A TEMPLATE FOR CODE GENERATOR DEVELOPMENT USING THE IMPACT-I C COMPILER

BY

ROGER ALEXANDER BRINGMANN

B.S., University of South Alabama, 1983

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1992

Urbana, Illinois

ACKNOWLEDGMENTS

I would first like to thank my advisor, Professor Wen-mei Hwu, for his guidance and his support. Without his faith in me at the beginning, I would not have had the opportunity to accomplish this goal. I have learned a great deal from him during the past two years in graduate school, and look forward to future opportunities.

I wish to extend my appreciation to William Chen and Scott Mahlke for their support and patience during this project. I would also like to thank them and John Holm for their work on the Scheduler and Rick Hank for his work on the Register Allocator. Without their help, this project would have been only a shell!

Finally, I wish to thank my wife Mary and my children Carrie and Johnathan. Their continued love and support has helped me make it through some very trying times during my adjustment to graduate school.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1. INTRODUCTION

A key element in the success of a compiler for research purposes or as a viable commercial product is the number and type of target processors supported. A compiler that supports multiple target processors will typically use a generic intermediate language or data structure when representing the converted or optimized instructions.

Code generation is the process of converting this intermediate language into the assembly instructions for a target processor. To successfully support a target processor, the developer must have an in-depth knowledge not only of the processor instruction set but of the organization of the processor. It is also crucial to know the programming conventions established by the manufacturer of the processor such as the allocation of specific registers for operating system use. The architecture and instruction set information are typically found in the processor *user's manual*. However, the established conventions may only be found within internal documents at the manufacturer.

During the 1990 and 1991 calendar years, extensive effort was spent in the development of code generators for the IMPACT-I C compiler. Code generators were developed for the MIPS R2000/R3000, SPARC, AMD-29000 and Intel i860 RISC processors. These projects were handled independently with varying levels of redundant effort. Even though all code generators started with a common data stream called Lcode [1] (Low-level intermediate code), the resulting code typically used incompatible data structure. These data structures required extensive massaging to use common tools such as the register allocator. Since development efforts were carried on independently, a global view of the interface requirements for a generic register allocator was not considered. Thus, common tasks such as physical register assignment were handled within the code generator as opposed to the register allocator.

When the time came to support machine level optimizations and instruction scheduling, the problems encountered up to that point were evaluated and a template guided code generator model was proposed to produce a uniform development methodology across all code generators. All key component interfaces were re-evaluated and extended as necessary

This thesis is intended as the primary design and interface specification for the template based development efforts. Its goal is to describe the code generation process and its surrounding environment, provide interface specifications for support tools that simplify the development process, and specify what parts of the code generator template are standard and what parts are processor specific. For each phase of code generation, support routines are outlined and examples of their use are given.

Chapter 2 provides an overview of the structure of the code generator template. Chapter 3 describes the operation and directory structure for new code generators. Chapters 4 through 7 provide the detailed information required to develop a code generator using the template. Finally, concluding remarks and performance results are given in Chapter 8.

2. OVERVIEW TO CODE GENERATION USING IMPACT-I

The creation of a code generator for IMPACT-I involves two development tasks. First, a machine specific file (***mspec***) must be developed. This file conveys the architectural characteristics and limitations of the target processor to earlier compilation phases. For example, if code is generated to reference variables, the correct reference scheme should be used such as signed offset from the stack. This will be discussed in detail in Section 3.

Once the *mspec* file has been developed, efforts can begin on the code generator. This process converts the original Lcode instructions into a representation called Mcode which can be more readily supported by the target processor. Mcode is composed of Lcode instructions which may be directly supported with no changes, and instructions which have be converted to a sequence sequence that can be supported. Section 2.1 briefly outlines the structure and function of each of the major components of a code generator using the new template. Section 2.2 introduces Lcode and Section 2.3 discusses the extensions to Lcode that embody Mcode.

2.1. Code Generator Structure

There are three main phases to code generation as shown in Figure 2.1. Phase I performs code annotation. Phase II performs Mcode optimization and register allocation. Phase III converts the final Mcode file to the appropriate assembly code for the target machine. These phases will be discussed in detail in the following sections.

2.1.1. Code Annotation

This phase involves the conversion from an Lcode format to an Mcode format. During this process, instructions with delay slots are marked and machine specific delays will be added to the instruction. Any unsupported Lcode instructions will be converted to supported Mcode instructions. For example, a frequent annotation involves incompatible sizes of immediate

values. The Lcode instructions can support 32-bit integer, floating point and double immediate. Few RISC architectures can support immediate fields of this magnitude.

2.1.2. Optimization and Register Allocation

Phase I code annotation can introduce redundant instructions. Therefore, the first task of this phase is to perform Mcode optimizations. The supported optimizations include common sub expression elimination, copy propagation and dead code elimination [4].

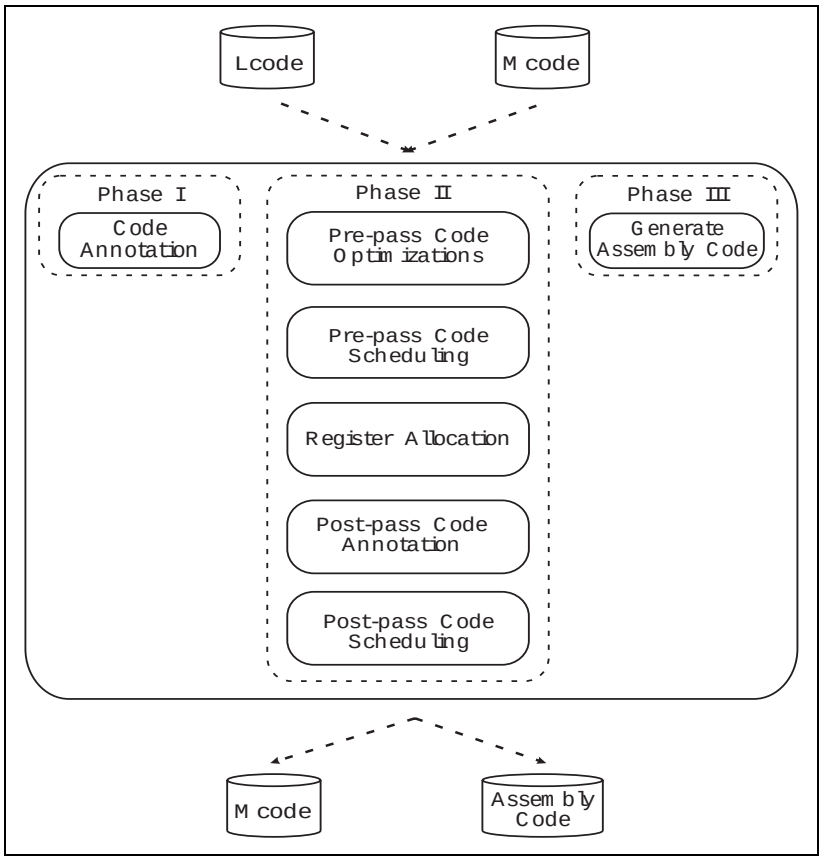

Figure 2.1: A Block Diagram of the IMPACT-I Code Generator

Pre-pass code scheduling has been shown [5] to provide a better schedule since it arranges instructions before register allocation. Register allocation can place artificial dependencies between instructions due to limitations in register files. To minimize these dependencies, pre-pass code scheduling moves instructions closer to their potential fill slots, thus increasing the live ranges. Unfortunately the increased live range increases the likelihood of spill code.

A generic register allocator will convert the virtual registers to the machine specific physical register number. The register allocator will decide whether to use a callee save or caller save register. If there are insufficient registers or the remaining registers would be a poor choice for selection, the register allocator will insert necessary spill and fill code to break up the live range of the register.

Post-pass code annotation and optimizations are performed after register allocation. Instructions that are dependent on information provided by the register allocator that is necessary to fill in the function prologue and epilogue will be annotated at this point. The optimizations are typically machine specific peep-hole optimizations.

Finally, post-pass code scheduling fills the load and branch delay slots as well as additional instructions that were marked as fill candidates.

2.1.3. Producing Assembly Code

The third phase will take the optimized Mcode file and produce appropriately formatted assembly code suitable for use with the standard available assemblers for the respective target machine. An in-depth knowledge of the machine instruction set is required for this phase.

2.2. An Introduction to Lcode

In order to understand the code generation process, one must first be familiar with the structure of Lcode files and key internal data structures. Lcode instructions are described with a three address notation. Lcode is an instruction set for a load/store architecture, supporting unlimited virtual registers and basic synchronization operations [1]. It is broken down into data and function blocks. The functions are composed of control blocks containing the Lcode instructions. Different phases of compilation can change the Lcode instructions. These phases will be highlighted as Lcode is discussed in subsequent sections.

The sections are broken down into the external representation of Lcode that is an ASCII representation and the internal representation that describes the most important data structures used within the C programs.

2.2.1. External Lcode Representation

2.2.1.1. File Layout

A typical Lcode file may be composed of a data Section and one or more functions. The data Section defines both static and dynamic variables. These variables are aligned to memory locations specified by the ***mspec*** file. Data types include unsigned character (1 byte), signed character (1 byte), unsigned short (2 bytes), signed short (2 bytes), unsigned integer (4 bytes), signed integer (4 bytes), single-precision float (4 bytes) and double-precision float (8 bytes). All integers are expressed as 2s complement number representation and all floating-point values are expressed in IEEE floating-point representation [1].

```
(function name exec_cnt)
    (cb "prologue")
    (cb ...)
        .
        .
        .
    (cb "epilogue")
(end name)
```

Figure 2.2: Lcode Function Layout

If inline expansion [2] is chosen as one of the compile-time optimizations, the program will be broken up into one file per function. This simplifies the merging process required by inlining. The data Section of the original program will be placed into a separate file called ***data.lc***. The file ***impact1_rename*** tells the name of each function stored in each created file.

