as a result of both accessing the same register. However, it is still correct, albeit conservative, if edges are inserted as if the code was unpredicated.

Edge delay computation. Each dependence edge is decorated with an *edge delay* that specifies the minimum number of cycles necessary, between the initiation of the predecessor operation and the initiation of the successor operation, in order to satisfy the dependence. The manner in which the edge delays are computed depends on whether the operations function under the freeze or drain interruption model.

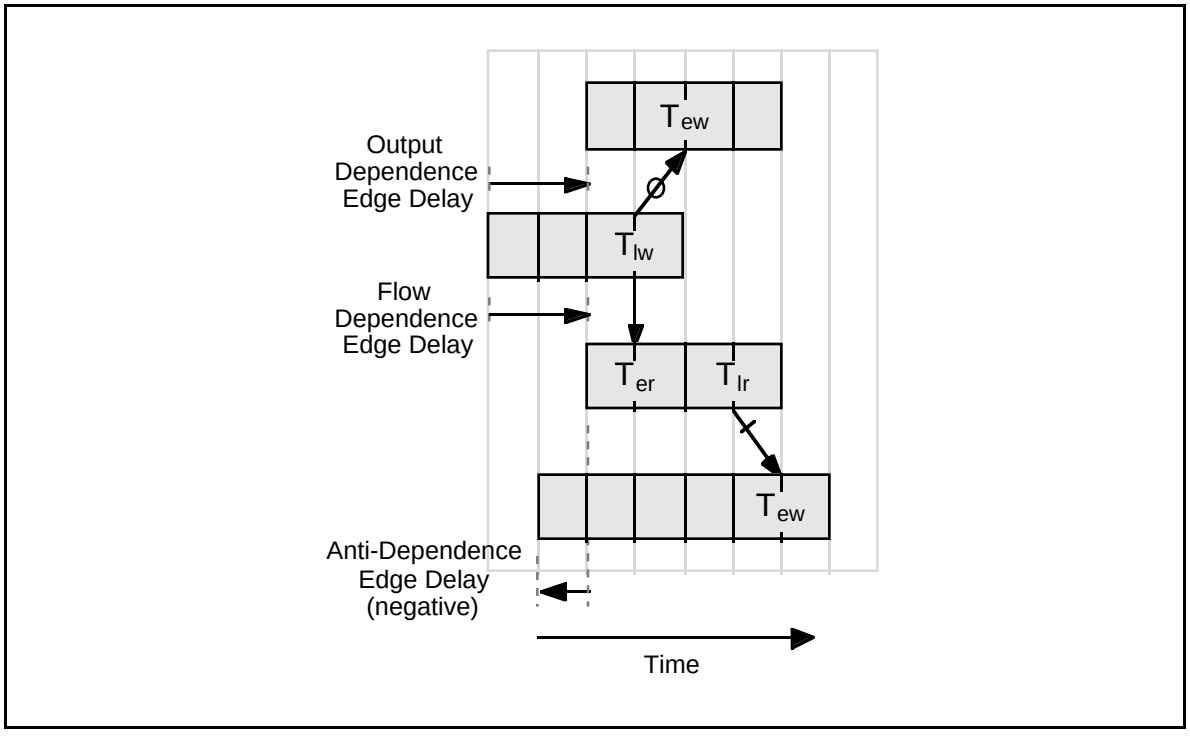

Figure 14: Edge delays for the three types of data dependences.

Consider first the formulae for computing edge delays when the successor operation uses the freeze model, regardless of which model the predecessor uses (Table 7). These

formulae are best understood by referring to Figure 14 and the dependence constraints below, where t(X) is the time at which operation X is initiated, P is the predecessor operation and S is the successor operation.

$$\text{Flow dependence:} \quad t(S) + T_{er}(S) \geq t(P) + T_{lw}(P)$$

$$\text{Anti-dependence:} \quad t(S) + T_{ew}(S) \geq t(P) + T_{lr}(P) + 1$$

$$\text{Output dependence:} \quad t(S) + T_{ew}(S) \geq t(P) + T_{lw}(P) + 1$$

For each type of dependence edge, the delay is the minimum value of t(S) - t(P) that is permitted by the corresponding constraint.

In light of the definition of these latencies, the operation producing a result writes the datum to its destination register no later than T_{lw} cycles from the initiation of that operation, whereas the operation consuming that datum reads its source register no earlier than T_{er} cycles from the initiation of that operation. Consequently, in the case of a flow dependence, the earliest time at which the successor operation reads its input may be scheduled to be as early as, but no earlier than, the latest time at which the predecessor operation writes its result. The flow dependence edge delay is, therefore, the difference between the latest write latency and the earliest read latency of the predecessor and successor operations, respectively. Likewise, as explained in Section 5.4, an anti-dependence (output dependence) means that the earliest time at which the successor operation writes its result may be scheduled to be no earlier than one cycle after the latest time at which the predecessor operation reads its input (writes its result).

Table 7: Computation of edge delays for the various types of dependences. In this table, P represents the predecessor operation and S represents the successor operation. The scheduler must satisfy the constraint that t(S) - t(P) is greater than or equal to the delay as computed according to this table, where t(X) represents the time at which operation X is scheduled to start execution.

Type of dependence	**Edge Delays when S operates with the Freeze Model**	**Edge Delays when S operates with the Drain Model**
Flow dependence	$T_{lw}(P) - T_{er}(S)$	$\max[0, T_{lw}(P) - T_{er}(S)]$
Anti-dependence	$T_{lr}(P) - T_{ew}(S) + 1$	$\max[0, T_{lr}(P) - T_{ew}(S) + 1]$
Output dependence	$T_{lw}(P) - T_{ew}(S) + 1$	$\max[0, T_{lw}(P) - T_{ew}(S) + 1]$

From Table 7 and Figure 14, we see that edge delays may turn out to be positive or negative depending on the latencies of the operands. A negative edge delay provides greater

scheduling freedom for the successor operation, i.e. the successor operation may even be initiated earlier than the predecessor operation and still meet the data dependence constraint.

Edge delays are computed slightly differently for machines that drain their pipelines on fielding an interruption. Draining the pipelines causes the operations that are currently in flight to read their inputs and write their results, in effect, at the time that the interruption takes place and, possibly, *before* their scheduled latency in terms of the schedule's virtual time. Predecessor operations that were scheduled to be initiated after the point of interruption will not be issued until after the interruption has been serviced and well after their successor operations have been drained from their pipelines. Consequently, they can end up performing their reads and writes *after* the reads and writes that they were supposed to precede, thereby violating the semantics of the program.

This problem is avoided by the enforcement of a simple rule--that a successor operation never be scheduled to be initiated earlier than a predecessor operation, if that successor operation operates under the drain model. If the successor operation operates with the freeze model, the rule permits the predecessor to be scheduled later than the successor. If this rule is enforced, it is impossible for an interruption to cause the successor operation to be issued and perform all of its processor state changes without the predecessor operation having been issued, either concurrently or earlier. Consequently, the interruption cannot result in any dependence constraint being violated. With this in mind, the edge delays are adjusted with the following additional constraint:

$$t(S) \geq t(P).$$

For each type of dependence edge, the only difference from the freeze case is that the edge delay is permitted to be no less than 0 (as specified in the rightmost column of Table 7).

The dependence edges due to the register accesses of loads, stores and branches have their edge delays computed as described above, and the latencies needed to compute them are provided in their latency descriptors just as for other operations. However, sync arcs are treated differently. The assumption is made that as long as memory operations to the same memory location get to a particular point in the memory system's pipeline in a particular order, they will be processed in that same order. So, with a view to preserving the desired order of memory accesses, but ignoring address-related exceptions, the delay on the Mem edge between two memory operations that may be to the same memory location should be set to

$$T_m(P) - T_m(S) + 1,$$

where T_m. is the memory serialization latency, and where P and S are the predecessor and successor operations, respectively.

