©Copyright 1991
David Gordon Bradlee

# Retargetable Instruction Scheduling for Pipelined Processors

by

David Gordon Bradlee

A dissertation submitted in partial fulfillment
of the requirements for the degree of

Doctor of Philosophy

University of Washington

1991

Approved by ______________________________

Susan J. Eggers

______________________________

Robert R. Henry

Program Authorized
to Offer Degree ______________________________

Date ______________________________

University of Washington

Abstract

# Retargetable Instruction Scheduling for Pipelined Processors

by David Gordon Bradlee

Chairperson of the Supervisory Committee: Professor Susan J. Eggers
Department of Computer Science and Engineering

Retargetable code generators for complex instruction set computers (CISCs) have focused on sophisticated pattern matching code selection, because CISCs provide many machine instruction sequence choices. Recent pipelined processors, known as reduced instruction set computers (RISCs), provide fewer instruction sequence choices, but expose pipeline and functional unit costs to the compiler. For RISCs the compiler's emphasis must be shifted from code selection to instruction scheduling, resulting in code generation issues that are different than those for CISCs. In particular, the machine description language for a retargetable RISC compiler must contain scheduling requirements. Also, the interaction between register allocation and instruction scheduling is significant.

This dissertation comprises three components. The first component discusses the Marion retargetable code generator system, which includes a machine description language that contains instruction scheduling requirements, along with other code generation information. Using Marion, code generators have been constructed for the MIPS R2000, Motorola 88000, Intel i860. The second component compares three code generation strategies for handling the interaction between instruction scheduling and register allocation, including one that I developed, RASE, that integrates the two phases. On a computation-intensive workload for three RISCs, RASE produces significantly better code than the Postpass strategy, which does not integrate the two phases, and slightly better code than IPS, which integrates the two phases to a lesser degree than RASE. The third component investigates the interaction between code generation strategies and architectural features, including register set size and structure, and operation and load latencies. On a computation-intensive workload, 64 registers yields a significant improvement over 32 registers for Postpass, but little improvement for IPS or RASE. This study also shows that, on architectures with long latencies or small register sets, or on programs with large basic blocks, RASE produces significantly better code than IPS.

# Table of Contents

# List of Figures

# List of Tables

# ACKNOWLEDGEMENTS

My co-advisors, Robert Henry and Susan Eggers, have formed a capable, energetic team. Robert has been my advisor for five and a half years. He has prompted me to examine diverse perspectives and has helped shape this research through his insight, flexibility and encouragement. Susan has been my advisor for only a year; in this short time she has helped me define specific goals and achieve them.

Larry Snyder read this dissertation and provided helpful comments. Hank Levy and Werner Stuetzle rounded out my committee.

Jean-Loup Baer, the chairperson of the Department of Computer Science and Engineering, has provided a flexible, open environment that fosters communication among professors and students, and participation by computer scientists outside of the department.

My Woodberry Forest School mentor, Harrison W. "Chuck" Straley, saw my budding interest in mathematics and inspired me to pursue it to the fullest. Under his tutelage I accelerated my study of mathematics, began exploring computer science and gave my first public scientific presentation.

I would like to acknowledge my Union College professors, Bill Zwicker, for putting up with me in his advanced mathematics classes, and Ted Schwartz, for his excellent computer science teaching and all-round contagious enthusiasm.

A number of fellow students deserve acknowledgement. Rakesh Kumar Sinha and Wendy Thrash reviewed parts of this dissertation. David "Pardo" Keppell discussed many ideas with me and was interested in everything. David "Pablo" Cohn has managed to share both the same office and the same house with me for over three years. Gail Harrison Alverson, Rick Zucker, Dave Socha, Calvin Lin, Charles Loop, Bill Griswold, Rajendra Raj, Kevin Sullivan, Radhika Thekkath, Chandu Thekkath, Rob Bedichek, Elizabeth Walkup, Peter Damron, Bill Mershon, Craig Anderson and Alex Klaiber all have made the graduate program interesting, exciting and, most of all, fun. Calvin and Charles made sure I got enough exercise. Craig and Alex fed Black Nose the cat. Bill Mershon helped build my deck.

Bob Scheulen, Trapper Jeff Robbins, Roxanne Everett, Scott Wittet, Roberta Riley and Susan Schwartz have all been tremendous friends, willing to listen to my complaints and encouraging me to participate in non-computer science activities.

Mark Roberts was a supportive friend and supervisor at Microsoft over numerous leaves of absence and levels of part-time work. David Jones initiated my flexible work schedule at Microsoft. Bob Scheulen discussed countless code generation and optimization issues with me, and gave me much moral support.

My great aunt, Marion Lovat Fraser, was a wonderful, positive presence in my life from the time I was a small child until she passed away a few years ago. After I began working with computers, Aunt Marion would always introduce me to her friends, "This is my nephew David. He's a computer." The code generator system that is central to this dissertation is named after Aunt Marion. Now I can say that *she* is a computer system.

My parents, Adele and Merrill Bradlee, instilled in me a positive attitude and a desire to learn. They have always provided a stable emotional base and encouraged me to pursue my own intellectual directions. My brothers, Jay and Rick Sprague, provided a learning environment in which to grow.

My wife's parents, Ernie and Marge Gardow, have been very encouraging and understanding.

Kathryn Gardow, my wife, has been invaluable. Kathryn was willing to move across the country so that I could attend graduate school. She has earned the money to pay the mortgage and to enable us to travel. She has dragged me into the mountains for much-needed exercise and interruption. She has created gourmet delights for the palette of this famished graduate student. She has provided social diversion. She has even proofread this dissertation. Most of all she has been strong, loving and completely supportive. Although she would have liked me to finish faster, she never suggested that I quit. To Kathryn I give my endless thanks.

# Chapter 1

# Introduction

A compiler inputs a high-level language program, transforms it into an intermediate language (IL), and then performs *code generation*, which outputs a semantically equivalent program in a *target language*, the target machine's instruction set. Code generation includes *code selection*, the process of mapping the IL onto the target language. Code generation may also include *global register allocation*, which maps user variables and compiler-generated temporaries to machine registers over an entire procedure, and *instruction scheduling*, which rearranges machine instructions to avoid pipeline delays, thereby producing code that more efficiently uses the target's pipelines and functional units.