2.2.1.2. Function Layout

Figure 2.2 shows the layout of a typical function in a Lcode file. The first line of a function, preceded by the keyword **function,** contains the name of the function (*name*) and the typical execution count of the function (*exec_cnt*). Execution count information may be gained

accurately from profiling [3] or estimated using static analysis. The first control block always contains the prologue that defines return register type, local variable requirements in bytes and outgoing parameter requirements in bytes.

As a minimum, a function will contain a control block with the prologue and a last control block with the epilogue. Therefore, there are a minimum of two control blocks per function. A function will never have more than one entry point, but may have multiple exit points. It is possible to have the epilogue duplicated in more than one control block because of one or more of the compiler optimizations [3].

2.2.1.3. Control Block Layout

Figure 2.3 shows the layout of a typical control block. Control blocks may have one or more exit points, but only one entry point. Control blocks with one exit point are the same as basic blocks. Control blocks with multiple exit points are called *superblocks* [3].

A control block begins with the keyword **cb** followed by a unique identification number (*id*) and an execution count (*cb_exec_cnt*). The execution count shows profiled-based or static information about number of times the control block has been executed.

```
(cb id cb_exec_cnt
    (flow cond dest cnt)
        …
    (flow cond dest cnt) )
    (op …)
        …
    (op …)
    [(branch_op …)]
        …
    [(branch_op …)]
```

Figure 2.3: Lcode Control Block Layout

Next, the flow information are given and preceded by the keyword **flow**. Flow information is composed of three fields: the condition by which the destination will be followed (*cond*), a destination control block number if the condition is met (*dest*), and the number of times that the

condition was met during profiling (*cnt*). It should be noted that unless profile information is available, the cnt field here will always be 0.0 for all paths.

If a control block has no branch at the end, the condition field is ignored and the destination is the next sequential control block. The condition field also will be ignored if the last instruction of the basic block is an unconditional branch. A basic block with a conditional branch will contain flow information for the taken and not taken cases.

Superblocks may contain multiple conditional branches but only one unconditional branch. Due to the multiple exit points in superblocks, Lcode instructions do not necessarily have the same execution count as specified in the control block header. The execution count of any instruction following a branch is computed by subtracting the *cnt* fields of all preceding branch instructions from *cb_exec_cnt*.

2.2.1.4. Instruction Layout

Lcode instruction begins with the entry **op** and is composed of four major parts, operation number, opcode, operands and attributes, as shown in Figure 2.4. The operation number is unique for each Lcode instruction in the function.

The opcode signifies the Lcode operation to perform. These groups are broken down into ALU operations, memory access and control flow. For a detailed description of each opcode and its required parameters refer to [1].

```
(op op_num
    (opcode
        ((dest_type dest_value))
        ((src1_type src1_value)
         (src2_type src2_value)
         (src3_type src3_value))
        ((attr_name attr_value) … )
    )
)
```

Figure 2.4: Lcode Instruction Layout

Lcode instructions support one destination and up to three source fields. The opcode shows the number and type of fields required at any given time. However, in some instances such as

memory stores, the third source field is used as the destination of the store. This is to prevent any incorrect register dependencies when using address indirect.

Each operand has two fields. The first field is a type that can be register (r), label (l), integer (i), single precession floating point (f), double precession floating point (f2), a macro (mac) or a control block number (cb). The second field is the data of the specified type.

2.2.1.5. Data Layout

There are five data classifications shown in Figure 2.5. A complete description of each operation in the classes is covered in [1]. Class 1 can currently only have an *op* value of **reserve.** This indicates a block of memory starting at the current point with *value* integer number of bytes should be reserved.

Class 1:
(op value)
Class 2:
(op name)
Class 3:
(op $expr_1$ $expr_2$)
Class 4:
(op value name)
Class 5:
(op size name expr*)

Figure 2.5: Lcode Data Item Layout

Class 2 can only have *op* values of **void** and **global**. In both cases, *name* is a label that may be used for reference purposes only. It is illegal to load from or to store to either of these locations. **Global** is used to promote the scope of the label to become visible outside the file.

Class 3 can have *op* values of **wb**, **ww**, **wi**, **wf**, **wf2**, **ws** that are used to write a value to an address determined by the computation result of $expr_1$. The sizes of these fields are byte, word, integer, float, double and string pointer respectively. These are discussed in more detail in Section 2.2.1.4. The second expression ($expr_2$) must evaluate to the type defined by *op*. Figure 2.6 lists the valid expression types. The results of all expressions except **f** and **f2** will be integers. Therefore, the arithmetic expressions can only be applied to integers and address labels.

Class 4 can have *op* values of **ASCII** or **asciz** (a null terminated string) that will result in the allocation of the number of bytes necessary for the string *value*. The starting address of the allocated block will be assigned to *name*. This class also includes the memory alignment operation. The *value* indicates the alignment boundary. The aligned memory location is assigned to *name*.

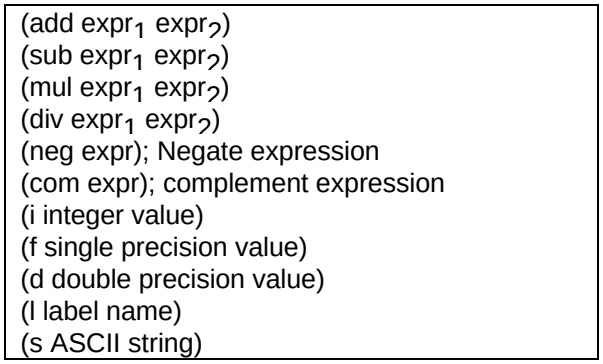

Figure 2.6: Lcode Expression Layout

Finally, class 5 may have *op* values of **byte**, **word**, **long**, **float**, or **double**. The *size* defines the number of units of type *op* to allocate. For example if *size* is ten and *op* is word (2 bytes), then twenty bytes will be allocated. The block of storage will be aligned to the appropriate boundary as defined by *op*. The aligned address will be assigned to *name*. Each location of the block, starting with the first at the aligned address, will be initialized by the one or more *expr**. It is illegal to have more expressions than locations to be initialized.

2.2.2. Internal Lcode and Mcode Representation

The goal in developing Mcode was to make it appear just like Lcode. Thus benefiting from a wealth of available tools. To this end, all data structures shown in subsequent sections refer to both Mcode and Lcode operations. There is one major difference between Lcode and Mcode due to the initial design of the Impact-I C compiler. All Lcode operations are stored in a large array. The Lcode unique identifier is specified by the location of the operation in the array. The organization of the Lcode instructions has been designed into numerous routines with the compiler. Thus, to take advantage of these routines, an equivalent array of Mcode operations

was created. For this reason, there are some duplicate routines covered in Chapter 5 where the only difference is indexing into the Lcode array instead of the Mcode array.

2.2.2.1. Function Data Structure

The function data structure shown in Figure 2.7 describes a function and its control blocks. The *name* field is the name of the original function. The *weight* describes the static or profiled execution count of the function. The *cb* array is a mapping of the function control numbers to the order encountered in the function. This mapping is necessary because several optimization techniques will cause creation of new control blocks. For debugging purposes, these new control block numbers are not renumbered. The field *n_cb* indicates the number of control blocks in the function.

```
typedef struct L_Func {
      char *name;
      double weight;
      int cb[L_MAX_CB];
      int n_cb;
      int s_local;
      int s_param;
      int s_swap;
      Set cb_set;
      Set op_set;
}
```

Figure 2.7: Lcode Function Data Structure

2.2.2.2. Control Block Data Structure

The control block data structure shown in Figure 2.8 describes a control block and it's Lcode operations. There is an implicit cb number associated with each **L_Cb** structure. This is calculated by taking the pointer to the control block and subtracting the value **L_Cb** that is an array containing all control blocks. The *weight* field is the profiled weight of the control block. The *weight2* field is the static weight of the control block. Profiled information can only be available if profiling has been performed. However, the static weight can be computed for the control block. First, loop detection is performed by calling **L_loop_detection(function_ptr)**.

Next, static weights are computed for the control block by calling **L_compute_static_cb_weight(cb_number)**.

```
typedef struct L_Cb {
      double weight;
      double weight2;
      L_Oper *first;
      L_Oper *last;
      struct L_Flow *src;
      struct L_Flow *dest;
      void *ext;
}
```

Figure 2.8: Lcode Control Block Data Structure

The fields *s_local*, *s_param* and *s_swap* provide run time information about local memory requirements, outgoing parameter register requirements and swap space requirements respectively. The fields *cb_set* and *op_set* are sets containing the unique identification numbers of all control blocks and Lcode operations. The sets are used to prevent duplication of control blocks or operations.

The next two fields, **first* and **last*, are pointers to the first and last Lcode operation within a control block. Control flow arcs into and out of the control block are given by **src* and **dest* respectively. Finally, an optional field, **ext*, is used for different purposes by different phases of code generation.

```
typedef struct L_Flow {
      int cc;
      short src_cb;
      short dest_cb;
      double weight;
      struct L_Flow *next;
}
```

Figure 2.9: Lcode Control Flow Data Structure

Control Flow arcs are described using the structure given in Figure 2.9. The field *cc* gives the branch condition (taken or not taken). The *src_cb* and *dest_cb* fields are used to show the source and destination control block number. If the flow structure is for a source arc, then *src_cb* will be the incoming control block number and *dest_cb* will be the control block that has defined the flow. If the flow structure is for a destination arc, then *dest_cb* will be the outgoing

control block number and *src_cb* will be the control block that has defined the flow. This provides a double linked list format that simplifies addition and deletion of nodes to control flow graph. The weight is the number of times that this control path was taken during profiling. Finally, **next* is a pointer to the next flow structure. This allows there to be multiple incoming and outgoing arcs for each control block.