In the presence of page faults, one must allow for the possibility that the predecessor operation will be re-executed after the page fault has been handled, which leads to some ambiguity in the memory serialization latency. To address this, the edge delay should be set to

$$T_{lm}(P) - T_{em}(S) + 1.$$

The above formula applies to the freeze case. In the drain case, it is further constrained to be non-negative.

If the hardware supports prioritized memory operations [15] then these edge delays can be set to 0. This is advisable only if it is statistically likely that the operations are to different memory locations, even though this cannot be proved at compile-time.

Next, consider Control1 edges from a branch to those non-speculative operations that should not be issued until after the branch has completed. Included are all operations that depend upon the branch going a particular way and which should not, therefore, be issued if the branch goes the other way. Also, a subsequent branch operation whose branch condition is not mutually exclusive with that of the preceding branch must be constrained to not issue until the preceding branch has completed[11]. In both cases, the edge delay is set to the branch latency, T_{br}, of the predecessor branch. This edge delay does not depend upon the interruption model used by the successor operations.

Finally, consider the Control1 edges to a branch from the operations that precede it in the sequential ordering. For correct execution, one must guarantee that all of these operations are issued before the branch operation completes. (If not, if the branch takes, then some of these operations which were supposed to have been executed will not even have been issued, and they never will.) This leads to a possibly negative delay between each such operation and the branch as given by $(1 - T_{br})$. Note that since branch operations always adhere to the freeze model, we do not need a second formula for the drain case.

[11] If the branch conditions are known to be mutually exclusive, the edge between the two branches can be deleted [45].

More conservatively, one can require that all preceding operations have completed by the time the branch completes. If so, the edge delay is given by (T_{op} - T_{br}), where T_{op} is the operation latency of the predecessor operation.

Tracking resource usage. Given a data dependence graph, properly decorated with edge delays, the actual task of scheduling can begin. The reservation tables of the various opcode-qualified operation sets and fully-qualified operations are used during scheduling to prevent scheduling of alternatives in a conflicting manner.

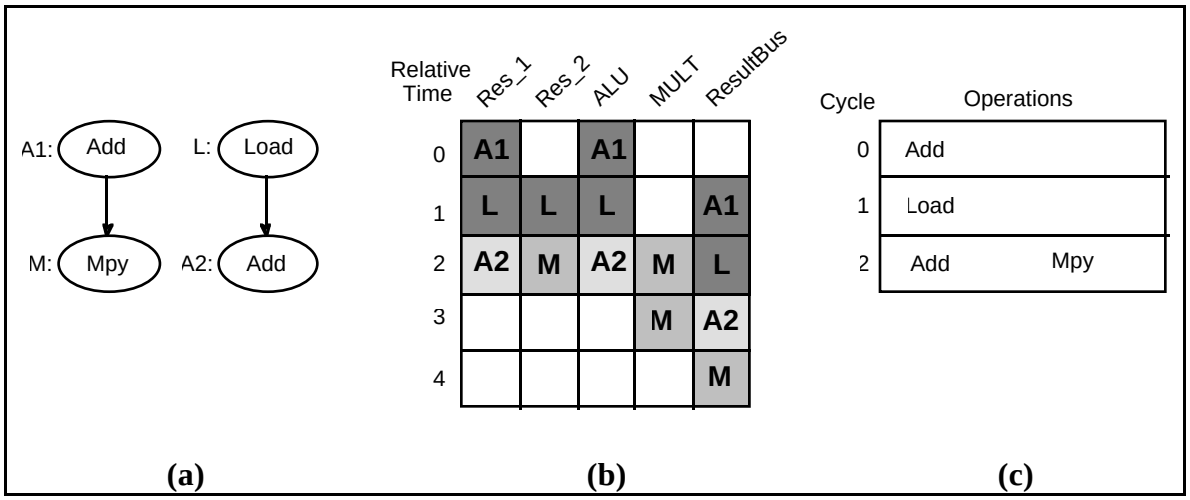

Figure 15: Resource tracking using the resource usage map. (a) A simple computation graph to be scheduled. (b) The resource usage map which keeps track of the partial schedules during scheduling. The figure shows the map after all four operations has been scheduled using the reservation tables of Figure 13. The operation A1 is scheduled to start at time 0, M at time 2, A2 at time 2 and L at time 1. (c) The final schedule for the computation graph shown in (a).

Consider the example computation graph shown in Figure 15a. Each operator in the graph is scheduled in some priority order (based on data dependence) by choosing a compiler-opcode from its assigned access-equivalent opcode set. The choice is made such that the reservation table of the selected alternative (Figure 13), offset by the proposed initiation time of the operation, does not conflict with any of the previously reserved resources. For this purpose, the scheduler internally maintains a *resource usage map* which keeps track of when each machine resource will be used by the previously scheduled operations (Figure 15b). In the example shown, suppose the operations are scheduled in the order A1, L, M, A2. The operation L cannot be scheduled at cycle 0 since its reservation table will conflict with the previously scheduled operation A1 at resource positions Res_1 (cycle

0), ALU (cycle 0), and ResultBus (cycle 1). On the other hand, after scheduling A1, L and M at times 0, 1 and 2, respectively, the operation A2 can fit nicely into the same cycle as M since it does not cause a resource conflict. Figure 15c shows the final schedule.

6.4 Register allocation and spill code insertion

In our phase ordering, scheduling is performed before register allocation, and the operators have been bound to compiler-opcodes. The job of the register allocator is to take virtual registers, that have been partially bound to access-equivalent register sets, and bind them to compiler-registers. In the process, it binds each opcode-qualified operation set to a fully-qualified operation. We shall restrict our discussion to the case in which either the code is not predicated or the interference graph construction step of register allocation is performed in a conservative fashion, essentially acting as if the code was not predicated. For a discussion of register allocation of predicated code, the reader is directed to the literature [46, 47].

Interference graph construction. Two virtual registers may be assigned the same compiler-register if doing so does not alter the flow dependences of any of the operations involved and if for the new anti- or output dependences that are introduced, no edge delays (as computed above) are violated by the pre-existing schedule. Register allocation is performed with the help of an interference graph which consists of a vertex per virtual register. If two virtual registers may not be assigned the same compiler-register, then an interference edge is inserted between the corresponding vertices in the interference graph. Register allocation is then formulated as a graph coloring problem on this interference graph.

We consider first the case of the freeze model. Let A and B be two virtual registers. Let W_A and W_B represent the set of operations that write to A and B, respectively. Likewise, let R_A and R_B represent the set of operations that read from A and B, respectively. If A and B are to be assigned the same compiler-register without altering any of the flow dependences, all of the reads and writes to one of the virtual registers (the first one) must precede all of the writes to the other virtual register (the second one) by at least one cycle[12]. If not, at least

[12] Note that in this case no flow dependence can be introduced by assigning the two virtual registers to the same compiler-register as long as the second virtual register is written prior to it being first read. If this assumption is false, then the offending operation is reading an uninitialized virtual register, and we take the position that it does not matter which value this read yields. In other words, although the flow dependence for this offending operation has changed, we take the position that the program semantics have not been altered in any material way.

one of the readers of either A or B will get the wrong value. Without loss of generality, assume that A is the first lifetime. This constraint can then be stated as

$$\min[t(W_{Bi})+T_{ew}(W_{Bi})] \geq \max[t(R_{Aq})+T_{lr}(R_{Aq}),\ t(W_{Ap})+T_{lw}(W_{Ap})] + 1$$

across all W_{Bi}, W_{Ap} and R_{Aq} that are members of W_B, W_A and R_A, respectively, and where t(X) is the scheduled initiation time of operation X. If the edge delay on every newly introduced anti- or output dependence edge delay is to be satisfied, so that no interference edge need be placed between A and B, the following two inequalities must be satisfied:

$$\text{Anti-dependence:} \quad t(W_{Bi}) + T_{ew}(W_{Bi}) \geq t(R_{Aq}) + T_{lr}(R_{Aq}) + 1$$

$$\text{Output dependence:} \quad t(W_{Bi}) + T_{ew}(W_{Bi}) \geq t(W_{Ap}) + T_{lw}(W_{Ap}) + 1$$

for any W_{Bi}, W_{Ap} and R_{Aq} that are members of W_B, W_A and R_A, respectively.