A *retargetable* code generator is one that can be changed automatically, by giving it a description of a new target machine, so that it generates code for that new target. Retargetable code generators are important for several reasons. First, processor performance continues to improve, inducing users to upgrade their systems. Second, the volume of software that must be ported to a new architecture is large enough to prohibit the extensive use of assembly language code to achieve high performance. Therefore, for a new architecture to compete, its compiler must generate quality code. Third, even within a processor family there can be significant architectural distinctions; between families the differences are magnified. Together, these factors imply that processor manufacturers need to be able to construct high quality code generators quickly. A retargetable system can make this possible, if it enables good code to be produced with a minimum of effort and, therefore, frees human resources for other tasks, such as architectural feature evaluation, target-independent compiler optimizations or other software projects.

Retargetable code generators developed in the last decade have focused on code selection for complex instruction set computer (CISC) architectures, such as the DEC VAX [Dig81], the IBM 360 [ABB64] and the Intel 8086 [Int83]. Since CISCs implement the most common operations many different ways and hide pipelining details and other feature costs, their companion code generators used machine specifications that facilitated sophisticated pattern matching code selection. These code generators could ignore instruction scheduling and, in some cases, did not perform global (procedure-wide) register allocation.

Recent pipelined processors, commonly known as reduced instruction set computers (RISCs), implement most operations only one way and expose pipeline and functional unit costs to the compiler. Therefore, the compiler's emphasis must be shifted from code selection to instruction scheduling. In addition, because computation must be done in registers and because the ratio of memory access time to cycle time is high, some form of global register allocation is necessary to effectively use the larger register sets found on most RISCs.

Because of the shift in functional emphasis, the code generation issues for RISCs are different than those for CISCs. First, for a retargetable RISC compiler, the machine specification must capture most scheduling information, including operation latencies and resource conflicts. Second, since RISC code selection is relatively simple, the interaction between code selection and register allocation is less important. Instead, the interaction between register allocation and instruction scheduling is significant, because the scheduler needs registers to overlap the execution of independent operations.

The goal of this dissertation is to investigate the following aspects of RISC code generation:

(1) Is retargetable RISC code generation feasible and profitable?

(2) Can the important aspects of code generation be specified in a practical form?

(3) Can that specification be transformed into an effective code generator and does it provide any benefit over hand-written code generation?

(4) In a RISC code generator, to what degree is communication necessary between register allocation and instruction scheduling?

My thesis is that the answer to all of these questions is "Yes." My approach to retargetable RISC code generation is to blend the experience of retargetable CISC code generators with techniques from instruction scheduling and register allocation to create the *Marion* retargetable code generation system. To demonstrate feasibility, I have constructed code generators for three commercially available RISC architectures and over thirty-five variations of these architectures. To examine the interaction between register allocation and instruction scheduling, I have compared three code generation strategies, each of which represents a different degree of communication between the two phases. I have also used Marion to investigate the effect of architectural variations on performance. The facility of building new code generators, which enabled the strategy comparisons and architectural experiments, exhibits the profitability of a retargetable system.

## 1.1 Retargetable Instruction Scheduling

There are several approaches to including scheduling information in a retargetable code generator. A simple method is to use an existing retargetable code generator to perform code selection and then add a table to drive instruction scheduling by associating operation latency and resource use information with each instruction. After code selection, the table is used to build a directed acyclic graph (DAG), from which instructions are scheduled. Retargeting involves replacing the table, in addition to providing a new code selector specification to the retargetable code generator. The advantage of this approach is that changing the table is straight-forward. For a simple RISC this is sufficient. The disadvantage is that information is kept in multiple formats. To retarget the code generator, both the code selector specification and scheduling information must be modified. In addition, many RISCs have some complicated features, such as latencies that are dependent on the consumer and producer instructions together, non-orthogonal bypasses, or the need to access parts of registers. These features complicate the basic scheme, requiring additional tables or procedures that must be modified when retargeting.

At the other end of the spectrum is a low-level model of the target machine that includes all stages, latches, buses, bypasses and multiplexers. This model could capture all useful scheduling information, but the code generator would have to include all of that detail, complicating implementation and maintenance. In addition, since the machine specification is detailed, it would be difficult to write, except by the architects themselves.

The Marion retargetable code generator system lies in between these two extremes. With Marion, the compiler writer constructs a machine description that lists each instruction along with the operation it performs and its scheduling requirements. The description language is easy to use, yet provides enough constructs to support a broad range of RISCs. It can model many complicated machine features, including pipelines that do not advance on every cycle or those that can be fed directly from other pipelines. From this description Marion builds a code generator that includes code selection, instruction scheduling and global register allocation.

Marion was designed specifically for uniprocessor RISCs that contain multiple functional units and multi-cycle operations. It has been used successfully to produce code generators for the Motorola 88000 [Mot88], the MIPS R2000 [Kan87] and the Intel i860 [Int89]. The 88000 and the R2000 are "traditional" RISC machines that are more similar than different; in contrast, the i860 can issue two instructions per cycle and has explicitly advanced floating point pipelines, making it an extremely challenging target. Marion is the first retargetable code generator system that includes a quality instruction scheduler and can handle machines in the i860's class.

## 1.2 Code Generation Strategies

RISC processors rely on compilers to perform register allocation to reduce memory references and instruction scheduling to avoid pipeline hazards. A compiler that effectively manages register use by these two phases gives computer architects greater flexibility in making architectural design decisions, and enables users to realize a RISC's high performance potential.

Many compilers for uniprocessor RISC systems perform global register allocation and instruction scheduling separately, with each phase ignorant of the requirements of the other. Instruction scheduling uses registers to exploit instruction-level parallelism;[1] register allocation uses registers to reduce memory references. Because the goals of the two phases often conflict, whichever phase is first imposes constraints on the other, sometimes producing inefficient code. For example, if register allocation is performed first, it may limit the instruction scheduler's rearrangement choices by assigning the same physical register to independent expression temporaries, which prevents the expressions from being overlapped. When scheduling precedes register allocation, the number of simultaneously live values, or *register pressure*, may be increased, causing many of these values to be spilled to memory.