2.2.2.3. Instruction Data Structure

This is an internal data structure Lcode operations are represented by the **L_Oper** data structure shown in Figure 2.10. There is a unique implicit Lcode operation number associated with each opcode that can be computed by subtracting the base address of the array **L_oper** from the pointer to the operation. It should be noted that this data structure is used both for Lcode and Mcode instructions. However, the Mcode instructions are kept in an array called **M_oper**. Therefore, to compute a Mcode operation number, substitute **M_oper** where **L_oper** was mentioned in the computation. The field **opcode* is a pointer operator name that was used in the Lcode file. The **opcode_attr* field is a pointer to an attribute structure described below.

```
typedef struct L_Oper {
      char *opcode;
      L_Opcode_Attr *opcode_attr;
      L_Operand dest;
      L_Operand src1;
      L_Operand src2;
      L_Operand src3;
      struct L_Oper *previous;
      struct L_Oper *next;
      L_Attr *attr;
      struct L_Oper *link;
      void *ext;
      double weight;
      int color;
}
```

Figure 2.10: Lcode Instruction Data Structure

There are four operand fields. Three are for source operands (***src1, src2, src3***) and one is for the destination operand (*dest*). The data structures for these fields are discussed below. For any given operation, the number of source and destination fields used is dependent on the operation.

There are two **L_Oper** pointers included in the structure. They point to the previous and next Lcode operations. If there is no previous or no next operation, the pointers will be set to null. The **attr* field is used to add processor specific annotations when annotating from Lcode to Mcode (discussed in Section 4). Since this data structure is used for both Lcode and Mcode, the **link* field is used as a pointer a parent Lcode operation. This supports debugging by allowing the intermixing of Lcode and Mcode operations in the output file. The **ext* field is used for compilation specific use.

The fields *weight* and *color* are used internal to modules such as register allocation. The static or profiled execution count of the operation is given by the field *weight.* This information is inserted after two steps. The first step was discussed in Section 2.2.2.2. After the control block weights are added, the operation weights can be added as follows. First, each control block must be colored. This is accomplished by calling *L_color_cb(cb_id)* to initialize the color field. Next, the operation weight is computed by calling *L_compute_oper_weight(operation_ptr, s, d).* The parameter ***s*** tells the function whether it should use static information to compute operation weight. The parameter *d* tells the function whether it should use available profiled information to compute operation weight. If both fields are set and the information is available, the profiled information will be used exclusively.

```
typedef struct L_Opcode_Attr {
    int id;
    char *name;
    int type;
    int ctype;
    int delay;
}
```

Figure 2.11: Lcode Opcode Attribute Data Structure

The opcode attribute field, shown in Figure 2.11, is used to describe each opcode. This information is kept in a global opcode array (**L_opcode**). The *id* field, unique for every Lcode opcode, is used as an index into the global array. The *name* field is a descriptive opcode name stored in the Lcode and Mcode files. The *type* field is used to categorize the opcodes as arithmetic, memory load, memory store, control or synchronization operations. The *ctype* field

```
typedef struct L_Operand {
      char type;
      char ctype;
      union {
            int cb;
            long i;
            float f;
            double f2;
            int mac;
            char *s;
            int r;
            char *l;
      } value;
      void *ext;
}
```

Figure 2.12: Lcode Operand Data Structure

shows resulting C data type for the operation. The *delay* shows the latency for the opcode. The contents of the attribute fields are filled during the initial start up process by the function **L_read_opcode_file** define in the file **L_opcode.c**. The configuration and latencies are defined in files of the form **L_OPCODE_***.

The L_Operand structure, shown in Figure 2.12, is used to describe the type and contents of each operand in an operation. The *type* field shows if the operand is a register, macro, label, string pointer, numeric value or unused. If the *type* field is a register or macro then the *ctype* shows the contents. Operand types of string and label are always given a *ctype* of integer since they are integer pointers.

The field *value* is a union that should be referenced by the appropriate entry associated with the *type* field. For example, if *type* is a macro, the reference should be to value.mac. Finally, **ext* is an optional field used by various phases of compilation.

2.2.2.4. Data Item Data Structure

The structure **L_Data**, shown in Figure 2.13 provides the necessary information about a Lcode data item. The field *type* defines what type of data element is defined (refer to Section 2.2.1.5 for class break down and types). Classes 1, 2 and 5 described in Section 2.2.1.5 were permitted to have *value* or *size* fields that are always integer numbers. The information for these fields is stored in the field *N*.

```
typedef struct L_Data {
      short type;
      int N;
      L_Expr *address;
      L_Expr *value;
      void *ext;
}
```

Figure 2.13: Lcode Data Item Data Structure

Classes 2 through 5 can have expressions or names. A name is an address calculation to a memory location. Name expressions are pointed to by the field **address*. Class 3 and 5 may have an additional expression that can be some arbitrary calculation. This calculation expression is pointed to by **value*.

The expression data structure, **L_Expr**, is shown in Figure 2.14. The field *type* is a uniquely identifies the valid fields in the expression data structure. If the type is a calculation, it is based on a series of calculations pointed to by the expressions **A*, **B* and **next*. Additionally, the type may represent a series of sequential initializations. Refer to Section 2.2.1.5 for more information on expressions.

```
typedef struct L_Expr {
      short type;
      union {
            long i;
            float f;
            double f2;
            char *l;
            char *s;
      } value;
      struct L_Expr *A, *B;
      struct L_Expr *next;
}
```

Figure 2.14: Lcode Expression Data Structure

2.3. Differences between Lcode and Mcode

Mcode is based on Lcode that was discussed in Section 2.2. Extensions have been added to Lcode to provide more distinctions between machine architectures uniformly. This facilitates the development of generic modules that perform Mcode optimizations, register allocation and instruction scheduling as discussed in Section 2.1.

Four standard extensions have been defined to make Mcode more generic. These extensions are supported by all aspects of the optimizer, code scheduler and register allocator. These extensions are key to providing a machine specific versions of Lcode while still maintaining a common language.

The extensions supply information to support *general delay slots* as may be encountered with a load, multiply or coprocessor call. Supplied information defines if the delay slot can be handled with hardware interlock and the number of delay slots that must be filled.

Branch delay slots are treated similarly to general delay slots but information is provided to indicate if there exists a squashing branch that may be used in place of a standard branch instruction. The squashing branch may be a squash on taken or squash on not taken.

It may be necessary or more efficient to use *macro registers* that are not handled by register allocator. These registers might include a *zero register* as supported by the SPARC, a *status register* as supported by both the Intel i860 and the SPARC or might be a register to control the variable length overlapping register window of the AMD-29000.

To reduce redundant computations during code generation, specific *define* opcodes [1] are used to propagate information from one phase of code generation to another. In the AMD-29000 code generator, these define instructions provide information on the number and type of registers used, breakdown on memory usage for a function (i.e., swap and spill space or alloc space). This allows a function to pass the information needed to control the variable length register window without having to recompute it in multiple of the phases.

Finally, machine specific annotations or extensions can be used to show a divergence from the standard use of an opcode. For example, the AMD-29000 instruction set can only support immediate values from 0 to 255. If it is necessary to add a negative number in the range -1 to -255, additional code would need to be inserted to move the negative number into a register first (requiring 1 to 2 instructions) before the instruction. If an extension was used, this instruction could be converted to a subtract where the first source operand could be subtracted from the absolute value of the second source operand and produce the same result. This would require an

extension field for the subtract opcode to tell the print time routine to use a subtract reverse instruction. Other types of examples will be discussed during the subsequent annotation sections.

3. GETTING STARTED

3.1. Usage

The convention for naming code generators is to place a capital **L** in front of the processor type. Thus, the AMD-29000 code generator is executed using the following command line.

Lamd29k [-target amd-29k] [-c #] [-i in_file] [-o out_file] [-verbose]

The option **target** specifies the processor type. This permits support of multiple similar processors by one code generator as well as providing necessary information to library routines as to the type of supported processor.

The option **c** specifies the code generator phase to be executed. If this parameter is missing or a 0 is provided, then all phases are executed. This requires that the input file be an Lcode file and the output file be the assembly file. If the option is a 1, then only code annotation will be performed. If the option is a 2, then only optimization, register allocation, post-pass code annotation and instruction scheduling will be performed. If the option is a 3, then only generation of assembly code will be performed. An option of 4 is also available that will perform the same option 1 followed by option 2. Typically options 1 through 4 are only used during the development process.

The options **i** and **o** specify the input and output file names. By convention, an input file has the extension *.lc* for Lcode and *.mc* for Mcode. An output file should have the extension *.s* for an assembly file, *.mco* for an optimized Mcode file (one that has gone through phase II). If no input or output files are specified, standard input and standard output are assumed respectively.

The option **verbose** tells the code generator to produce all support information to the screen and file.

3.2. Directory Structure

The development organization of a code generator is shown in Figure 3.1. In this example the Lamd29k directory is the code generator directory. The main directory contains the executable, a README file describing the usage of the code generator, and the main entry point for a code generator. The file *l_codegen_main.c* handles the **c** option described in Section 3.1, opens necessary input files and calls the initialization modules based on the user specified phases. The include file *l_codegen_main.h* contains the necessary systems libraries for I/O and string manipulation, as well as all include files necessary for a code generator.