Table 8: Computation of register lifetimes and initiation lifetimes for a virtual register. t(X) is the time at which operation X is scheduled to start execution. i ranges over all operations that write to that virtual register and j ranges over all operations that read from that virtual register.

Register Birth Time	$\min[\ t(W_i)+T_{ew}(W_i)\ ]$,	over all i
Register Death Time	$\max[\ t(W_i)+T_{lw}(W_i),\ t(R_j)+T_{lr}(R_j)\ ] + 1$,	over all i, j
Initiation Birth Time	$\min[\ t(W_i)\ ]$,	over all i such that W_i drains
Initiation Death Time	$\max[\ t(W_i),\ t(R_j)\ ]$,	over all i, j

Since the two inequalities must be satisfied between every member of W_B and every member of either R_A or W_A, respectively, the two conditions that must both be true for there to be no interference edge between A and B are:

$$\text{Anti-dependences:} \quad \min[t(W_{Bi})+T_{ew}(W_{Bi})] \geq \max[t(R_{Aq})+T_{lr}(R_{Aq})] + 1$$

$$\text{Output dependences:} \quad \min[t(W_{Bi})+T_{ew}(W_{Bi})] \geq \max[t(W_{Ap})+T_{lw}(W_{Ap})] + 1$$

which can be combined into the single inequality:

$$\min[t(W_{Bi})+T_{ew}(W_{Bi})] \geq \max[t(R_{Aq})+T_{lr}(R_{Aq}),\ t(W_{Ap})+T_{lw}(W_{Ap})] + 1$$

over all i, p and q. Note that this is the same constraint as the one needed to ensure that no flow dependence is altered.

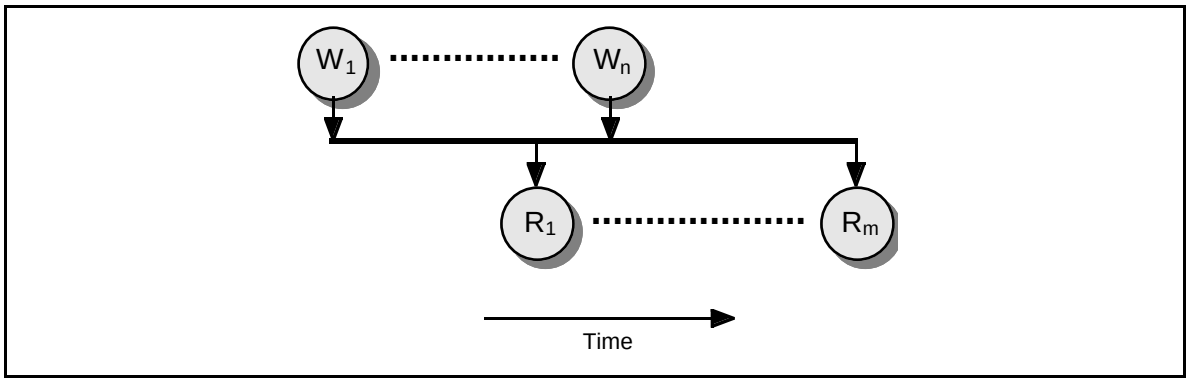

Figure 16: The lifetime of a register as determined by the times at which it might be written by producer operations W_1,..,W_n, and the times at which it might be read by consumer operations R_1,..,R_m.

Instead of computing the terms on the two sides of the inequality repeatedly for each pair of virtual registers, one may observe that the left hand side is a function only of B, and the right hand side is a function only of A. These two terms can be computed, once and for all, for each virtual register as shown in Table 8. The *register birth time* is the earliest time at which any of the operations that write to that virtual register, can end up doing so. The earliest time at which an operation writes a result is computed as the sum of the operation's scheduled initiation time and the earliest write latency, T_{ew}, for that destination operand. The *register death time* is one cycle more than the latest time at which any of the operations, that read from, or write to, that virtual register, can do so. The latest time at which an operation accesses a virtual register is computed as the sum of that operation's scheduled initiation time and either the latest read latency, T_{lr}, if it is an input operand or the latest write latency, T_{lw}, if it is a destination operand. In order to ascertain these latencies, the latency descriptors associated with the opcode-qualified operation sets, are consulted. The time interval between the birth and death times is the *register lifetime* of that virtual register (Figure 16).

Using the concept of register lifetimes, the interference criterion may be expressed as follows:

> two virtual registers have an interference edge between them in the interference graph if and only if their register lifetimes overlap.

This apparently convoluted way of deriving something very similar to the well-known condition for interference provides us the mechanism for dealing with the drain model as well. Let Y_A represent the set of operations that either read or write A. Let X_B represents the

set of operations that both write B and drain on the occurrence of an interruption. Referring to Table 7, we see that for the drain model an additional condition must be satisfied if an interference edge is not to be inserted between virtual registers A and B. Once again assuming that the earliest scheduled operation that accesses A is scheduled earlier than the earliest scheduled operation that writes to B, we have the following additional inequality that must be satisfied if an interference edge is to not be inserted:

$$t(X_{Bi}) \geq t(Y_{Ap})$$

for any X_{Bi} and Y_{Ap} that are members of X_B and Y_A, respectively. Since the inequality must be satisfied between every such member of X_B and every member of Y_A, the additional condition that must be true for there to be no interference edge between A and B is:

$$\min[t(X_{Bi})] \geq \max[t(Y_{Ap})]$$

over all i and p[13].

As before, we define the *initiation birth time* as the earliest initiation time of any of the operations that both write to that virtual register and drain on the occurrence of an interruption (Table 8). The *initiation death time* is the latest initiation time of any of the operations, that either read from or write to that virtual register. The time interval between these two times is the *initiation lifetime* of that virtual register.

Then, in the case that at least one of the operations that write the second (later) lifetime drain on the occurrence of an interruption, the interference criterion may be expressed as follows:

> two virtual registers have an interference edge between them in the interference graph if and only if either their register lifetimes overlap or their initiation lifetimes overlap.

The register allocation phase is preceded by a lifetime analysis step in which the register lifetime and, if necessary, the initiation lifetime, of each virtual register are computed, which enables the construction of the interference graph. After that, register allocation is performed using graph coloring. Each virtual register is constrained to be colored with (i.e., to be assigned) only the compiler-registers in its access-equivalent register set. Virtual registers with an interference edge between them must not be colored with either the same

---

[13] This constraint should be understood to be non-existent if the set X_B is empty.

compiler-register or with compiler-registers that have one or more bits in common. To ascertain this, the register-package descriptor is consulted. If the available compiler-registers are not sufficient to color all the variables, then some of the variables are spilled to memory by inserting the appropriate spill code (stores and loads). Again, any number of heuristics exist for determining an efficient allocation which attempts to minimize the amount of spill code.

6.5 Post-pass scheduling

The spill code inserted during register allocation must also be scheduled. For certain types of scheduling problems, when using certain scheduling algorithms, it is possible to allocate the registers and introduce the spill code during scheduling [29]. This is known as integrated scheduling and register allocation. In such cases, one can schedule the spill code as it is generated. In other cases, such integrated scheduling is very difficult [48, 30, 31] and the only option is to rerun the scheduler once again. Except for the recently introduced spill code, which consists of register-qualified operation sets, the rest of the operations are fully-qualified. As noted earlier, we favor the strategy of unbinding these fully-qualified operations to the level of register-qualified operation sets, by unbinding their opcodes to the original access-equivalent opcode sets, so as to give the post-pass scheduler the freedom to pick the best option.