To determine the extent of communication needed between the two phases, I compare the performance of three code generation strategies. The *code generation strategy* refers to the invocation order of and degree of communication between register allocation and instruction scheduling. The strategies include the following:

(1) a simple strategy called Postpass, in which global register allocation is performed prior to instruction scheduling, with no communication between the phases;

[1] *Instruction-level parallelism* at a point in the program refers to the number of instructions that could legally be executing concurrently. On uniprocessors, *realized* instruction-level parallelism is the number of operations executing concurrently in the various pipeline stages and functional units.

(2) a strategy with an intermediate degree of communication, called IPS, in which the scheduler is invoked before register allocation and must schedule within a local register limit; and

(3) a tightly integrated strategy that I developed, called RASE, in which the instruction scheduler communicates with the register allocator by giving the allocator cost estimates that quantify the effect of its allocation choices on the subsequently generated schedule.

The results show that some level of communication is necessary to generate code that efficiently uses the registers. The intermediate strategy is sufficient for most programs on most architectures. Nevertheless, on architectures with a small number of registers or long latencies, the tightly integrated strategy is superior for larger programs.

## 1.3 Architecture Investigations

One of the benefits of a retargetable compiler system, when used with profiling and analysis tools, is that it can allow designers to experiment with architectural variations in an effort to produce a superior compiler/architecture partnership.

I have used Marion to examine the effect on RISC performance of code generation strategy, register set size and organization, and operation and load latencies. The experiments vary the number of registers from 16 to 128, in both split and shared register organizations and examine two sets of floating point operation latencies and two sets of load latencies, all in the context of the three different code generation strategies. The results show that the sophisticated code generation strategies produce superior schedules over all architectural variations and can significantly reduce the number of registers needed to achieve good performance.

## 1.4 Dissertation Organization

The first part of this dissertation discusses retargetable instruction scheduling issues and describes the Marion system. Chapter 2 outlines the salient RISC architectural features that affect code generation. These include register set size and structure, operation latencies, pipeline structures and individual machine idiosyncrasies. Chapter 3 describes the major components of the Marion system: the code generator generator, the front end and intermediate language, the code selector and the register allocator. This chapter also outlines Marion's machine description language, along with previous retargetable compiler research. Chapter 4 describes the data structures and methods employed by the Marion instruction schedulers. It introduces *temporal scheduling*, which schedules for explicitly advanced pipelines (pipelines that do not advance on every clock cycle), and presents the temporal scheduling algorithms and proofs of correctness. Previous instruction scheduling research is also discussed here.

The second part of this dissertation discusses the experiments that have been performed with Marion. Chapter 5 gives the methodology used by the experiments, including the workload and measurement criteria. Chapter 6 compares the three code generation strategies. It outlines the problem with compilers that lack communication between register allocation and instruction scheduling, describes each of the strategies, and presents the results of the comparison. Chapter 7 assesses the effect on performance of register set size and structure versus code generation strategy. Other architectural aspects, including operation and load latencies, are examined as well.

The third part of this dissertation includes Chapter 8, which evaluates Marion. It presents data on the size and speed of the system, and size and composition of the machine descriptions. It also examines how Marion handles various RISC features. Chapter 9 concludes with a discussion of future directions.

# Chapter 2

# Pipelined Architectures

The focus of this research is on computer architectures commonly known as RISCs [PS82], in which pipelines and functional units are exposed to the compiler. This chapter outlines typical RISC features, particularly those that affect code generation, and discusses notable features of specific architectures, including the MIPS R2000, Motorola 88000 and Intel i860.

Several properties are typical among RISC architectures [HP90]:

(1) Integer and addressing operations use general purpose registers; floating point operations use either the same register set or a separate general purpose register set. General purpose registers are interchangeable; no instruction requires a specific one.

(2) ALU and floating point instruction operands are registers; only load and store instructions access memory.

(3) Each instruction's resource requirements, including pipeline stages and functional units, are known at compile-time.

The first two properties taken together allow register allocation to be separated from code selection without any negative impact on the generated code; this is because the instruction sequence to compute an expression is independent of which registers are used as operands and independent of whether intermediate results are placed in memory. The third property allows the compiler to perform instruction scheduling to avoid pipeline hazards; pipeline hazards waste machine cycles, decreasing performance.

There are three types of pipeline hazards:

(1) a *data hazard* occurs when an instruction's input operand is not yet available, because either the load of the operand or the computation of the operand has not completed;

(2) a *structural hazard* occurs when two instructions need the same resource on the same cycle;

(3) a *control hazard* is caused by a branch instruction, if the branch target address is not known when the CPU is ready to fetch the target instruction.

A data hazard example is a floating point add followed by a store of the result. If the add latency is 3 cycles, then the processor will stall for 2 cycles, while the add completes, before executing the store. A structural hazard can occur on a shared register file architecture, for example, if a floating point operation and an integer operation both require the write-back bus on the same cycle. Most RISC architectures attempt to avoid control hazards via delayed branches. A *delayed branch* is a branch instruction that has one or more delay slots following the branch. Instructions in delay slots are issued after the branch but before the branch target. The compiler must fill these slots with no-ops, if no useful instructions can be scheduled there.

The code generator's view of the target machine comprises the instruction set, the register file and, for RISCs, the pipelines and functional units. The following are some relevant attributes that must be considered by a retargetable RISC code generator system and the effects they have on code generator design:

(1) a register file split into integer and floating point partitions versus a shared register file. The register allocator must handle multiple register sets and register pairs.

(2) pipelined versus non-pipelined floating point execution units. The instruction scheduler must keep track of resource use to avoid structural hazards.

(3) the number of instructions issued per cycle. The scheduler must consider this to determine where data and structural hazards could occur.

The next three sections discuss the architectures that are models for those used in the studies in this dissertation. All perform 32-bit integer operations and 32- and 64-bit floating point operations.

## 2.1 MIPS R2000

The MIPS R2000 architecture [Kan87] has a split register file that consists of 32 integer registers and sixteen 64-bit floating point registers. The integer register $r0$ is hardwired to 0. Integer multiply and divide place their results in the special registers called HI and LO; instructions are provided to move from HI and LO to the integer registers.

The R2000 CPU contains a 5-stage instruction pipeline; one instruction is issued per cycle. All integer operations, except multiply and divide, use the ALU stage for one cycle. A non-pipelined functional unit performs integer multiply and divide. All integer ALU instructions have either two register operands and one immediate operand or three register operands. Multiply and divide each take two input register operands and send the result to the HI and LO registers.