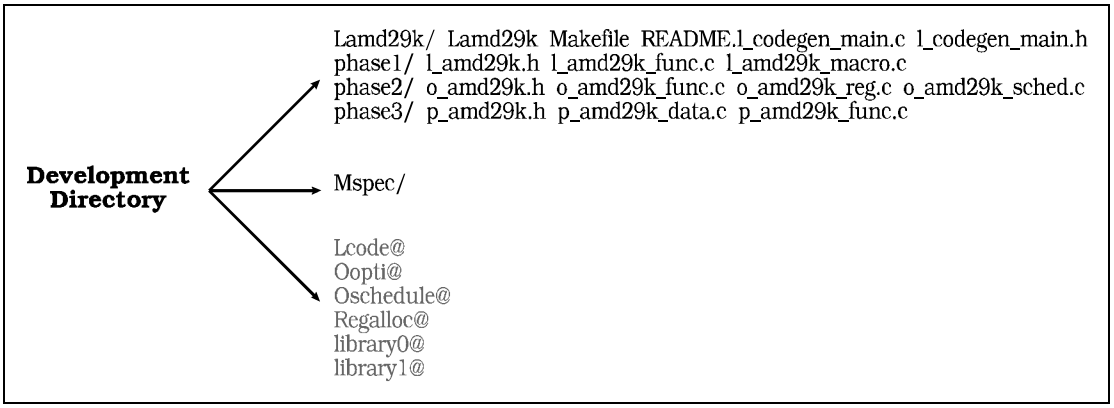

Figure 3.1: Directory Organization for an IMPACT-I Code Generator

The sub directories *phase1/, phase2/ and phase3/* correspond to their respective phases discussed in Chapter 2. The file *l_amd29k.h* is the main include file for phase I. The file *l_amd29k_func.c* handles all code annotation required for a function. The file *l_amd29k_macro.h* provides the necessary interfaces required for custom macro registers. These files will be covered in more detail in Chapter 5.

The file *o_amd29k.h* is the main include file for phase II. The file *o_amd29k_func.c* is the main interface point for functions in phase II. It ensures the appropriate calling order for all parts of this phase. It also contains all routines to support post-pass code annotation and peep-hole optimization. The file *o_amd29k_reg.c* is the interface module for the generic Mcode

register allocator. The file *o_amd29k_sched.c* is the interface module for the generic Mcode instruction scheduler.

The file *p_amd29k.h* is the main include file for phase III. The file *p_amd29k_func.c* converts all Mcode functions to their appropriate assembly code. The file *p_amd29k_data.c* converts all Mcode data items to their appropriate assembly code.

The half-toned names correspond to UNIX links to shared directories. The directory *Lcode* contains all Lcode functions that are discussed in Appendix A. The directory *Oopti* contains the code that performs the optimizations. The directory *Oschedule* contains the code that performs instruction scheduling. The directory *Regalloc* contains the code that performs register allocation. The directories *library0* and *library1* contain system library routines for the IMPACT-I compiler.

A UNIX shell script, *setup_codegen,* will create this organization automatically. The only input parameter is the main directory name. It is suggested that a sub-directory for the code generator be created since there are a number of critical directory links created. After creating the directory hierarchy, the shell script will copy the latest versions of the template files into the appropriate sub directories.

4. MACHINE SPECIFIC REQUIREMENTS

4.1. Overview and Organization

In order for the front end of the compiler to efficiently produce Lcode, we must inform it of specific characteristics of the target machine. This information includes the number of incoming and outgoing parameters allowed in registers, method of referencing variables on the stack, structure alignment, and parameter sizes. All of this information is conveyed to the compiler through the *mspec* or machine specification for the respective processor. These files are stored in the directory **Mspec** shown in Figure 3.1. Unlike other sections of the code generator, it is best to start with the machine specification file for a processor that closely matches the target processor. There can be significant differences in this interface based upon whether the processor stores data in a BIG-ENDIAN or LITTLE-ENDIAN manner.

Once the Mspec file is created, it must be compiled in with the module **Hcode** before it can be tested. It is often useful in the early stages to copy the complete directory structure for this module. After satisfactory results are achieved, the specification file can be released.

4.2. Interface Requirements

4.2.1. Initial Changes

The first step is the assignment of a target processor name of the form **amd-29k** or **dec-mips**. This is used by any calling module for the machine specification file and by the code generator to distinguish between different models. At the same time, a name is assigned that will be used as part of all function names in the file. This name is typically of the form

amd29k. All references to names in subsequent sections assume a starting point of the AMD-29000 specification file.

Once assigned a descriptive name for the target processor, make the following set of global changes to the mspec. Change all occurrences of **AMD29K** in the *define* statements to the appropriate processor name. Next, change all occurrences of **amd29k** in function names to the processor name. Remember that processor name used in the function names must be identical to the one assigned.

4.2.2. Field Size and Alignment

At the beginning of the mspec file are a group of size and alignment definitions. The size definitions are used to describe all parameter sizes in bits. If a parameter size is variable, then assign it -1. For example, a double precision field would be assigned a size of 64-bits.

The alignment fields are used to describe how to align each of the fields to the nearest bit boundary. If the alignment is variable such as with structures, then a minus one is used.

4.2.3. Macro Descriptions

Currently, a processor may have up to 16 incoming and 16 outgoing parameters. The first incoming register is assigned a base offset of 0 and the first outgoing parameter register is assigned a base offset equal to the maximum number of incoming parameter registers. These are defined using **M_AMD29K_IN_BASE** and **M_AMD29K_OUT_BASE**. The maximum number if incoming parameters must be defined using **M_AMD29K_MAX_FNVAR_REG**. This is used later in the specification file to determine if parameters must be passed through memory or register.

In some calling conventions, parameters are always stored on the stack such as the return address. This storage can be defined using **M_AMD29K_PARAM_FP_OFFSET**. There is no way to determine if a function is a leaf or non-leaf function when these routines are called. Therefore, variable information should be computed within the code generator.

4.2.4. Array Support

The compiler determines the initial alignment of an array using the function **M_amd29k_array_align** It determines the initial offset using the function **M_amd29k_array_layout**. The initial array offset is assumed to be aligned and to the first physical entry in the array. Finally, it determines the array size using **M_amd29k_array_size**. These routines are generic and take advantage of the size and alignment information provided in Section 4.2.2.

4.2.5. Union Support

The compiler determines the initial alignment of a union using the function **M_amd29k_union_align**. It determines the initial offsets using the function **M_amd29k_union_layout**. The initial union offsets are assumed to be aligned. Finally, it determines the union size using the function **M_amd29k_union_size**. These routines are generic and take advantage of the size and alignment information provided in Section 4.2.2.

4.2.6. Structure Support

The compiler determines the initial alignment of a structure using the function **M_amd29k_struct_align**. It determines the initial offsets to fields using the function **M_amd29k_struct_layout**. This routine lays out the fields attempting to minimized memory requirements. Finally, it determines the structure size using **M_amd29k_struct_size**. It is important to have a good understanding of structure layout and alignment for the processor before attempting modifications to these routines.

4.2.7. Function Variable Support

The compiler determines function parameter information using the function **M_amd29k_fnvar_layout**. This routine is responsible for defining if parameters are referenced through the stack pointer or the frame pointer as well as the number and initial offsets for the incoming and outgoing parameters.

The first part of the function determines how each parameter is to be passed. If a parameter will fit in a register, then it is marked to be passed through register and the number of used parameter registers are incremented. If it can only be passed through memory, it is marked in this manner. If the maximum number of parameter registers have been used, the remaining parameters are passed through memory. Minor modifications will permit passing parameters through either integer or floating point registers by assigning the appropriate type of parameter register at this stage..

After defining the passing convention for each of the parameters, memory is allocated on the stack with appropriate alignment to permit stacking of the parameters.

4.2.8. Local Variable Support

The compiler determines the offset for local variables using the function **M_amd29k_lvar_layout**. This function determines if local variables are to be referenced from the stack or the frame pointer as well as defining the size and alignment of each parameter. It most cases, the local variables are aligned to the maximum size boundary to ensure that double precision variables are correctly aligned.

4.2.9. Miscellaneous

There are several routines that convey important information to the compiler. First, the function **M_amd29k_no_short_int** tells the compiler if it should interpret short integers that are typically 16-bit as long integers that are 32-bit. The compiler can determine if the processor is LITTLE-ENDIAN or BIG-ENDIAN with the function **M_amd29k_layout_order**. Finally, it can determine macro register numbers used to reference return structures using **M_amd29k_structure_pointer** and return parameters using **M_amd29k_return_register**.

5. PHASE I -- CODE ANNOTATION

5.1. Overview and Organization

Code annotation refers to the process of converting one instruction format to another. The amount of effort expended in this phase is directly proportional to the similarity betwee the target instruction set architecture and Lcode. Typically, there will be group of instructions that map directly to the target processor and require no annotation, a group that may require a small amount of annotation and a group that may be unsupported directly and require significant annotation. Before beginning this step, understand clearly how each instruction works and the types of operands it may support.

A template for this phase exists to ease the development process. The template is currently in the form of the AMD-29000 Lcode annotator demonstrating the functionality of this phase of code generator on a working machine. There are a set of functions within the annotator that demonstrate all of the useful functions which can be used to ease code development.

The template is composed of three files which meet all of the internal and external interface requirements for phase I. The file **l_amd29k_func.c** contains all annotation routines. The file **l_amd29k.h** contains all variable and function definitions. The file **l_amd29k_macro.c** permits definition of processor specific macro registers that can be legally parsed and printed by the standard Lcode libraries.

5.1.1. Generic Features of l_amd29k_func.c

The function **L_process_func** is called once for every function within the original Lcode file. It prints the function header, control block headers and ensures that the Lcode operations are emitted prior to their annotated Mcode operations in debugging mode. It calls

L_annotate_oper once per Lcode operations. This routine is a generic routine and should not be modified.

The function **L_annotate_oper** is the main parsing routine for Lcode operations. It is composed of a large switch statement containing all supported Lcode operations. Typically operations which require minimal or no annotation will be converted to Mcode operations in the switch statement. Operations which can expand to multiple Mcode operations should be annotated in separate subroutines.

5.1.2. Generic Features of l_amd29k_macro.c

A macro is a descriptive name for a register or a variable which can be embedded within Mcode operations. The following two functions provide an interface which permits definition of macros.