6.6 Assembly and object code emission

This phase is responsible for emitting either assembly code or object code for the program. It must observe all of the usual file format conventions regarding code and data segments, linking and relocation information, and symbolic debugging information. However, the aspect of this phase that we wish to highlight is the translation of the fully-qualified operations into the architectural operations that can actually be executed on the target machine.

In a fully-qualified operation, the opcode and each of the operands have already been bound to compiler-opcodes and compiler-registers, respectively. Each compiler-opcode has an associated bundle-macro descriptor which defines the implementation of the compiler-opcode. The compiler-opcode is replaced by the set of architectural opcodes and internal variables in the bundle-macro. Note that the architectural opcodes have all been implicitly scheduled by virtue of their fixed schedule time relative to one another. In like manner, each compiler-register is expanded into architectural registers as specified by the associated

register-package descriptor. At this point, assembly code may be emitted using the assembly code mnemonic information for the target architecture.

Object code assembly requires additional information regarding binary encodings of architectural operations and the available instruction templates. The machine description database must carry a list of available templates covering all legal combinations of architectural operations. Often, alternate templates that are shorter or have other architectural advantages (e.g. low power) may be used for frequently occurring operation combinations. The shortest template that can encode all the architectural operations scheduled on the same cycle is selected in order to reduce the code size. Unused operation slots are filled with no-ops. Finally, each instruction is encoded and emitted to the object file following object code format conventions.

7 Machine description in Elcor

We now present a brief description of the organization of the machine description database (mdes) in Elcor, our EPIC research compiler targeting the HPL-PD family of architectures [15]. A more complete discussion is provided in a companion technical report [49].

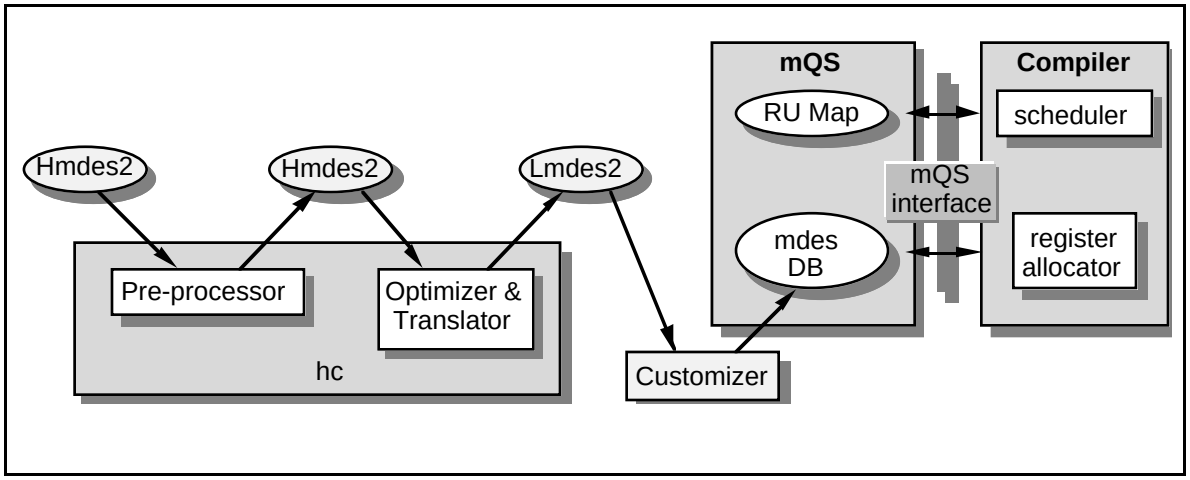

Figure 17: The mdes infrastructure in Elcor.

The mdes infrastructure in Elcor is shown in Figure 17. In order to make the compiler fully parameterized with respect to the target machine information, we separate the modules that need the information from the machine-description database in which this information is stored. The compiler modules are only allowed to make a fixed set of queries to the

database through an *mdes query system (mQS) interface*. Any form of database organization could be used for this purpose as long as the interface is well defined. However, specializing the internal structure of the database to expedite the more frequent queries results in substantial performance improvements.

In order to specify the machine description information externally in a textual form, the Elcor compiler uses the high-level machine description database specification Hmdes2 [50]. This external format is organized as a general relational database description language (DBL) which supports a high-level, human-editable textual form and a low-level machine-readable form. Tools are provided to expand macros in the high-level form and to compile it down to the low-level form. The exact definitions of the various database relations used by a database are also defined within the same framework and serve to specialize the file format for a given database schema.

As shown in Figure 17, the machine-description for a given target architecture is expressed in Hmdes2 format as a text file. After macro processing and compilation, the corrresponding low-level specification, expressed in Lmdes2, is loaded into Elcor using a *customizer* module that reads the specification and builds the internal data structures of the mdes database. A detailed description of our variation of Hmdes2, the internal data structure and the mQS interface are available in a companion technical report [49].

7.1 Machine-description database optimizations

An efficient implementation of a machine-description database differs quite substantially from the most direct and obvious organization that reflects the OBL. The improved organization can be viewed as the result of a series of optimizations with the naive organization as the starting point. We outline here the set of optimizations applied to the structure of the mdes in our implementation of it within Elcor [49]. Except for one optimization which is aimed at speeding up the queries made by the scheduler, their objective is to reduce the size of the database.

Typically, the most time-consuming part of EPIC code generation is scheduling. It makes sense, therefore, to accelerate the queries made by the scheduler. The most frequent query involves inspecting the set of alternatives corresponding to a given access-equivalent operation set. The naive representation of the OBL would be as a tree structure rooted in each access-equivalent operation set. The children would be the opcode-qualified operation sets, each of whose children would represent the alternatives. Executing this query would

entail a relatively slow walk over this tree. Instead, in our implementation, the access-equivalent operation sets point directly to a list of the corresponding alternatives, each of which has a reverse pointer to the opcode-qualified operation set for which it is an alternative. This speeds up the query, but at the cost of significant redundancy. Fortunately, the next optimization partially compensates for this.

The elimination of redundant information is always important. In the database, it results in reduced size. In the Hmdes2 specification, it leads to conciseness and a better ability to maintain consistency and correctness. The solution, which also serves as a space optimization, is to replace all of the duplicate items of information with a pointer to a single copy. Examples of information that typically are highly redundant and that benefit greatly from this optimization include the latency descriptors and the reservation tables within the operation descriptors. Often, this optimization creates further opportunities for itself. For instance, two alternatives, A and B, in different access-equivalent operation sets, that are identical except that they point to different, but identical, latency descriptors, C and D, respectively, become redundant once C and D have been replaced by pointers to the same record. A and B themselves can now be combined since they are identical.

Another space saving optimization is Cartesian factoring. One can take advantage of this opportunity when a (large) set of tuples can be represented as the Cartesian product of the sets in a tuple of (small) sets. The primary application of this optimization is to RS-tuples. Consider a set of operation sets that are identical except for their RS-tuples. First, the common part of the operation sets can be factored out using the previous optimization. Then the set of RS-tuples can be reduced to a tuple whose elements are sets of register sets. The resulting representation is not much larger than that for a single operation set.

One way to save space is to just not represent certain information. For instance, fully-qualified operations and register-qualified operation sets are not represented in the mdes. The register tuple, for an operation for which register allocation has been performed, is maintained in the compiler's intermediate representation (outside the mdes). The contextual knowledge of the register tuple to which an operation is bound, allows each opcode-qualified or access-equivalent operation set to also represent a fully-qualified operation or a register-qualified operation, respectively.