Floating point instructions are recognized and handled by the floating point coprocessor. All floating point operations use the floating point ALU for two or more cycles. Multiply and divide each use separate non-pipelined units for additional cycles.

The CPU performs all load and store instructions. The only addressing mode is *register + displacement*. The load is followed by one delay slot, which must be filled by the compiler. Double precision floating point values require two instructions to load or store. Instructions to move between the CPU and FPU register sets are also provided.

The R2000 has no condition codes. The *beq* and *bne* compare and branch instructions take two register operands; other compare and branch instructions compare a register against zero. Thus, $(a < b)$ requires two instructions: a subtract followed by a compare and branch if less than zero. The floating point compare instructions take two register operands. The boolean result is sent to the CPU; after a 2-cycle latency, a conditional branch may test the result. All branches and jumps have one delay slot that must be filled by the compiler; the instruction in the delay slot is always executed. The R2000 also has conditional call instructions and set-on-condition instructions. The set-on-condition compares two values and sets the result register to 1, if the condition is true, or 0, if false.

Table 2.1: **R2000 latencies**. Operations not shown have a latency of 1. Both integer multiply and divide require an additional instruction to move the result back to the integer registers.

| Instruction | Latency (cycles) |
|---|---|
| load (int) | 2 |
| load (float) | 2 |
| compare (float) | 2 |
| mul (int) | 11 |
| div (int) | 36 |
| add (float) | 2 |
| mul (float) | 4 |
| mul (double) | 5 |
| div (float) | 12 |
| div (double) | 19 |

The FPU supports single and double precision floating point operations, including multiply and divide. Conversions between integer and floats are also handled by the FPU. Latencies are shown in Table 2.1.

## 2.2 Motorola 88000

The Motorola 88000 architecture [Mot88] has 32 general purpose registers that are used for both integer and floating point values. Double precision floats use register pairs. Register $r0$ is hard-wired to 0.

The CPU has a 4-stage instruction pipeline. All integer operations, except multiply and divide, use the ALU stage for one cycle. All ALU operations take three operands. Integer multiply and divide use the FPU. A 3-stage data unit pipeline performs address calculations and accesses the memory hierarchy.

The FPU is split into two pipelines: a 4-stage multiply pipeline, which performs floating point and integer multiply, and a 3-stage add pipeline, which handles integer divide and all other floating point operations. The two pipelines share additional initial and final stages. Divide iterates within the first add pipestage; all other operations are completely pipelined. All floating point instructions, except unary operations, take three register operands.

Three operations can contend for the final FPU stage: a multiply, an add and an integer multiply that exits the multiply pipeline early. A priority scheme gives integer multiply the highest priority and then floating point multiply the next highest priority. Additionally, contention for the register write-back bus can occur between the FPU, the integer ALU and the data unit. The ALU is given the highest priority and data unit the lowest.

The 88000 has three addressing modes, *register + displacement*, *register + register* and scaled index register (*register + register * scale*). The 16-bit displacement is zero-extended. For scaled index register mode the scale is the size of the object referenced (either 1, 2, 4 or 8 bytes).

Bit field instructions set, clear, extract and rotate bit fields within a register. Shift instructions are default cases of these operations.

Table 2.2: **88000 latencies**. Operations not shown have a latency of 1.

| Instruction | Latency (cycles) |
|---|---:|
| load (int) | 3 |
| load (float) | 3 |
| load (float) | 4 |
| compare (float) | 5 |
| add (float) | 5 |
| mul (int) | 4 |
| mul (float) | 6 |
| mul (double) | 9 |
| div (int) | 38 |
| div (float) | 30 |
| div (double) | 60 |

To conditionally branch, the compare instruction sets condition bits in the destination general purpose register. Then a branch-on-condition instruction tests the condition bits and branches if appropriate. Branches have both delayed and non-delayed forms. Delayed branches have one delay slot; the instruction in the slot is always executed.

Table 2.2 gives the 88000 operation and load latencies. In some cases, the latency depends on the instruction that consumes a value. For example, if the result of a double precision floating point add is stored to memory, then the add latency is 6 instead of 5.

## 2.3 Intel i860

The Intel i860 [Int89] has 32 integer registers and thirty-two 32-bit floating point registers. The floating point registers can be accessed in 64-bit even/odd pairs for double precision. The integer register $r0$ and the floating point registers $f0$ and $f1$ are hard-wired to 0.

The i860 has a 4-stage instruction pipeline. The integer ALU performs all integer operations in one cycle, except multiply and divide. All ALU operations take three operands. The FPU performs all floating point operations and integer multiply. There is no divide instruction; a reciprocal and a sequence of subtracts and multiplies performs the operation. An FPU instruction can be issued together with an ALU or memory instruction on each cycle.

The CPU unit performs loads and stores for both integer and floating point registers. Addressing modes include *register + displacement* and *register + register*. However, an integer store may use only the former. In addition, floating point loads and stores may use auto-increment versions of both modes. The i860 also supports a pipelined floating point load that bypasses the cache and uses a load pipeline; the load pipeline is not used by other load or store instructions.

Most conditional branches use the condition code bit, which is set by integer ALU operations and by floating point compare operations. Conditional branches come in two forms: without a delay slot and with a delay slot that is executed only if the branch is taken. An instruction that immediately follows a branch without a delay slot is executed only if the branch is not taken. If the branch is taken, the pipeline stalls until the branch target instruction is fetched. The unconditional branch has a delay slot that is always executed. Other conditional branches perform compare and branch on equal or not equal; these have no delay slots. Finally, the *bla* instruction decrements an integer register (a loop counter) and sets the loop condition code bit LCC to 0 when the loop counter reaches 0. The *bla* instruction branches if the previous value of LCC is 1. The delay slot instruction is always executed.

The FPU contains a 3-stage floating point adder, a 3-stage floating point multiplier, a floating point load pipeline and a graphics unit pipeline. The load pipeline is used only by the pipelined loads that bypass the cache (mentioned above). The graphics pipeline supports various graphics instructions, but is not considered in this research.