The function **M_init_macros** will add any machine specific macros to permit accurate parsing of the input data stream. Examples of machine specific macros might be the *zero* register contained in the SPARC. This routine must exist to prevent link time errors but may be empty if no custom macro registers are required. This function is called from within **l_code.c**.

The function **M_get_macro_name** is required to print the string equivalent of any processor specific macros defined in **M_init_macros** based on the specified unique identifier. This routine must exist to permit accurate linking but should return a "?" to indicate any unsupported macros. This function is called from within **l_code.c**.

5.2. Initialization

The function **L_init** is a generic initialization routine is provide as a standard part of the interface to permit start-up initialization for code annotation. This routine will be called from **l_codegen_main.c** prior to calling **L_process_func**. Any other forms of initialization should be handled as required during the annotation process.

5.3. Methodology

There are a number of general ground rules which should be adhered to in order to create a code generator using the Mcode template. This will be discussed in detail within this Section.

The most important rule is to maintain complete compatibility with the existing Lcode operations. This means that all source operands defined in [1] must be present. Any missing source operands may eliminate or create erroneous dependencies which can produce disastrous results during phase II. There is an extension field which is appropriate for indicating divergence from standard Lcode commands. Any unneeded fields can be ignored by the code generator during later phases.

There are a group of library routines in Appendix A which should be used by the code generator. These routines will ease the development process, ensure correct initialization of critical fields and allow future extensions to Lcode without causing turmoil in all code generators. Using these routines also prevents taking advantage of internal data structures for short cuts that may later be changed to improve performance or reduce memory requirements.

Some of the Lcode instructions support up to three 32-bit source fields in the form of integers, labels or string pointers. They may also support up to two single or double precision floating point numbers in source fields. Most of the RISC processors have a fixed instruction size of 32-bits making it impossible to support even one of these fields directly. In addition, few CISC processors can support all of these fields at once. Therefore, annotating apparently harmless Lcode commands may require generation of instructions for one or more temporary registers. Do not arbitrarily generate extra instructions for these immediate fields. Some of them may fit within certain instructions!

Another problem related to the limited instruction size is that most architectures use multiple instructions to load the upper and lower halves of registers. The AMD-29000 can require up to four instructions to load a double precision immediate. This type of annotation should not occur during the phase I but after register allocation covered in Section 6.5. This simplifies register allocation and allows the reuse of the Lcode optimizations routines.

Some architecture's require that immediate fields always be in a specific source parameter. This may require swapping of source operands. Remember that subtraction and division are not commutative operations. Some processors provide instructions to handle take care of the commutativity.

Always attempt to convert an unsupported Lcode instruction into a series of simpler instructions that are supportable. If this is not possible, leave the instruction unsupported and generate an error to flag during phase I. If the instruction is encountered frequently, work with other members of the IMPACT group to resolve the problem.

5.4 Creating an Mcode Instruction

It is important to know the exact series of steps required to create an Mcode operation. First, it is mandatory to call either **L_create_moper** or **L_duplicate_moper** to create an Mcode operation. **L_create_moper** creates Mcode operation with the opcode specified in the parameter. A pointer to a parent Lcode command is provided for debugging purposes. All fields other than the opcode and link field are initialized to undefined or zero. This means that all of the required parameters must be specified. If a desired Mcode command is identical or very similar to an existing Lcode or Mcode operations, make an exact duplicate of in by calling **L_duplicate_moper**. In this case, it is only necessary to provide a pointer to the operation to be duplicated. The parent Lcode operation is set to the parent of the duplicated operation or the pointer to the Lcode command. This is easy to determine since an Lcode command will always have its link field set to null.

Once the operation is created, modify any necessary operand fields. Library routines allow creation of support fields from scratch. It is also possible to duplicate a field by calling **L_duplicate_operand**.

After the operands have been created, specify any necessary extensions using **L_set_extension**. Remember that the usage of these extensions is up to dependent on the target processor! It is also necessary to specify any delay slot information using **L_set_fill_specs**.

Finally, insert the new operation into the control block. This can be accomplished in many ways, however, the most efficient way is to use **L_insert_before**. This function will insert the operation immediately before the original Lcode operation. The higher level routine will take care of deleting the Lcode operation from the control block and eliminate any apparent confusion. As a result of using **L_insert_before**, no pointers need to propagated through the function and will always be inserting the instructions in the order created.

5.5. Annotation Examples

This Section will show examples of frequently performed annotations and the appropriate code that is used to perform these annotations. In all of the examples, a new Mcode operation is always inserted before the existing Lcode operation. After the instruction has been completely annotated, the Lcode operation will be removed from the control block and the only reference will be a link which was initialized when the new Mcode operation was created. For further details see Section 5.3.

5.5.1. Passing Information Between Phases

It is possible to pass information from one phase of code generation to another by create and Lcode *DEFINE* operation. The AMD code generator uses this as a means of reducing computation overhead. Phase I creates a set of define operations which are used to communicate the number of incoming and outgoing parameters, the swap space and alloc space requirements as well as the number of callee and caller save registers. Several of these are filled in during phase II after register allocation. This information is used to accurately create the prologue and epilogue sections for a function and to provide diagnostic information about the function when the **verbose** option is selected. It is useful and usually necessary to define custom macros to accomplish this task.

```
new_oper = L_create_moper(Lop_DEFINE, oper);
L_new_macro_operand(&new_oper->dest, L_MAC_INPARAM, L_CTYPE_INT);
L_new_int_operand(&new_oper->src1, incoming_regs);
L_insert_before(mcb, oper, new_oper);
```

Figure 5.1: Defining a Field

Figure 5.1 shows the appropriate code to define the incoming parameter registers for a function. An Lcode function does not provide this information and therefore it is necessary to scan the function for this information. Once this information has been determined, it can be passed between phases using the code from this example. The functions are discussed in detail in appendix A. The variable *new_oper* is a pointer to the new Mcode command, *oper* is the pointer to the original Lcode command, *incoming_regs* contains the number of incoming parameters and *mcb* is the control block that the new operation is being inserted into. The *new_oper* is inserted before the existing Lcode *oper*. The call to **L_new_moper** includes the pointer to the Lcode operation. This pointer will be stored in the *link* field which was discussed in Section 2.2.2.3.

5.5.2. Directly Supported Lcode Operations

If the processor can support an Lcode command without any changes, the code example given in Figure 5.2 should be used. This sample creates a new Mcode operation by duplicating the existing Lcode operation and inserting it into the control block.

```
L_insert_before ( mcb, oper, L_duplicate_moper(oper));
```

Figure 5.2: Duplicating an Operation

5.5.3. Unsupported Lcode Operations

In the event that an Lcode operation can not be supported, it must be converted to an equivalent set of supportable Lcode operations. As an example, the AMD-29000 can not support the conditional branches as they are defined by [1]. This requires that all conditional branches be converted to a binary case using a compare and then a branch.

```
new_oper = L_create_moper(Lop_GT, oper);
breg = L_n_reg++;
L_new_register_operand(&new_oper->dest, breg, L_CTYPE_INT);
L_duplicate_operand(&new_oper->src1, &oper->src1);
L_duplicate_operand(&new_oper->src2, &oper->src2);
L_insert_before(mcb, oper, new_oper);

new_oper = L_create_moper(Lop_BEQ, oper);
L_new_register_operand(&new_oper->src1, breg, L_CTYPE_INT);
L_new_int_operand(&new_oper->src2, 0);
L_duplicate_operand(&new_oper->src3, &oper->src3);
L_insert_before(mcb, oper, new_oper);
```

Figure 5.3: Unsupported Lcode Operations

Figure 5.3 shows how to annotate a *branch greater than* instruction into a series of instructions that can be supported by the AMD-29000. The first group of 6 lines create a compare instruction with a temporary destination register. The second group of 5 lines create a *branch equal* operation based on the condition of the compare and using the original destination.

5.5.4. Unsupported Field Sizes

Since different instructions have differing field requirements, a generic set of routines were developed to split fields into separate conditions. The function **L_register_only** may be called if only a register or macro register may be used in a field. This routine will create an Lcode operation with the appropriate move operation to a temporary register and insert the new operation into the control block. If no operation is created, a null pointer is returned, otherwise the pointer to the new operation is returned. This permits the calling routine to determine the temporary destination which must be inserted into the annotated Lcode operation.

Figure 5.4 is a modification of Figure 5.3 taking into account potential unsupported field sizes. The first two lines ensure that the appropriate field sizes are used for the source operands. A simple check for a returned null pointer determines whether to use the original source register or the new temporary register.

```
operand_oper1 = L_register_only(&oper->src1, oper);
operand_oper2 = L_register_or_const(&oper->src2, oper);

new_oper = L_create_moper(Lop_GT, oper);
breg = L_n_reg++;
L_new_register_operand(&new_oper->dest, breg, L_CTYPE_INT);
if (operand_oper1)
     L_duplicate_operand(&new_oper->src1, &operand_oper1->dest);
else
     L_duplicate_operand(&new_oper->src1, &oper->src1);
if (operand_oper2)
     L_duplicate_operand(&new_oper->src2, &operand_oper2->dest);
else
     L_duplicate_operand(&new_oper->src2, &oper->src2);
L_insert_before(mcb, oper, new_oper);

new_oper = L_create_moper(Lop_BEQ, oper);
L_new_register_operand(&new_oper->src1, breg, L_CTYPE_INT);
L_new_int_operand(&new_oper->src2, 0);
L_duplicate_operand(&new_oper->src3, &oper->src3);
L_insert_before(mcb, oper, new_oper);
```

Figure 5.4: Unsupported Field Sizes

5.5.5. Instructions with Delay Slots

There are two types of delay slots which should be marked during the first phase of code generation. The first type are any instructions which may change the flow of control away from the next sequential instruction. These include conditional and unconditional branches, subroutine calls and return from subroutine calls. In most architecture's, these instructions cause stalls in the instruction pipeline when the next address is not available earlier enough. In order to eliminate these stalls, delay slots are used. Thus, the instruction scheduler can put an instruction into the slot to prevent degradation in performance.