A final optimization is to replace data, that is infrequently or never queried, by a function that assembles that data on demand. One good example is the opcode set of an operation set. This is never examined since the scheduler works instead with the list of alternatives

for that operation set. The opcode set need not be represented explicitly in the database except at the leaves where the opcode set degenerates to a compiler-opcode. If it is needed, the opcode set can be constructed by walking the tree rooted in the operation set and assembling the compiler-opcodes at the leaves. In a sense, this is the reverse of the first optimization.

7.2 The mdes query system (mQS)

Information contained within the machine description database is made available to the Elcor compiler modules through a query interface. This insulates the modules from the internal structure of the database and makes the task of modifying the internal structure of the code generator much simpler. This interface consists of the following sets of queries:

1. Opcode and operand parameter queries
2. Register parameter queries
3. RU (Resource Usage) Manager queries
4. RMSL (Resource Minimum Schedule Length) Manager queries

In order to avoid any language-specific nuances, we describe below a slightly abstract version of the actual query functions in Elcor [49]. The function names appear in **bold** face and their arguments in *italics.* This is followed by a description of what the function does.

7.2.1 Opcode and operand parameter queries

In the following functions, the *opcode* argument specifies an access-equivalent opcode set or a compiler-opcode, the *alt* argument or return parameter specifies an alternative, the *ioreq* argument specifies the input/output requirement of an operation as an access-equivalent RS-tuple, and the *port* argument refers to the particular operand of the operation. We assume that all opcodes in an access-equivalent opcode set have the same set of opcode attributes (Section 5.2).

1. int **number_of_inputs**(*opcode*) - returns the number of input operands.
2. int **number_of_outputs**(*opcode*) - returns the number of output operands.
3. bool **is_predicated**(*opcode*) - returns whether or not the given opcode (set) has a guarding predicate input.

4. bool **has_speculative_version**(*opcode*) - returns whether or not the given opcode (set) has a corresponding speculative version. If so, the name of the speculative operation may be formed by adding a pre-specified modifier to the given opcode.

5. int **operation_priority**(*opcode*) - The HPL-PD architecture permits the specification of a relative priority among the various memory ports (load/store units) [15]. In a given machine, the memory ports either may or may not be prioritized. This query, which is valid only for compiler-opcodes, returns the priority of that opcode. The priority guarantees a sequential order for the memory operations issued within the same cycle, if they are to the same memory location. It therefore allows the scheduler to schedule in the same instruction memory operations that might, but are expected not to, have flow, anti- and output dependences between them. This is accomplished by assigning a zero latency to the edge between such operations, and assigning a higher priority opcode to the predecessor operation than to the successor operation.

3. int **earliest_read_time**(*opcode, ioreq, port*)

   int **latest_read_time**(*opcode, ioreq, port*)

   int **earliest_write_time**(*opcode, ioreq, port*)

   int **latest_write_time**(*opcode, ioreq, port*)

   These functions are used to obtain the latency parameters for any given operand (Section 5.4). The *opcode* and the *ioreq* arguments together identify either an access-equivalent operation set or an opcode-qualified operation[14]. In the former case, the latency values are computed as a summary of all the constituent alternatives (see Section 5.4), while in the latter case, exact values are returned.

7.2.2 Register parameter queries

The following query functions are used to obtain properties of (sets of) compiler-registers. The *regname* argument specifies an access-equivalent register set or, in some contexts, a particular compiler-register. By definition, all compiler-registers in an access-equivalent register set have the same set of storage attributes (Section 5.3).

---

[14] In the case where a compiler-opcode is shared across multiple alternatives, one may additionally need to pinpoint the specific alternative used in the opcode-qualified operation.

1. list<string> **available_reg_sets**() – returns the names of all the distinct access-equivalent register sets supported in the mdes. Note that there may be structural overlap between the registers of two distinct sets. For example, single-precision floating-point registers and their odd-even pairs constituting double-precision floating-point registers are distinct compiler-register sets that have structural overlap.

2. bool **reg_overlaps**(*regname1, regname2*) - returns whether or not there is a structural overlap between the two (sets of) compiler-registers. Two register sets are said to overlap if any register in one set has an overlap with any register in the other set.

3. int **reg_static_size**(*regname*) - returns the number of static registers in the given register set.

4. int **reg_rotating_size**(*regname*) - returns the number of rotating registers in the given register set.

5. int **reg_width**(*regname*) - returns the width of the given register (set).

6. bool **supports_rot_reg**(*regname*) - returns whether or not the given register (set) supports rotating register addressing.

7. bool **reg_has_speculative_bit**(*regname*) - returns whether or not the given register (set) has an extra tag bit to support speculative execution.

8. bool **reg_is_allocatable**(*regname*) - returns whether or not the given register (set) can be considered for register allocation. For instance, literal registers are not allocatable since they contain a fixed value.

7.2.3 RU manager queries

As described earlier, the scheduler needs to select operation alternatives that do not conflict with the resource usage of the current partial schedule. Rather than directly expose the internal structure of the database to the scheduler, we provide an independent *resource usage manager (RU manager)* that does all the book-keeping on behalf of the scheduler. The RU manager is responsible for directly interfacing with the machine description database and allocating and manipulating the resource usage map as instructed by the scheduler; it does not make any scheduling decisions on its own. The following queries describe this interface:

1. void **alloc_RUmap**(*maxlength*)

   void **delete_RUmap**()

   void **print_RUmap**(*stream*)

   void **init_RUmap**(*is_modulo, length*)

   The above functions allocate, delete, print and initialize the internal resource usage map respectively. A map may be allocated once and used many times as long as it is re-initialized after each use. The initialization specifies whether the map is to be used for modulo-scheduling or not, and if so, the initiation interval of the modulo-schedule needs to be specified. The only difference between a regular RU map and a modulo-scheduled RU map is that in case of the latter the scheduling cycle is computed using modulo-arithmetic with respect to the given initiation interval [31].

2. alt **get_next_nonconfl_alt**(*opcode, ioreq, time*)

   void **place_alt**(*alt, time*)

   void **remove_alt**(*alt, time*)

   These functions perform the basic tasks of scheduling. The first function may be called repeatedly to obtain the next available non-conflicting alternative for the given access-equivalent operation set (as specified by the *opcode* and the *ioreq* arguments) at the given scheduling time. The returned alternative, if found, identifies the compiler-opcode and its priority. Internally, the resource manager scans through each of the alternatives present under the specified access-equivalent operation set, and matches its reservation table offset by the given scheduling time with the remaining available slots in the resource usage map. The function behaves like an iterator; each call starts the search from where the previous call left it.

   The second function is called when the scheduler decides to commit an alternative into the schedule at a given scheduling time. The schedule is committed by offseting the reservation table of the given alternative by the given scheduling time and then marking the corresponding slots in the resource usage map as used.

   The third function is used to deschedule the given opcode-qualified operation, specified as the alternative that was previously scheduled at the given time, from the internal resource usage map.

3. list<alt> **get_conflicting_ops**(*opcode, ioreq, time*) - If the scheduler needs to backtrack, this function returns all the previously scheduled alternatives that conflict with any alternative of the given access-equivalent operation set (as specified by the *opcode* and the *ioreq* arguments) if scheduled at the current cycle. The idea is that if all these alternatives were to be descheduled, the scheduler would then have complete freedom in scheduling the current operation.

7.2.4 RMSL manager queries

For certain analyses, a lower bound of the resource requirements of a given set of operations may be desired (e.g., the computation of the resource-limited bound, ResMII, on the initiation interval for modulo scheduling of loops [31]). Complete scheduling is far too accurate and slow for this purpose. The *resource minimum schedule length manager (RMSL manager)* provides an alternate interface that may be used to compute a lower bound on the resource usage. An RMSL map is similar to an RU map except that it only keeps the total number of the various resources needed by the selected alternatives rather than laying them out in a timeline. This interface is described below:

1. void **alloc_RMSLmap**()

   void **dealloc_RMSLmap**()

   void **init_RMSLmap**()

   These functions allocate, deallocate and initialize the internal resource counters for the computation of the resource lower bound.