FPU instructions come in two varieties: *scalar* and *pipelined*. Scalar instructions require exclusive use of the entire FPU, except for the register write-back hardware. Each pipelined instruction explicitly advances a particular pipeline (or pipelines), leaving other pipelines unchanged. All four pipelines are explicitly advanced.

Some scalar instructions have no pipelined equivalent. These include reciprocal, reciprocal square root and the *fmlow* instruction, which is used to perform integer multiply.

A pipelined instruction takes three register operands: two sources and one result. When the instruction is executed, the source registers are fed into the appropriate pipeline. The result register gets whatever is coming out of the pipeline on the cycle that the instruction is issued; this is the result of a previous operation, not the result of the operation that is launched by the instruction. For example, the following sequence performs $f7 \leftarrow f5 + f6$:

| | |
|---|---|
| *pfadd* $f5, f6, f0$ | ; stage1 $\leftarrow f5 * f6$ |
| *pfadd* $f0, f0, f0$ | ; stage2 $\leftarrow$ stage1 |
| *pfadd* $f0, f0, f0$ | ; stage3 $\leftarrow$ stage2 |
| *pfadd* $f0, f0, f7$ | ; $f7 \leftarrow$ stage3 |

The first *pfadd* launches the add, the second and third advance the pipeline and the fourth catches the result.

Most pipelined instructions advance only one pipeline. However, the i860 has a set of instructions that advance the multiply and add pipelines together. These instructions take only three register operands. Therefore, two additional source operands and one additional destination operand are needed. These are provided by special registers KI, KR and T or by *chaining*, feeding the output of the adder directly to the multiplier or vice versa. Some instructions set one or more of the special registers; others leave them alone. The T register is fed only by the multiplier and feeds the adder; KI and KR are loaded from general purpose registers and feed the multiplier.

Table 2.3 shows the i860 operation and load latencies. Like the 88000, there are cases where the latency depends on the interaction between two instructions, particularly floating point stores and floating point operation write-backs.

Table 2.3: **i860 latencies**. Operations not shown have a latency of 1.

| Instruction | Latency (cycles) |
|---|---:|
| load (int) | 2 |
| load (float) | 3 |
| load (double) | 3 |
| compare (float) | 2 |
| add (float) | 3 |
| add (double) | 3 |
| mul (int) | 3 |
| mul (float) | 3 |
| mul (double) | 4 |

# Chapter 3

# Marion Code Generation Construction System

The Marion Code Generator Construction System is a retargetable code generator system designed specifically for uniprocessor RISCs that contain multiple functional units and multi-cycle operations. Each code generator is constructed from a natural machine description and performs code selection, instruction scheduling and global register allocation. The description language is easy to use, yet provides enough constructs to support a broad range of RISCs.

Marion comprises three major components (see Figure 3.1): the code generator generator, the compiler front end and the compiler back end. The input to the code generator generator is a machine description that describes the target's registers, functional units, pipeline stages and instructions, including scheduling properties; the output is a set of automatically generated tables and routines. The Lcc front end [FH90] consumes ANSI C and generates an intermediate language (IL) of directed acyclic graphs (DAGs) built from typed low-level operators. The back end transforms the IL, according to transformations specified in the machine description, performs code selection by pattern matching and then gives control to the code generation strategy.

The code generation strategy directs the invocation of and degree of communication between instruction scheduling and global (procedure-wide) register allocation. It also includes the scheduling algorithm. The register allocator and scheduling support (which handles the low-level scheduling details) are independent of both the code generation strategy and the target machine. In addition, the strategy module accesses target-dependent data only through the target- and strategy-independent routines. The separation of the strategy module from the rest of the code generator allows a strategy to be replaced quickly, without changing the other components or the interface between the target-dependent and target-independent portions of the code generator.

Marion makes several assumptions about the target architecture. First, the architecture must have general purpose register sets for integer, addressing and floating point operations. Second, the scheduling requirements for each instruction, including which functional units and pipeline stages are used on each cycle, must be known at compile time. Third, only load and store operations may access memory. RISCs generally comply with these assumptions. Machines with hidden pipelines, such as the VAX, could be modeled using Marion by supplying null scheduling information and using a null scheduler.

This chapter first examines related work in modeling instruction scheduling and retargetable code generators. It then describes the Marion's machine description language, *Maril*, and system components, including the front end, the IL, the code generator generator, the code selector and the register allocator. Chapter 4 describes Marion's instruction scheduling and some additional scheduling-related Maril constructs.

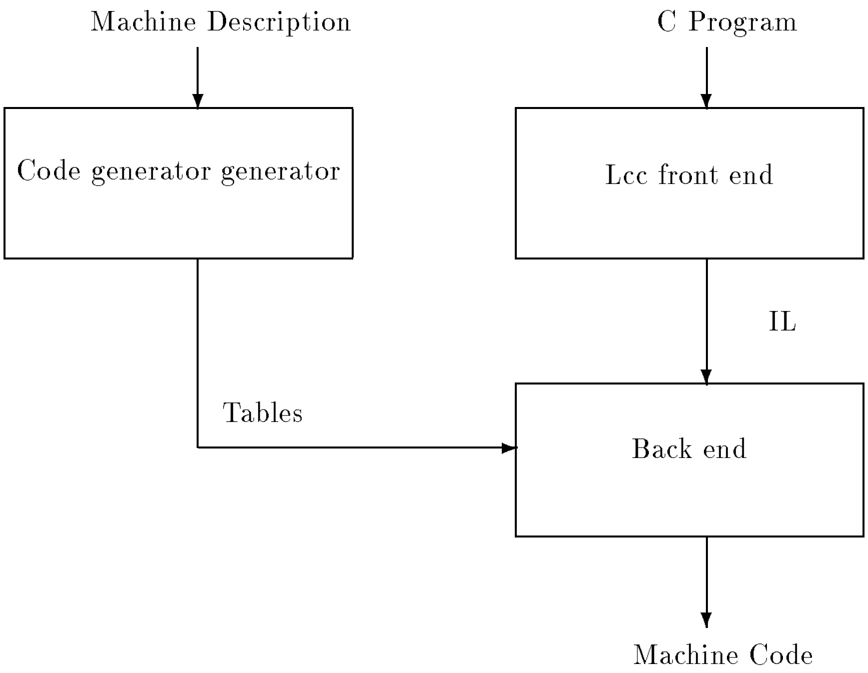



Figure 3.1: **Marion structure**. The code generator generator is invoked once per target. The tables it produces are linked in with the compiler.