The second type of delay can be from instructions such as loads. Due to memory latencies, the destination of the load should not be accessed in the next instruction. These types of delay slots may be resolved inside the processor using hardware interlock if the results are not available in time thus preventing an instruction from executing and producing an incorrect result. To simplify the internal hardware of the processor, interlock may not be provided thus requiring the compiler to fill the delay slots with useful instructions or nop instructions.

Commonly, hardware interlock is provided for general delay slots (an exception to this is the load instruction in the MIPS R2000/R3000). Most pipelined processors require static filling of branch delay slots.

In order to statically fill a branch delay slot, an instruction must be found which does not alter the operation of the program. Quite commonly, an instruction is moved from above or below the branch into the delay slot. In some circumstances it is possible to move an instruction from the destination of the branch into the delay slot. To accomplish this, one of three conditions must be met. Either the instruction must be common to all paths from the branch, the instruction must not alter a register that is used in one path or the other (not in the live-out of the branch), or it must be possible to prevent writing the results of the instruction if an incorrect decision is made. The last case is called a squashing branch. This means that if the compiler incorrectly makes a guess as to the direction of the branch, the instruction can be aborted causing no harm. Thus the compiler can insert the instruction from the destination of the most likely path into the delay slot.

In order for the instruction scheduling routines of phase II to efficiently and correctly handle these types of instructions, all instructions with delays must be marked. A **nop** instruction will be inserted if the operation has not hardware interlock and no useful instruction can be filled into its delay slot. The routine **L_set_fill_specs** is used to mark these instructions appropriately. This routine must be called for each of these instructions. Details for this function are covered in appendix A.

```
new_oper = L_create_moper(Lop_BEQ, oper);
L_new_register_operand(&new_oper->src1, breg, L_CTYPE_INT);
L_new_int_operand(&new_oper->src2, 0);
L_duplicate_operand(&new_oper->src3, &oper->src3);
L_set_fill_specs(new_oper, L_NO_STATUS, 1);
L_insert_before(mcb, oper, new_oper);
```

Figure 5.5: Marking Delay Slots

Continuing with the example from Figure 5.5, we finally finish by providing status information about the conditional branch and the number of delay slots. This example shows

that this conditional branch does not support squashing branches and has no hardware interlock. It also indicates that there is one branch delay slot which should be filled.

5.5.6. More Efficient Operations

There are cases where an architecture may provide some more efficient ways of handling Lcode operations than the straight forward annotations would produce. The original code segment shown in Figure 5.7 has one problem for the AMD-29000. An immediate field must be the second source field. A the straight-forward annotation would produce two instructions to address this problem. However, a more efficient coding can take advantage of the subtract reverse instruction provide by AMD. The new code subtracts r5 from 1 and accomplishes the same solution in a more efficient manner.

```
Original Code: sub     r6,1,r5

Straight-forward: mov  r4,1
                  add  r6,r4,r5

Efficient:        subr r6,r5,1
```

Figure 5.6: Efficient Annotation

5.5.7. Extensions

The efficient annotation shown in Figure 5.6 is good except Lcode does not support a subtract reverse. The first impulse is to extend Lcode. There can always be good reasons to add useful commands. Unfortunately, it is not feasible to add a new set of commands every time a new code generator is developed. Therefore, an extension field is used to indicate any deviations from a standard Lcode operation. The meaning and use of this field is left up to each code generator. The AMD-29000 code generator uses it to mark a deviation from a standard command.

Figure 5.7 has converted the subtract instruction mentioned to a subtract reverse by setting the extension field. This extension is only recognized by phase III in the AMD-29000 code generator.

```
new_oper = L_create_moper(Lop_SUB, oper);
L_duplicate_operand(&new_oper->dest, &oper->dest);
L_duplicate_operand(&new_oper->src1, &oper->src1);
L_new_int_operand(&tmp_oper, -src2->value.i);
L_duplicate_operand(&new_oper->src2, tmp_oper);
L_set_extension(new_oper, SUBR);
L_insert_before(mcb, oper, new_oper);
```

Figure 5.7: Use of Extensions

6. PHASE II -- OPTIMIZATION, REGISTER ALLOCATION AND INSTRUCTION SCHEDULING

6.1. Overview and Organization

Phase II combines several features of the code generator into one module with the goal of faster overall performance. It provides a generic optimizer, register allocation, pre-pass and post-pass code scheduling and post-pass code annotation.

A template for this phase exists to ease the development process. It is composed of four files that meet all of the internal and external requirements for phase II. The file **o_amd29k_func.c** is the main entry point that controls which aspects of phase II are executed. Each aspect of this phase can be enabled or disabled independently with compile time switches. The main entry point is **O_process_func** that is called once per function. It then calls each part of this phase sequentially for each function. This function is generic and should not be modified.

The file **o_amd29k.h** contains all variable and function definitions. The file **o_amd29k_reg.c** is the interface to the generic register allocator and will be discussed in further detail in Section 6.4. The file **o_amd29k_sched.c** is the interface to the generic instruction scheduler and will be discussed in greater detail in Section 6.6.

6.2. Initialization

The function **O_init** is a generic initialization routine provided as a standard part of the interface to permit one-time initialization for phase II. This routine is called from **l_codegen_main.c** prior to calling **O_process_func**.

A second initialization routine, **O_perform_init**, may be called once per function. This provides a simple means of performing initialization that may be required for the register

allocator of some architectures. Specifically, since the AMD-29000 has a variable length register window with parameter registers stored in the window, we must tell the register allocator about different numbers of available registers from one function to the next.

6.3. Optimization

6.3.1. Purpose

The code annotation process of phase I may generate redundant code within control blocks as a result of incompatibilities between Lcode and the target instruction set. Common sub expression elimination, copy propagation and dead code elimination have been supported to remove these redundancies. These optimizations will only be performed on a control block basis. It is assumed that the more significant global optimizations were performed by earlier compilation phases such as **Lopti** and **Ltrace**.

6.3.2. Interface Requirements

The optimizations may be enabled or disabled by the compile time switch **MACHINE_OPT** located in **o_amd29k_func.c**. If the switch is enabled, **O_perform_optimizations** will be called with a pointer to the current function.

6.4. Register Allocation

6.4.1. Purpose

The register allocator was designed to support traditional register architectures as well as those with fixed and variable length register windows. The register files may be separate or overlapped for integer, single-precision or double precision registers. The number of available registers may be defined once per function or once per file allowing changes as needed. The register allocator will no only perform efficient allocation but will also insert necessary spill and fill code.

6.4.2. Interface Requirements

6.4.2.1 Register Banks

The physical register configuration of a target processor is divided into a series of register banks. A register bank is uniquely defined by the register saving convention and the data type. The register saving convention can be either **caller save** implying that critical registers are saved by the calling function or **callee save** implying that critical registers are saved by the called function.

There are four additional parameters that will fully describe the register bank: the number of registers, the size of the register, an offset into the register map and a target processor register map. The size of the register defines integer registers or single/double/quad floating point registers. The register map is an array of physical register numbers which will be assigned upon exit from the register allocator. The offset permits sharing of the same array of physical registers between multiple banks or can be used to create the variable register window requirements.

```
R_define_physical_bank (R_CALLER, R_INT, 0, CALLER_INT_REG, 1,
                        caller_reg_map);
R_define_physical_bank (R_CALLER, R_FLOAT, 0, CALLER_FLOAT_REG, 1,
                        caller_reg_map);
R_define_physical_bank (R_CALLER_, R_DOUBLE, 0,CALLER_DOUBLE_REG, 2,
                        caller_reg_map);
```

Figure 6.1: Creating Caller Save Register Banks

Figure 6.1 creates a bank of caller save registers for integer, floating point and double precision floating point. Each bank begins at offset zero and uses the same register map. Thus, this example has created three overlapping caller save register banks as required for the AMD-29000. A separate register map can be assigned to each bank, ensuring non-overlapping register banks.

Figure 6.2 creates a bank of callee save registers for integer, floating point and double precision floating point. This example shows the variable length register window capability as well as overlapping register banks. The total number of available registers is modified once per

function call by subtracting the number of incoming and outgoing parameter registers. Then the starting offset within the register map is adjusted for the number of outgoing parameter registers.

```
R_define_physical_bank (R_CALLEE, R_INT, CALLER_INT_REG,
                        CALLEE_INT_REG-incoming-outgoing, 1,
                        &callee_reg_map[outgoing]);
R_define_physical_bank (R_CALLEE, R_FLOAT, CALLER_FLOAT_REG,
                        CALLEE_FLOAT_REG- incoming-outgoing, 1,
                        &callee_reg_map[outgoing]);
R_define_physical_bank (R_CALLEE, R_DOUBLE, CALLER_DOUBLE_REG,
                        CALLEE_DOUBLE_REG-(incoming+outgoing)/2,
                        2, &callee_reg_map[outgoing]);
```

Figure 6.2: Creating Callee Save Register Banks

An optional bank of registers may be defined that will be used to load and store virtual registers that can not be allocated to physical registers. These are defined in the same manner as shown in the above examples with the class substitution of R_SPILL instead of R_CALLEE or R_CALLER. If bank of spill registers is not defined, some will be allocated from the caller save bank if required.

6.4.2.2. Costs Associated with Register Banks

The code generator provides information about the cost of utilizing a callee or caller save register to make efficient use of the banks. The **caller cost** is the number of instructions or cycles required to save and subsequently restore a register around a function call. The **callee cost** is the number of instructions or cycles to save and subsequently restore a register at the entrance and exit to a function. Most architectures have a fixed cost to save and restore a register. However, windowing architectures have different costs. The **callee cost** is the penalty assessed to adjust the register window regardless of the number of registers used. There is a **subsequent callee cost** that is used to compute the overhead for each subsequent callee register. These costs may also be computed differently if the function is a leaf or non-leaf function. The functions **O_caller_cost** and **O_callee_cost** are provided as standard interfaces for computation of these costs.