2. **accumulate_nextop**(*opcode, ioreq*) - accumulates the resource counts of the best alternative for the given access-equivalent operation set (as specified by the *opcode* and the *ioreq* arguments). It selects the least resource-critical alternative based on the current accumulation, i.e., the alternative that advances the maximum resource count across all resources by the minimum amount. In this computation, the times of the resource usages are ignored; only the specific resources used are counted. (Note that this is only a heuristic for computing the ResMII.)

3. int **resource_lower_bound**() - returns the currently accumulated resource lower bound as the maximum resource usage count over all resources.

7.3 Code quality of mdes-driven compilers

A natural concern has to do with how much is compromised, in terms of performance or code quality, by writing a compiler in an mdes-driven fashion. In addressing this issue, it is valuable to distinguish between the three factors that impinge upon it.

The first one is that we are choosing to restrict our interest to the space of EPIC processors outlined in Section 2. As pointed out in the Introduction, both the first generation of VLIW and EPIC were consciously restricted to a style of architecture that lends itself to having a high-quality compiler written for it. There is a definite trade-off here. For the same peak performance, a traditional style of DSP will almost surely be smaller and cheaper than an EPIC processor. This is because DSPs are essentially special-purpose processors, very finely tuned to perform certain computations (most notably the inner-product). On the other hand, the EPIC processor is able to achieve a larger fraction of its peak performance on a larger set of computations and, most importantly, it is possible to do so via a compiler.

The second factor is the phase ordering that we have chosen, and its impact on code quality, given that one has elected to stay within our space of EPIC processors. The one fact that is clear is that, for even a moderately parallel EPIC processor, attempting to perform all the steps as one joint optimization is quite impractical from the viewpoint of computational complexity. Some phase ordering is essential. It is important to remember that the space of EPIC processors was defined so as to make high quality, phase ordered compilers possible. An example is the presence of a relatively large number of registers per register file so that the scheduler can perform its task viewing register spill as a somewhat exceptional event, thereby allowing register allocation to be performed as a subsequent phase.

Finally, given our space of EPIC processors and our phase ordering, the question that can be asked is whether being mdes-driven causes the compiler to generate lower quality code. We are not aware of any situation in which querying the mdes for machine information rather than hard-coding it into the compiler would have helped generate better code, except in the following indirect sense. Writing an mdes-driven compiler presupposes two things. One is that it is possible to even write a high quality compiler for the target processor. The second is that it is possible to formalize the code generation process. Both involve restricting oneself to a space of processors that are sufficiently compiler-friendly. As we have noted, such processors can be less cost-effective on narrow workloads, and in this

sense, and only in this sense, does one pay a price if one wishes to use an mdes-driven compiler.

8 Related work

8.1 Antecedents

The earliest work, of which we are aware, that used a reservation table to model the hardware resource usage of operations in a computer was that by Davidson and his students [42]. Davidson coined the term and developed a theory for optimally scheduling operations that share resources within a multi-function pipeline. This model of resource usage has been rediscovered multiple times since then, e.g., as templates [51] and as SRUs (set of resource usages) [52].

In the early 1980's, mdes-driven compilers [28, 29] were developed for the VLIW mini-supercomputer products that were built and marketed by Cydrome [13] and Multiflow [12]. The work discussed in this paper is a direct descendant of the Cydrome work on mdes-driven compilers which was motivated by the very pragmatic need to be able to develop a compiler while the details of the machine were still in flux. The machine modelling capability had to be quite sophisticated since the Cydra 5 was a real product for which design decisions had to be made which violated many of the simplifying assumptions often made by researchers (such as single cycle operations, fully pipelined functional units, simple reservation tables and homogeneous, even identical, functional units).

This work was continued in 1989 as part of the EPIC research at HP Laboratories by the authors, but with the motivation of being able to experiment with and evaluate widely varying EPIC processors. Subsequently, in collaboration with the University of Illinois' IMPACT project, the mdes-driven capability was further developed with an eye to supporting certain aspects of superscalar processors. Additionally, a high level, human-friendly machine description language was developed to facilitate the definition of the mdes for a machine [53, 50]. HP Laboratories' Elcor compiler and the University of Illinois' IMPACT compiler are both mdes-driven as far as scheduling and register allocation are concerned.

8.2 The PICO project: processor-compiler co-design

The retargetable compiler technology described in this report forms the central component of our overall research infrastructure that also includes tools for automatically exploring the

space of application specific EPIC architectures, the design and synthesis of a target architecture and the automatic extraction of a machine description database for that target. This effectively retargets the Elcor compiler to that architecture in order to evaluate its performance. This overall infrastructure is called PICO (Program In Chip Out). A detailed description of this system is considerably beyond the scope of this report. Instead, we provide here just a brief overview.

Given an application for which a custom EPIC architecture needs to be designed, we first use the Elcor compiler to obtain opcode usage statistics that drive the architecture design space exploration. The spacewalker explores the architectural design space, varying of the order of twenty design parameters such as the number and kinds of functional units, register files and their resource usage constraints. Each architectural design point is then synthesized into our structural representation, which we call the Architecture Intermediate Representation (AIR), that represents the detailed block-level connectivity of the machine. We then automatically extract from AIR the relevant machine description information needed in the mdes, which drives the Elcor compiler during scheduling and register allocation. After scheduling, instruction templates and formats are designed that optimize the total code size of the program based on the schedule generated by Elcor. The instruction fetch and decode circuitry is then synthesized and added to the AIR description at which point the structural VHDL can be emitted and the cost of the machine can be computed. Finally, the code can be assembled, using the chosen instruction templates and formats, and emitted for execution on the target architecture.

The ability to perform such processor-compiler codesign stems from the fact that we can automatically extract the machine description information of the type described in this report from a structural description of the processor and that, once this is done, our mdes-driven compiler is automatically retargeted to that processor.

8.3 Other work

The work on retargetable code generation for general-purpose sequential processors [16-20] has been primarily focused on taking advantage of CISC opcodes. Although this work is applicable to EPIC compilation, it is largely orthogonal and complementary to the theme of this paper. Our primary concern lies not with the translation from semantic to generic operation sets, but with binding the generic operation sets to architectural operations efficiently while minimizing the resulting schedule length.

Also, the work done in the embedded processor community on the topic of retargetable compilers [22, 24, 21, 4, 23, 5, 25] is largely orthogonal to the focus of this paper. In addition to the opcode pattern matching that was the concern of the general-purpose processing community, researchers in the embedded processor area have also had to contend with the heterogeneous registers and the irregular connectivity of contemporary DSPs. This has required that the mapping from semantic operations to architectural operations be performed in one step, in contrast to the phase ordering that we have put forth in this paper. Accordingly, the pattern matching paradigm has been extended to include not just the opcodes, but the registers and interconnect as well. Essential as this is to those targeting DSPs, it is only of secondary importance to those targeting EPIC processors with higher levels of ILP.

However, there are some interesting parallel concepts that are worth highlighting. One is that our machine description viewpoint is similar to that in nML [24] in that the machine description is from an architectural or programmer's viewpoint, rather than being structural in nature. It differs from the CBC view of compilation [24], in that our mdes does not explicitly specify the legal instructions templates (the sets of operations that can be specified simultaneously in one EPIC instruction). Instead, the scheduler (implicitly) assembles such information dynamically based on the reservation tables which specify how operations can conflict, with respect to both the instruction formats and the usage of the other resources of the machine. (However, our code emission phase does consult an AND-OR instruction format tree to specify and use the correct instruction format corresponding to a given set of legal concurrently issued operations.)