## 3.1 Related Work

I implemented yet another retargetable code generator, because no existing system has a machine description language that incorporates scheduling constraints or builds an instruction scheduler from the description.

The ISP (instruction set processor) description language [BN71] was not appropriate for Marion, because it contains more detail than necessary in some areas, but not enough in others. For example, the code generator does not need a detailed program status word format description, but it does need to know the scheduling properties of each instruction.

Tree and string grammars have been used by retargetable compilers for complex instruction set computers (CISCs) to describe target machines. Various systems produce sophisticated code selectors for CISCs, but none incorporate global register allocation or instruction scheduling. Four system families are outlined below.

Glanville and Graham [GG78, GHS82] built one of the first retargetable code generator systems. Their system takes a machine description grammar as input and produces a code generator that employs LR parsing techniques to perform pattern matching on the IL. Grammar productions comprise patterns, which contain terminal and non-terminal symbols, and replacements, which are single non-terminals. Actions associated with productions cause machine code to be generated. Henry's CODEGEN [AGH+84, Hen87, HD89a, HD89b] is a successor of the Graham-Glanville system. It takes PPRACs (pattern, predicate, replacement, action, cost) as input and produces a tree pattern matcher. PPRACs are similar to the productions in the Graham-Glanville system, but have predicates and costs to direct the matching. The matcher can employ a number of techniques, including LR parsing, recursive descent parsing, top-down matching and bottom-up matching. The techniques differ in generated code efficiency, compiler construction time, compiler speed and code/table size. The compiler writer can choose the technique appropriate for his system. CODEGEN also includes a compiler writer directed tree transformer that aids in mapping the IL to the target machine. CODEGEN's primary advantage is that virtually all aspects of the target machine can be captured in a single framework: the grammar syntax. The disadvantage is that it is difficult to capture some side effects, such as auto-increment addressing. In addition, the tables that drive matching and reduction can be large; however, Fraser and Henry have successfully minimized this problem [FH91].

The second family, by Ganapathi and Fischer, used an attribute grammar to produce a code generator [GF82, GF85]. Attributes and predicates associated with grammar productions help describe architectural restrictions. This approach puts less emphasis on the syntax than the Graham-Glanville systems. The advantage is that the parse table size is smaller and side effects can be more easily captured. The disadvantage is that the compiler writer must use two models: the grammar syntax and the attribute rules. A descendant of this system [AGT89] uses a dynamic programming tree rewriting algorithm to generate code.

The third family, Davidson and Fraser's PO (and successors) [DF80, DF84a, DF84b, FW86], takes a different approach to CISC code generation. Naive machine code is generated and then optimized, using a peephole optimizer produced from a machine description. PO's machine descriptions are simpler than the descriptions for the Graham-Glanville family, allowing easier retargeting, but no there is flexibility in choosing between compiler speed and code/table size. R. R. Kessler's Peep [Kes84] and P. B. Kessler's system [Kes86] are similar.

Davidson has extended PO to perform instruction reorganization, including targeting and evaluation order determination, to reduce register use [Dav86]. The system is primarily for CISCs, but for one RISC target the reorganization also considered pipeline delays. However, the function to check for delays is hand-written; no scheduling information is derived from the machine specification.

The fourth retargetable code generation system for CISCs is the Production-Quality Compiler-Compiler project [CNL79, Cat80, LCH+80, Lev81], which is a descendant of the Bliss-11 compiler [WJW+75]. This project attempted to include a high degree of detail about the target machine, so that tradeoffs in optimization, code selection and register allocation could be examined. Because it tried to incorporate too much target machine detail, the system was difficult to use.

Some previous systems have incorporated instruction scheduling information in some form. First, the B♯ system of Bird [Bir87] merges instruction selection and scheduling into an L-attributed Graham-Glanville style code generator. Bird's model takes only trees and does not take DAGs. It also requires that the trees have width less than the number of registers, because it cannot handle spills, and assumes that all instructions use the pipeline identically. Second, Wall and Powell's Mahler [WP87] uses scheduling information but does not have an explicit machine description language. Mahler uses an assembly language for a simple virtual machine without a pipeline and with an infinite number of registers. The scheduling details are contained within the Mahler translator; therefore, a change in the target architecture requires changing the translator by hand.

Landskov, Davidson *et al.* [LDSM80, DLSM81] describe a machine model to support microcode compaction. In their model, each micro-operation is described by a 6-tuple indicating the name, inputs, output, resources needs, clock phase and instruction fields for the operation. The inputs and output can be registers, buses or latches. Resource needs are functional units or pipestages. The clock phase supports micro-operations that use only a portion of the machine's cycle. The model is flexible enough to handle a number of compaction problems, but it contains more detail than is needed for RISCS; for example, the clock phase and instruction fields are not necessary. In addition, the 6-tuple lacks the instruction's operation, which is needed for code selection.

The GNU C compiler [Sta89] is a successful retargetable compiler that uses interpreted machine descriptions resembling Davidson and Fraser's PO. Although GNU C has been targeted to many RISCs, the machine description contains no scheduling information. Tiemann has written a scheduler for GNU C [Tie89]. Target-dependent latency and resource information is encapsulated. (He does not indicate what the encapsulated form is.)

## 3.2 Front End and Intermediate Language

Marion uses Lcc, a robust, easily portable, ANSI-C front end written by Fraser and Hanson [FH90]. It performs local optimizations, including common subexpression elimination and constant folding, but no global or loop optimizations.

The IL consists of basic blocks made up of DAGs of low-level typed operators. A thread runs through some DAG nodes, mostly statement-level operators, such as *assign* and *branch*; this maintains a legal ordering. The front end transforms all high-level control flow operators, such as *for*, *while* and *if*, into low-level compare and branch operations. All C Language operators with side effects, such as the assignment operators, the pre- and post-increment and pre- and post-decrement operators and the conditional operator, are transformed into explicit arithmetic, logical, assignment and branch operations. All type conversions are explicit.

Marion applies two sets of transformations to the IL and then feeds it to the code selector. The code selector uses a top-down pattern matching algorithm, which matches subject tree nodes against pattern tree nodes. *Subject* trees are subtrees of the IL DAG that is the input to the code selector. *Pattern* trees comprise IL operators and are derived from the machine description.