6.4.2.3. Register Spilling and Filling

Unfortunately, due to limitations in register files or for efficiency reasons based on callee and caller costs, it is necessary to spill registers to memory and fill registers from memory. As a result, there are three standard routines provided by each code generator to facilitate this process. The function **O_fill_reg** will load a register of the specified type from the specified offset into the specified register. The function **O_spill_reg** does the opposite of **O_fill_reg**. The function **O_move_reg** is used to copy the contents of one register to another. This may be necessary if the register type required is not available.

6.5. Post-pass Code Annotation

6.5.1. Purpose

Post-pass code annotation is designed for two primary purposes. First, it permits annotation of Lcode commands that can not be completed until after register allocation. Specifically, it is difficult to insert prologue and epilogue code without knowing the register requirements. Second, it allows the annotation of instructions that were inserted in phase I but may need to be annotated to more than one instruction.

6.5.2. Interface Requirements

The function **O_annotate** is called once per function after register allocation. Execution of this function is enabled or disabled by the compile time switch **CODE_ANNOTATE**. Annotations performed here are done exactly as described in Chapter 5. Again, it is important that all rules mentioned in Chapter 5 be maintained during these annotations. For example, the system has no knowledge of upper or lower halves of registers.

If the processor can manipulate upper and lower halves of registers, there are two goals of annotation after register allocation. First, it is necessary to tell phase III to create more than one instruction and what the instruction should look like. Second, create an equivalent number of instructions as required so that post-pass code scheduling has more candidates for fill slots.

Once these instructions are created, it is necessary to ensure an appropriate dependence between them so that they are not reordered and thus produce incorrect results. Figure 6.3 shows an example with Mcode that accomplishes this goal. The appropriate annotation code will be required to create the multiple operations. In this example, the *mov_f2* operations is converted to an initial move and three subsequent add instructions, creating the appropriate number of instructions. The destination of each previous instruction is the source of the subsequent instruction in the group, thus ensuring the correct dependence. Finally, an extension is added to each of the operations to denote the loading of the lower or upper half of the lower or upper register, thus allowing phase III to produce the correct code.

```
Before:
(Mop 39 mov_f2 ((r 27 f2)) ((f2 3.14150000000e+00) () ()) (()) ())

After:
(Mop 39 mov_f2 ((r 27 f2)) ((f2 3.14150000000e+00) () ()) (()) ()
   (ext DBL_LREG_CONST))
(Mop 40 add_f2 ((r 27 f2)) ((f2 3.14150000000e+00) (r 27 f2) ())
   (()) () (ext DBL_LREG_CONSTH))
(Mop 41 mov_f2 ((r 27 f2)) ((f2 3.14150000000e+00) (r 27 f2) ())
   (()) () (ext DBL_HREG_CONST))
(Mop 42 mov_f2 ((r 27 f2)) ((f2 3.14150000000e+00) (r 27 f2) ())
   (()) () (ext DBL_HREG_CONSTH))
```

Figure 6.3: Annotation of Double Precision Move

6.6. Instruction Scheduling

6.6.1. Purpose

There are four aspects of the instruction scheduling. These include pre-pass and post-pass code scheduling, filling branch slots from destination and filling in no-ops where hardware interlock is not provided.

Pre-pass code scheduling will move instructions closer to likely fill slots thereby increasing the register live ranges and making it less likely that the register allocator will reuse the register. Post-pass code scheduling will fill delay slots from above the instruction or below the instruction but within the control block. Any unfilled branch delay slots will finally be filled from destination or in the worst case with a no-op.

6.6.2. Interface Requirements

Depending on the number of available registers in the target architecture and the sophistication of the assembler, it may be desirable to disable the scheduler. For example, the MIPS assembler performs instruction scheduling. It is possible to enable or disable pre-pass and post-pass instruction scheduling using the compile time switches **PREPASS_SCHED** and **POSTPASS_SCHED** respectively. Filling from destination address may be enabled or disabled using the compile time switch **FILL_FROM_DEST**.

Even if the scheduler is not used, the standard function to fill in unfilled delay slots that can not be addressed with hardware interlock may be required. This is provided by the function **O_handle_unfilled** that will correctly work regardless of use or lack of use of any of the previous scheduling features.

To use any of the generic scheduling functions, a standard interface routine must be provided as a call back from the scheduler to the respective code generator. This function, **O_inq_valid_filler**, is used to create dependence arcs which are unique to a specific architecture. Specifically, the AMD-29000 limits the types of instructions that may be placed immediately after an instruction that modifies the register window. This routine will inform the scheduler about illegal instruction sequences.

The function, **O_inq_valid_filler**, is provide an initial instruction, a potential instruction to immediately follow the initial instruction, and a flag indicating if it is pre-pass or post-pass call. This permits handling dependencies that may only be determined after register allocation has been completed.

One common dependence for a call during pre-pass scheduling is to prevent movement of instructions that may expand into multiple instructions in post-pass code annotation. The scheduler only attempts to move instructions it needs to fill delay slots, however, if the instruction later expands into multiple instructions, it may increase the execution of a common path. For example, if an instruction is moved into a loop and it later expands into multiple

instructions, the processor will execute at least one extra instruction that will ultimately degrade performance.

7. PHASE III -- GENERATION OF ASSEMBLY CODE

7.1. Overview and Organization

Phase III is the final stage of code generation, responsible for emitting the appropriate assembly code and data Sections. It is useful during the debugging stages to emit information such as register and memory requirements to verify the accuracy of the function prologue and epilogue Sections. Most of the early debugging problems occur in these Sections. A template for this phase exists to ease the development process. It is composed of three files that meet all internal and external requirements. The file **p_amd29k_func.c** contains all code required to generate assembly code for a function. The main entry point is **P_process_func** that is called once per function. Typically, the number of supported operations will be less than during phase I annotation. The interface to this function is generic and should not be modified.

The file **p_amd29k.h** contains all variable and function definitions. The file **p_amd29k_data.c** contains all code required to produce data segments for a function. The main entry point is **P_process_data.** Large portions of this file can be left unchanged providing a fast template for data segments. This will be covered in more depth in Section 7.3.2.

7.2. Initialization

The function **P_init** is a generic initialization routine provided as a standard part of the interface to permit one-time initialization for phase III. This routine is called from **l_codegen_main.c** before calling either **P_process_data** or **P_process_func**.

7.3. Interface Requirements

7.3.1. Generating Assembly Code for Functions

Generation of assembly code follows a very similar procedure as annotation of Lcode. Most instructions are similar and can be converted using standard routines. Some of the instructions will require custom routines due to greater complexity. There are five main routines that support the generation of assembly code including **P_process_func**, **P_print_oper**, **P_print_operand_asm**, **P_print_reg**, and **P_print_string_literal**.

7.3.1.1. Function Requirements

Unlike most of the other entry functions, **P_process_func** may require significant modifications to meet the requirements of a given code generator. All string literals are emitted before the start of the function. This will need to be changed if the assembler expects a different approach. Next, function debug information is printed. Currently, this is tied to the **verbose** switch and produces information about registers and memory requirements. Finally, a standard segment of code is executed that scans through all control blocks within the function and calls **P_print_oper** for each Mcode operation.

7.3.1.2. Instruction Requirements

The function **P_print_oper** is similar to **L_annotate_oper**. This function is composed of a large switch statement containing all supported Mcode operations. Straightforward operations can be emitted with the standard function **P_print_normal_oper**. This routine produces a formatted instruction with the opcode first, the destination, source one and source two. This can be easily modified to rearrange the operands based on the requirements of the instruction set. The opcode is an input parameter to this function allowing easy translation of the Mcode operations to different opcodes based upon the extension. A good example of this was shown in Figure 6.3 in the annotation of double-precision move immediate instructions. To maintain the appropriate dependencies, this was implemented with a move followed by three subsequent add

instructions. The extension field is used to determine that the add instructions are really move instructions, thus the appropriate opcode would be passed to **P_print_normal_oper**.

7.3.1.3. Operand Requirements

A standard function, **P_print_operand_asm**, is used to simplify the work for emitting assembly instructions. This routine will handle all supported operand types for the processor. Most of the function is generic with the exception of macro registers. These are customized to each code generator by adding any custom macro registers and formatting the appropriate register and specifiers.

The function **P_print_reg** is used to print out the appropriate register convention based on the physical register number. The AMD-29000 version uses a boundary condition to print global registers or local registers.

7.3.1.4. Other Requirements

As mentioned in Section 7.3.1.1, string literals will be produced at the beginning of the function. The function **P_print_string_literal** is used to print the literals. This routine eliminates any redundant literals within the same file to minimize memory requirements. It will then emit the appropriate Section information to store literals and finally emit the string literal using the function **P_print_string** found in **P_amd29k_data.c**. Modifications will be required to use the appropriate assembly directives for the target processor processor.

7.3.2. Generating Data Sections

Generation of data Sections is a straightforward process. Very few modifications should be required to support most data elements. There are three main routines that support the generation of assembly code including **P_process_data**, **P_process_expr** and **P_print_string**.

The function **P_process_data** is a large switch statement supporting all data types defined in [1]. Modifications will be required to use the appropriate assembler directives for alignment,

global definition and Section definition. Minor modifications are required to replace the data size opcodes of the AMD-29000 with those supported by the target processor.

The function **P_process_expr** will emit expressions as they are defined in Section 2.2.1.5. The processor should require little or no change to this function. The last function is **P_print_string**. This function will produce both null terminated and non-null terminated strings. The architecture may require some modifications to this function. In particular, some architectures have an assembler directive that will automatically null terminate strings. This function assumes that the architecture does not support this directive.