Our bundle-macros are similar in concept to the bundles created in the CHESS compiler [22] except that we assume that bundle-macros are defined off-line, while creating the mdes, whereas the CHESS compiler defines the relevant bundles during compilation. Presumably, techniques similar to the instruction-set extraction capability in the Record compiler [54] or the pattern extraction capability in the CBC compiler could be used to define our bundle-macros automatically.

Our access-equivalent operation sets (implicitly) specify the equivalent of the Instruction Set Graph (ISG) [22], i.e., operations and their connectivity to storage elements. However, our capability only models register-to-register behavior between static storage resources; transitory storage resource usages are buried inside our bundle-macros. Our access-equivalent operation sets concentrate on the logical connectivity between opcodes and registers, while abstracting away from the interconnect structure and the potential

structural hazards inherent in its usage. Instead, in our scheme, all structural hazards are accounted for in our reservation tables which also accounts for instruction encoding conflicts.

Our expanded computation graph is similar to the Trellis diagram [23]. A point of difference is that the trellis diagram splits each register option into multiple states to support the code generation process. Whereas the trellis diagram is intended to be used to actually generate the code, corresponding to an expression tree, for DSPs, the expanded computation graph is just a conceptual tool to help us develop the notion of access-equivalent operation sets which are of much greater practical value in generating code for an EPIC processor.

The Instruction Set Definition language (ISDL) [55] developed at MIT is another machine description language which, like ours, takes an architectural or programmer's viewpoint rather than a structural viewpoint. However, it differs substantially in the two areas that are central to scheduling and register allocation. Firstly, it uses a very simple model for latencies. Secondly, similar to the CBC view of compilation, it uses the notion of constraints to define valid operation groupings within an instruction. ISDL is used by a retargetable code generator, called AVIV [25]. Unlike the phase ordered approach presented in this paper, AVIV simultaneously performs functional unit binding, register bank allocation and scheduling. The split node DAG used in AVIV to enumerate a candidate set of schedules is similar to the expanded computation graph described in Section 3 except that register options are not represented explicitly in the split node DAG. AVIV uses the constraints specified in the ISDL description of the target architecture to filter out those schedules that would be legal if only data dependences were taken into account, but which are actually illegal for the target machine. ISDL's use of constraints to represent resource conflicts seems more suited for an enumerate-and-check approach for generating valid schedules, whereas our use of reservation tables to model resource conflicts allows us to directly and incrementally construct valid schedules (and only the valid ones).

An interesting sub-topic of mdes-driven code generation for EPIC processors is that of machine description optimization. Both scheduling time and mdes size can be decreased if the reservation tables are represented as AND/OR trees of reservation table fragments [56]. This increases the amount of factorization of common pieces of reservation tables and reduces the size of the database by reusing this information. A complementary optimization is to construct a simpler set of synthetic reservation tables corresponding to a smaller

number of synthetic resources but which are equivalent to the original reservation tables with respect to the constraints that they place upon the legal relative schedule times of every pair of operations [57].

9 Conclusions

In this report, we have articulated the unique challenges faced by the code generator for an EPIC processor. The key challenge is the typically large number of options available to the EPIC code generator for implementing a given semantic operation, and the expectations placed upon it to take advantage of the multiplicity of available options to yield good schedules. The problem faced by the code generator is that a premature binding of opcodes and registers leads to very poor schedules. Ideally, the scheduler and register allocator should be given full freedom to choose any option, but this can lead to incorrect code. The strategy of performing partitioning, scheduling and register allocation as one integrated phase, is completely impractical given the very large number of options in a processor designed for a high level of instruction-level parallelism.

The strategy that we have articulated is one of incremental binding and delayed code selection, distributed over the entire code generation process, with a view to giving the scheduler and register allocator maximal freedom of choice without sacrificing correctness or efficiency. In support of this, we introduced three key concepts: the expanded computation graph, the notion of full-connectedness, and the necessity of annotating the operators and variables in the computation graph with fully-connected opcode and register option sets at the point just prior to scheduling and register allocation. This, we argued, is necessary for maintaining efficiency while giving maximal freedom to the scheduler and register allocator.

We also introduced the concept of access-equivalent operation sets, which makes it possible to articulate a practical and efficient procedure for annotating the operators and variables of a computation graph with access-equivalent opcode and register option sets. Furthermore, the access-equivalent operation sets for a given target machine can be computed once and for all when constructing the machine-description database, instead of repeatedly doing so at compile-time.

We built upon these concepts and defined a formal model of the binding process from semantic operations down to architectural operations. The central concept here is that of the Operation Binding Lattice (OBL), which reflects a judicious phase-ordering for EPIC code

generators. The phases are: initial code selection, pre-pass operation binding (including partitioning if needed), scheduling (the key step in opcode selection), register allocation (the penultimate step in register binding), post-pass scheduling (the penultimate step in opcode selection), and code emission (the final step of delayed code selection). We also explained how this phase ordering refines the access-equivalent operation sets for a processor.

We defined the information that must be provided in the machine-description database in order to support our preferred phase ordering, the OBL, and the corresponding binding process. The result is an abstract, compiler-centric view of the processor, which is derived from the programmer's view of the machine, i.e., the Architecture Manual, rather than a structural description of the hardware. Moreover, the structure of this machine-description database supports realistic EPIC processors with all of their attendent complications, e.g., operations with complex, multi-cycle resource usages and differential latencies for their source and destination operands.

The key advantage of our approach is that a large part of the analysis of the target processor, needed for good code generation, has been taken off-line instead of being done repeatedly during compilation. The definition of the operation, opcode and register descriptors in the Operation Binding Lattice (including the generic and access-equivalent operation sets, and the opcode-qualified, register-qualified, and fully-qualified operations) can all be performed at the time of building the machine-description database. Similarly, the bundle-macros, reservation tables, and latency descriptors can all be defined off-line. Once this information has been defined for a given target processor, and it has been loaded into the machine-description database, the retargeting of the code generator has been completed.

We briefly described our implementation of the above concepts in Elcor, our EPIC research compiler, and the various types of optimizations that we have applied to the structure of its machine-description database in order to reduce its size and speed up important queries. We also described the mdes Query System, which provides an information-hiding interface between the code generator modules and the mdes.

At this point, the concepts discussed in this report have been developed and tested across one or other of three thorough implementations: the product-quality Cydra 5 compiler [28], our Elcor research compiler which is routinely and automatically retargeted to hundreds of EPIC ASIPs, and the University of Illinois' IMPACT compiler [30] which has been retargeted to half a dozen commercial microprocessors.

References

1. ST18950 User Manual. (SGS-Thompson Microelectronics, 1993).

2. TMS320C2x User's Guide. (Texas Instruments, 1993).

3. DSP56000 24-bit Digital Signal Processor Family Manual. (Motorola, Inc., 1995).

4. P. Marwedel and G. Goossens (Editor). Code Generation for Embedded Processors. (Kluwer Academic Publishers, Boston, Massachussetts, 1995).

5. C. Liem. Retargetable Compilers for Embedded Core Processors. (Kluwer Academic Publishers, Dordrecht, The Netherlands, 1997).

6. R. Leupers. Retargetable Code Generation for Digital Signal Processors. (Kluwer Academic Publishers, Dordrecht. The Netherlands, 1997).

7. A. E. Charlesworth. An approach to scientific array processing: the architectural design of the AP-120B/FPS-164 Family. Computer 14, 9 (1981), 18-27.

8. B. R. Rau, C. D. Glaeser and E. M. Greenawalt. Architectural support for the efficient generation of code for horizontal architectures. Proc. Symposium on Architectural Support for Programming Languages and Operating Systems (Palo Alto, March 1982), 96-99.