The first set of IL transformations performs data allocation, using some machine dependent information. For each user object, the data allocator allocates one of the following:

(1) a pseudo-register, if the object is a simple automatic variable that may reside in a register;

(2) a stack offset from the frame or stack pointer, if the object is automatic, but cannot reside in a register;

(3) an offset from the global data pointer, if the machine model supports one and the object is static; or

(4) a relocatable address.

The second set of transformations puts the IL into a canonical form:

(1) address operators are expanded, (*e.g.*, into *frame pointer + offset*), or changed to pseudo-register references;

(2) nodes are inserted to move function arguments and return values into registers or stack locations; and

(3) the function prologue and epilogue, including trees to save and restore preserved registers, are made explicit.

Marion employs a simple memory reference disambiguation algorithm that supports local common subexpression elimination and permits the instruction scheduler to reorder memory references. Given two address expressions, the algorithm eliminates addends that appear in both; if the remaining addends are demonstrably different, such as different constants, then the references are distinct. This scheme handles array references whose index expressions contain the same set of variables. Ellis *et al.* [FERN84, Ell85] developed a more sophisticated algorithm that derives and compares symbolic expressions for the array addresses, covering a larger number of cases.

## 3.3 Marion Machine Description Language

Marion's machine description language, Maril, incorporates ideas from ISP, CODEGEN, the PO derivatives and the microcode work. Maril requires processor attributes to be declared explicitly, in a manner similar to ISP's. Like PO, each instruction in a machine description is listed along with the operation it performs, expressed in terms of IL operators; pattern trees derived from these expressions are used to match the subject IL. Like CODEGEN, Maril includes tree transformations, which Marion applies to the IL prior to code selection, to facilitate the IL-to-target-machine mapping. Maril also borrows from the microcode compaction model in its specification of resource requirements and restrictions on instruction packing.[1] Marion's contribution is the incorporation of instruction scheduling information into a machine description language that is used for all phases of code generation.

A Maril machine description comprises three sections: **Declare** specifies features of the architecture, such as the registers, pipeline stages and functional units; **CWVM** (Compiler Writer's Virtual Machine) describes a simple runtime model, primarily for parameter binding; **Instr** lists each machine instruction, along with its function and scheduling requirements, and includes tree transformations.

Most aspects of Maril are straight-forward. The information in **Declare** and **CWVM** and the instruction directives all follow directly from an architectural description, such as a programmer's reference manual. Most scheduling properties can be expressed simply, as operation latencies and pipeline resource requirements associated with instructions. Maril's innovation is in the handling of architectural details: the special latency specifications, the escape functions, the instruction packing restrictions and the support for explicitly advanced pipelines. The significance of Maril is that a user can quickly write a basic machine description that generates correct code for all IL constructs and then concentrate on tuning the description with Maril's special constructs to produce more efficient code.

To illustrate Maril, a toy processor (TOYP) is used. TOYP supports five operations: load, store, add, compare and branch. It has eight 32-bit general purpose registers that can be used in 64-bit pairs to support double precision floating point. TOYP contains a 5-stage instruction

[1] Instruction word *packing* refers to the process of fitting individual operations into an instruction word.

```
declare {
%reg r[0:7] (int);                      /* integer regs */
%reg d[0:3] (double);                   /* double float regs */
%equiv r[0] d[0];                       /* d regs overlay r regs */
%resource IF; ID; IE; IA; IW;           /* fetch; decode; execute; access mem; writeback */
%resource F1; F2; F3; F4; F5;           /* floating point add pipe */
%def const16 [−32768:32767];            /* signed immediate */
%def uconst16 [0:65535];                /* unsigned immediate */
%def nconst16 [−65535:−1];              /* negative immediate */
%label rlab [−32768:32767] +relative;   /* branch offset */
%memory m[0:2147483647];                /* memory bank */
}
```

Figure 3.2: **TOYP declarations**. The *r* registers hold integer values; the *d* registers hold double float values. Resources include the instruction pipeline and floating point add pipeline stages. The range *const16* is used by load, store and add instructions.

pipeline and a 5-stage floating point add pipeline. The source language is assumed to be similarly limited.

### 3.3.1 Declarations

The **Declare** section (Figure 3.2) describes the target machine's registers, memory, and resources, such as pipeline stages, functional units and buses; it also gives definitions for instructions' immediate operands. Items that are referenced in subsequent sections must be declared here.

Each **%reg** declaration describes an array of registers along with the datatypes that can reside in them. Maril supports the signed C Language native types. The size of each register is inferred from the size of the largest type it may hold. The **%equiv** directive indicates that two register sets are overlaid. For example, TOYP's *d* registers overlay the *r* registers; one *d* register overlays two *r* registers.

A **%resource** declaration names a processor resource. A *resource* is a pipeline stage, data bus or functional unit. A resource name has no implicit meaning or properties associated with it.

A **%def** declaration specifies an integer constant range for use as an instruction's immediate operand. A **%label** declaration is similar to a **%def**, but is used for branch offsets. Optional flags may appear on both of these declarations. The **+relative** flag indicates that the label is a relative branch offset. The **+relocatable** flag permits a **%def** pattern node to match a subject node that represents a relocatable address. The **+halfreloc** flag permits a **%def** to match a subject node that represents half of a relocatable address.

Memory banks may be declared using the **%memory** directive. An instruction indicates it references memory by using the memory object as an array indexed by the effective address.