8. CONCLUSIONS

8.1. General

The initial inclination at the early stages of the project was to start from scratch and redefine a machine specific intermediate code that was different from Lcode. As the project progressed, it became apparent that the development efforts required to accomplish such a goal were significant and would not meet the time requirements of the project. It would also increase the complexity in maintaining such a system. As a result, Mcode derived into minor modifications to existing Lcode. A great deal of effort was spent to ensure that existing code from the optimizer and scheduler could be easily ported into the system. As a result, optimization routines and scheduling routines were ported to the template in a matter of days instead months.

The first completed code generator using this model was the AMD-29000. Currently development efforts are in progress for the HP-PA and a port of the existing SPARC code generator to the new model.

8.2. Results

The desire is always to see good performance results at the conclusion of a project. Table 8.1 shows the performance of the AMD-29000 code generator before the new model and the new AMD-29000 code generator using the model. The new code generator averages 16.7 percent faster than the original code generator. It is obvious that these performance improvements resulted from a more efficient register allocator, the use of pre-pass code optimizations and the scheduling. The original code generator did not support instruction scheduling at all. The key benefit that the template approach provided is the developer of a code generator can spend the time focused on learning the architecture of the processor. It is

impractical for development effort to require expertise on register allocation, optimization and instruction scheduling. If this were the case, the development cycle would be intolerable or the quality of the code generators would be inconsistent. Now the standard features will provide consistency that is high-quality across all code generators.

Test	Original Code Generator	New Code Generator	Original/ New
Perm	2398736	2024745	1.185
Towers	2490888	2473419	1.007
Queens	1789606	1730230	1.034
Intmm	4012932	2606923	1.539
Mm	4660989	4491189	1.038
Puzzle	11950376	10505222	1.137
Quick	1992477	1824400	1.092
Bubble	2400717	2325523	1.032
Tree	4273124	3824896	1.117
Oscar	10246233	7797199	1.314
Total	46216078	39603746	1.167

Table 8.1: Execution Times of Stanford Test Suites

APPENDIX A

This section describes useful interface routines which facilitate the code generation process. Except where specifically noted, all routines support both Lcode and Mcode operations.

I/O Routines:

void L_open_input_file(char *file_name); Opens specified file file_name. Returns error if file does not exist.

void L_close_input_file(char *file_name); Closes specified file file_name.

int L_get_input(); Reads next Lcode/Mcode entry in file file_name. Will return a data item (word, etc.) or a function. All fields are parsed, verified and inserted into appropriate data structures. A data description is returned in the data structure L_data. A function description is returned in the data structure L_fn. L_INPUT_EOF is returned when the end of file is encountered.

void L_print_loper(FILE *F, L_Oper *oper); Prints an Lcode operation with the Lop indicator to output F.

void L_print_moper(FILE *F, L_Oper *oper, int verbose); Prints an Mcode operation with the Mop indicator to output F. If the verbose option is true, then the parent Lcode operation will also be printed.

void L_print_mcb(FILE *F, L_cb *cb, int verbose); Prints an Mcode operation with the Mop indicator to output F. If the verbose option is true, then the parent Lcode operations will also be printed.

void L_print_mfunc(FILE *F, L_Func *fn, int verbose); Prints an Lcode function to output F. If the verbose option is true, then the parent Lcode operation will also be printed.

void L_print_data(FILE *F, L_Data *data); Prints an Lcode data element to output F.

Functions that Manipulate Mcode:

L_Oper *L_create_moper(int opcode_id, L_Oper *parent); Creates a new Mcode operation with opcode_id. The link pointer is set to the parent Lcode operation. It fills in the opcode

name and attribute fields and initializes all other fields to 0. It is the responsibility of the calling routine to fill in appropriate source and destination fields.

L_Oper *L_duplicate_moper(L_Oper *oper); Creates an exact duplicate of the input operation. The input operation may be Lcode or Mcode. If it is Lcode, it will be linked as the parent Lcode operation. If it is Mcode, the parent of the Mcode operation will be linked as the parent Lcode operation.

void L_delete_oper(int cb_id, L_Oper *opA); Deletes specified operation from function. cb_id is used to speed up this process since special cases must be handled if operation is first or last operation in the control block.

void L_insert_before(int cb_id, L_Oper *opA, L_Oper *opB); Inserts operation opB before operation opA. cb_id is provided to speed up insertion in the event that opA was the first operation in the control block.

void L_insert_after(int cb_id, L_Oper *opA, L_Oper *opB); Inserts operation opB after operation opA. cb_id is provided to speed up insertion in the event that opA was the last operation in the control block.

Functions that Manipulate Mcode Operands:

void L_new_cb_operand(L_Operand *op, int cb_id); Initializes the fields for operand op with the cb type descriptor, ctype of integer and value field to cb_id. It is responsibility of calling routine to ensure that the field is a valid source or destination operand. No checking is performed to ensure valid use of operands based upon opcode.

void L_new_int_operand(L_Operand *op, int value); Initializes the fields for operand op with the int type descriptor, ctype of integer and value field to value. It is responsibility of calling routine to ensure that the field is a valid source or destination operand. No checking is performed to ensure valid use of operands based upon opcode.

void L_new_float_operand(L_Operand *op, int value); Initializes the fields for operand op with the float type descriptor, ctype of float and value field to value. It is responsibility of calling routine to ensure that the field is a valid source or destination operand. No checking is performed to ensure valid use of operands based upon opcode.

void L_new_double_operand(L_Operand *op, int value); Initializes the fields for operand op with the double type descriptor, ctype of double and value field to value. It is responsibility of calling routine to ensure that the field is a valid source or destination operand. No checking is performed to ensure valid use of operands based upon opcode.

void L_new_string_operand(L_Operand *op, int value); Initializes the fields for operand op with the string type descriptor, ctype of string and value field to the pointer to value. It is responsibility of calling routine to ensure that the field is a valid source or destination operand. No checking is performed to ensure valid use of operands based upon opcode.

void L_new_macro_operand(L_Operand *op, int mac, int ctype); Initializes the fields for operand op with the macro type descriptor, ctype of the specified macro ctype and value field to mac. It is responsibility of calling routine to ensure that the field is a valid source or destination operand. No checking is performed to ensure valid use of operands based upon opcode.

void L_new_register_operand(L_Operand *op, int index, int ctype); Initializes the fields for operand op with the reg type descriptor, ctype of the specified register ctype and value field to index. It is responsibility of calling routine to ensure that the field is a valid source or destination operand. No checking is performed to ensure valid use of operands based upon opcode.

void L_new_label_operand(L_Operand *op, char *label); Initializes the fields for operand op with the label type descriptor, ctype of label and value field to the pointer to value. It is responsibility of calling routine to ensure that the field is a valid source or destination operand. No checking is performed to ensure valid use of operands based upon opcode.

void L_duplicate_operand(L_Operand *dest, L_Operand *src); Copies all operand fields from the operand specified by src to the operand specified by dest. It is responsibility of calling routine to ensure that the field is a valid source or destination operand. No checking is performed to ensure valid use of operands based upon opcode.

Mcode Specific Extensions:

extern void L_set_extension(L_Oper *oper, int ext); Will set the extension of the specified operation to the 32-bit integer value defined by ext. Any previously defined extension fields will be replaced.

extern void L_get_extension(L_Oper *oper, int *ext); Will return the current extension field for the specified operation. If there is no defined extension field, 0 will be returned.

extern void L_set_fill_specs(L_Oper *oper, int fs, int slots); Sets the fill status and number of fill slots to the specified values in fs and slots respectively. Status are defined as follows:

L_NO_STATUS:	No status
L_HW_INTERLOCK:	Processor supports hardware interlock for operation
L_SQUASH_TAKEN_BRANCH:	Squash instruction after branch if branch taken
L_SQUASH_NOT_TAKEN_BRANCH:	Squash instruction after branch if branch not taken.

All of these fields except L_NO_STATUS are bit fields that may be ORed to set a status. The number of fill slots indicates the number of available delay slots for the instruction. Any previously set parameters will be replaced by subsequent calls.

extern void L_get_fill_specs(L_Oper *oper, int *fs, int *slots); Returns the fill status and number of fill slots in fs and slots respectively. Status are defined as follows:

L_NO_STATUS:	No status

L_HW_INTERLOCK:	Processor supports hardware interlock for operation
L_SQUASH_TAKEN_BRANCH:	Squash instruction after branch if branch taken
L_SQUASH_NOT_TAKEN_BRANCH:	Squash instruction after branch if branch not taken.

All of these fields except L_NO_STATUS are bit fields that may be ORed to set a status. The number of fill slots indicates the number of available delay slots for the instruction. Any previously set parameters will be replaced by subsequent calls.

REFERENCES

[1] P.P. Chang and W.W. Hwu, "The Lcode Language And Its Environment", Center for Reliable and High-Performance Computing Report, University of Illinois, January 1991.

[2] P.P. Chang and W.W. Hwu, "Profile-Guided Automatic Inline Expansion for C Programs", Center for Reliable and High-Performance Computing Report, University of Illinois, April 1991.

[3] P.P. Chang, S.A. Mahlke and W.W. Hwu, "Using Profile Information to Assist Classic Code Optimizations", Center for Reliable and High-Performance Computing Report, University of Illinois, April 1991.

[4] A.V. Aho, R. Sethi and J.D. Ullman, Compilers: Principles, Techniques, and Tools, Addison-Wesley Publishing Company, 1986.

[5] P.P. Chang, D.M. Lavery and W.W. Hwu, "The Importance of Prepass Code Scheduling for Superscaler and Superpipelined Processors", Center for Reliable and High-Performance Computing Report, University of Illinois, May 1991.