9. J. A. Fisher. Very long instruction word architectures and the ELI-512. Proc. Tenth Annual International Symposium on Computer Architecture (Stockholm, Sweden, June 1983), 140-150.

10. TMS320C62xx CPU and Instruction Set Reference Guide. (Texas Instruments, 1997).

11. Trimedia TM-1 Media Processor Data Book. (Philips Semiconductors, Trimedia Product Group, 1997).

12. R. P. Colwell, R. P. Nix, J. J. O'Donnell, D. B. Papworth and P. K. Rodman. A VLIW architecture for a trace scheduling compiler. IEEE Transactions on Computers C-37, 8 (August 1988), 967-979.

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

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

15. V. Kathail, M. Schlansker and B. R. Rau. HPL-PD Architecture Specification: Version 1.1. Technical Report HPL-93-80 (R.1). Hewlett-Packard Laboratories, September 1998.

16. R. G. G. Cattell. Formalization and Automatic Derivation of Code Generators. Ph.D. Thesis. Carnegie-Mellon University, Pittsburgh, 1978.

17. R. S. Glanville and S. L. Graham. A new method for compiler code generation. Proc. 5th Annual ACM Symposium on Principles of Programming Languages (1978), 231-240.

18. M. Ganapathi, C. N. Fisher and J. L. Hennessy. Retargetable compiler code generation. ACM Computing Surveys 14, 4 (December 1992), 573-592.

19. C. W. Fraser, D. R. Hanson and T. A. Proebsting. Engineering a simple, efficient code-generator generator. ACM Letters of Programming Languages and Systems 1, 3 (September 1992), 213-226.

20. R. M. Stallman. Using and porting GNU CC, version 2.4. Free Software Foundation, June 1993.

21. P. G. Paulin, C. Liem, T. C. May and S. Sutarwala. FlexWare: a flexible firmware development environment for embedded systems, in Code Generation for Embedded Processors, P. Marwedel and G. Goossens (Editor). (Kluwer Academic Publishers, 1995), 65-84.

22. D. Lanneer, J. Van Praet, A. K. Kifli, K. Schoofs, W. Geurts, F. Thoen and G. Goossens. CHESS: retargetable code generation for embedded DSP processors, in Code Generation for Embedded Processors, P. Marwedel and G. Goossens (Editor). (Kluwer Academic Publishers, 1995), 85-102.

23. B. Wess. Code generation based on trellis diagrams, in Code Generation for Embedded Processors, P. Marwedel and G. Goossens (Editor). (Kluwer Academic Publishers, 1995), 188-202.

24. A. Fauth. Beyond tool specific machine descriptions, in Code Generation for Embedded Processors, P. Marwedel and G. Goossens (Editor). (Kluwer Academic Publishers, 1995), 138-152.

25. S. Hanono and S. Devadas. Instruction selection, resource allocation, and scheduling in the Aviv retargetable code generator. Proc. ACM/IEEE Design Automation Conference (1998).

26. J. A. Fisher. Trace scheduling: a technique for global microcode compaction. IEEE Transactions on Computers C-30, 7 (July 1981), 478-490.

27. B. R. Rau and C. D. Glaeser. Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. Proc. Fourteenth Annual Workshop on Microprogramming (October 1981), 183-198.

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

29. P. G. Lowney, S. M. Freudenberger, T. J. Karzes, W. D. Lichtenstein, R. P. Nix, J. S. O'Donnell and J. C. Ruttenberg. The Multiflow trace scheduling compiler. The Journal of Supercomputing 7, 1/2 (May 1993), 51-142.

30. W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm and D. M. Lavery. The superblock: an effective technique for VLIW and superscalar compilation. The Journal of Supercomputing 7, 1/2 (May 1993), 229-248.

31. B. R. Rau. Iterative modulo scheduling. International Journal of Parallel Processing 24, 1 (February 1996), 3-64.

32. M. S. Schlansker, V. Kathail and S. Anik. Parallelization of control recurrences for ILP processors. International Journal of Parallel Processing 24, 1 (February 1996), 65-102.

33. M. S. Schlansker, S. A. Mahlke and R. A. Johnson. Bypassing the Branch Bottleneck Using Control Critical Path Reduction. Technical Report. Hewlett-Packard Laboratories, (to appear) 1998.

34. J. R. Ellis. Bulldog: A Compiler for VLIW Architectures. (The MIT Press, Cambridge, Massachussetts, 1985).

35. G. Desoli. Instruction Assignment for Clustered VLIW DSP Compilers: A New Approach. HPL Technical Report HPL-98-13. Hewlett-Packard Laboratories, February 1998.

36. E. Nystrom and A. E. Eichenberger. Effective Cluster Assignment For Modulo Scheduling. Technical Report. Department of Electrical and Computer Engineering, North Carolina State University, June 1998.

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

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

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

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

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

42. E. S. Davidson, L. E. Shar, A. T. Thomas and J. H. Patel. Effective control for pipelined computers. Proc. COMPCON '90 (San Francisco, February 1975), 181-184.

43. C. W. Fraser and D. Hanson. A retargetable compiler for ANSI C. ACM SIGPLAN Notices 26, 10 (October 1991), 29-43.

44. C. W. Fraser and R. R. Henry. BURG--Fast optimal instruction selection and tree parsing. ACM SIGPLAN Notices 27, 4 (April 1992), 68-76.

45. M. S. Schlansker and V. Kathail. Critical path reduction for scalar programs. Proc. 28th Annual International Symposium on Microarchitecture (Ann Arbor, Michigan, November 1995), 57-69.

46. A. E. Eichenberger and E. S. Davidson. Register allocation for predicated code. Proc. 28th Annual International Symposium on Microarchitecture (Ann Arbor, Michigan, November 1995), 180-191.

47. D. M. Gillies, D.-c. R. Ju, R. Johnson and M. Schlansker. Global predicate analysis and its application to register allocation. Proc. 29th Annual IEEE/ACM International Symposium on Microarchitecture (Paris, France, December 1996), 114-125.

48. B. R. Rau, M. Lee, P. Tirumalai and M. S. Schlansker. Register allocation for software pipelined loops. Proc. SIGPLAN'92 Conference on Programming Language Design and Implementation (San Francisco, June 17-19 1992).

49. S. Aditya, V. Kathail and B. R. Rau. Elcor's Machine Description System: Version 3.0. HPL Technical Report HPL-98-128. Hewlett-Packard Laboratories, September 1998.

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

51. M. Tokoro, E. Tamura, K. Takase and K. Tamaru. An approach to microprogram optimization considering resource occupancy and instruction formats. Proc. 10th Annual Workshop on Microprogramming (Niagara Falls, New York, November 1977), 92-108.

52. W. Schenk. Retargetable code generation for parallel, pipelined processor structures, in Code Generation for Embedded Processors, P. Marwedel and G. Goossens (Editor). (Kluwer Academic Publishers, 1995), 119-135.

53. J. C. Gyllenhaal. A Machine Description Language for Compilation. M. S. Thesis. Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, 1994.

54. R. Leupers and P. Marwedel. Retargetable generation of code selectors from HDL processor models. Proc. European Design and Test Conference (March 1997), 140-144.

55. G. Hadjiyiannis, S. Hanono and S. Devadas. ISDL: An instruction set description language for retargetability. Proc. ACM/IEEE Design Automation Conference (1997).

56. J. C. Gyllenhaal, W.-m. W. Hwu and B. R. Rau. Optimization of machine descriptions for efficient use. Proc. 29th Annual IEEE/ACM International Symposium on Microarchitecture (Paris, France, December 1996), 349-358.

57. A. E. Eichenberger and E. S. Davidson. A reduced multipipeline machine description that preserves scheduling constraints. Proc. SIGPLAN'96 Conference on

Programming Language Design and Implementation (Philadelphia, Pennsylvania, May 1996), 12-20.