### 3.3.2 Runtime Model

The **CWVM** section (Figure 3.3) specifies the runtime model to which the generated code must conform. Directives specify which register sets are for general purpose use (**%general**), which registers are available for allocation by the global register allocator (**%allocable**), and which are

```
cwvm {
%general (int) r;                 /* r gpr for int */
%general (double) d;              /* d gpr for double */
%allocable r[1:5];                /* r[1] through r[5] available to register allocator */
%calleesave r[4:7];               /* r[4] through r[7] saved by callee */
%sp r[7] +down;                   /* stack pointer */
%fp r[6] +down;                   /* frame pointer */
%retaddr r[1];                    /* return address */
%hard r[0] 0;                     /* r[0] always 0 */
%arg (int) r[2] 1;                /* 1st int arg in r[2]*/
%arg (int) r[3] 2;                /* 2nd int arg in r[3] */
%arg (double) d[1] 1;             /* 1st double arg in d[1] */
%result r[2] (int);               /* int result in r[2] */
%result d[1] (double);            /* double result in d[1] */
}
```

Figure 3.3: **TOYP Compiler Writer's Virtual Machine**. The *r* and *d* register sets are general purpose. Either two integer parameters or one double float parameter may be passed in registers; the result is returned in *r[2]*, if integer, or *d[1]*, if double. Registers *r[1]* through *r[5]* may be allocated by the global register allocator; *r[4]* through *r[7]* are preserved across a call.

preserved across function calls (**%calleesave**). Registers not specified as **%calleesave** are assumed to be saved by the caller. Directives also specify which registers are used for the return address (**%retaddr**), the stack pointer (**%sp**) and the frame pointer (**%fp**). With **%sp** the user specifies **+down** or **+up** to indicate the direction of stack growth. With **%fp**, **+down** indicates that frame offsets may be positive and negative; **+up** indicates that they may be only positive. **%hard** specifies that a register is hard-wired to a particular value.

Marion requires the user to specify two registers to be the stack and frame pointers. Optionally, a third register may be declared as a global data pointer (**%gp**) to point to static data. This directive includes the maximum data area size. Marion allocates a module's static objects in the area until it is filled, after which objects receive 32-bit relocatable addresses.

The **%arg** and **%result** directives specify parameter passing conventions. Each directive gives a type and a register; **%arg** also gives a parameter number. For example, the first **%arg** directive in Figure 3.3 indicates that the first integer argument is passed in *r[2]*; the **%result** directives indicate that the result is returned in *r[2]*, if integer, or *d[1]*, if double.

Marion supports only relatively simple CWVMs. Some CWVMs use a virtual frame pointer relative to the stack pointer. Others have specialized calling conventions; for example, a floating point argument may be passed in either an integer or floating point register, depending on the preceding arguments. Still other CWVMs support register windows. More complex CWVMs could be supported with additional directives (and have been in other systems [Hen87]). For example, register windows could be supported by extending Maril to allow incoming and outgoing parameters and calleesave and callersave registers to all be specified separately. The stack model is simple because this research concentrates on instruction scheduling.

### 3.3.3 Instructions

The **Instr** section (Figure 3.4) lists machine instructions, special instructions and *glue* transformations. The description for each instruction indicates its purpose and scheduling requirements. Glue transformations help complete the mapping between the IL and the target's instruction set.

Each **%instr** directive describes a single machine instruction in five parts. The first part is the instruction mnemonic and operands. An operand may be a generic member of a register set, a specific register, an immediate range that appears in a **%def** declaration or a specific immediate value. The second part (in parentheses) is an optional type constraint that is used during code selection. A constraint of the form *(type)* restricts all register operands to *type*. A constraint of the form *($i==type)* restricts operand *i* to *type*.

The third part (in braces) is a single assignment C expression (excluding side effects and conditional operators) that indicates what operation the instruction performs. Expression operands are integer constants or references to the instruction's operands, *e.g.*, $1. The patterns used by the matcher are derived from these expressions. If an instruction yields two results (or a side effect), such as a register and a condition code, then Maril semantics require the user to write a separate instruction directive for each result. This may cause two instructions to be generated where one would suffice; it also prevents the use of auto-increment addressing modes. Many RISCs do not have instruction side effects, but both the i860 and the Sun SPARC [Sun87] have condition codes and the i860 has autoincrement addressing. At the expense of occasional inefficient code, this restriction retains simplicity in deriving the patterns from the description and in matching the patterns. Peephole optimization techniques for exploiting instruction side effects are well known and could be incorporated into Marion, but no research has been done on the interaction of peephole optimizations and instruction scheduling.

The fourth part of a directive (in brackets) specifies the hardware resources needed by the instruction on each cycle after the instruction is issued. Semicolons separate cycles; commas separate resources within a cycle. For example, in Figure 3.4, the *add* instruction uses the instruction pipeline, one stage per cycle; the *fadd.d* instruction requires the instruction decode stage (ID) and the first floating point adder stage (F1) each for two cycles, using both on the third cycle.

Finally, the instruction directive includes a triple *(cost,latency,slots)* that specifies more scheduling information:

> (1) *Cost* is used only to distinguish actual instructions, which cost 1, from zero cost dummy instructions. Dummy instructions are useful for type conversions in the IL that require no actual instructions.
>
> (2) *Latency* is the number of cycles before the instruction's result can be used by another instruction. Usually the latency is the number of cycles an instruction spends in execution stages, *i.e.*, stages other than fetch, decode or writeback. The presence or absence of bypass hardware, however, can affect this relationship. Therefore, Marion requires the user to indicate the latency directly, at the risk of some redundancy.
>
> (3) *Slots* specifies the number of delay slots following the instruction that must be filled by the scheduler. It is typically used for call and branch instructions, but can also be used for other instructions, such as a delayed load whose delay slot must be filled. A positive *slots* value indicates that instructions in the delay slots are always executed. A negative value indicates that they are executed only if the branch is taken. A zero

```
%instr  add r, r, r                   (int )     /* integer add */
          {$1 = $2 + $3;}
          [IF; ID; IE; IA; IW;]        (1,1,0)
%instr  addu r, r[0], #uconst16       ($1==int ) /* manifest constant */
          {$1 = $3;}
          [IF; ID; IE; IA; IW;]        (1,1,0)
%instr  fadd.d d, d, d                           /* double float add */
          {$1 = $2 + $3;}
          [IF; ID; F1,ID; F1;F2;F3;F4;F5; IW,F5; IW;] (1,6,0)
%instr  ld r, r, #const16                        /* load */
          {$1 = m[$2+$3];}
          [IF; ID; IE; IA; IW;]        (1,3,0)
%instr  ld.d d, d, #const16                      /* double float load */
          {$1 = m[$2+$3];}
          [IF; ID; IE; IA; IW,IA; IW;] (1,4,0)
%instr  st r, r, #const16                        /* store */
          {m[$2+$3] = $1;}
          [IF; ID; IE; IA;]            (1,1,0)
%instr  st.d d, d, #const16                      /* double float store */
          {m[$2+$3] = $1;}
          [IF; ID; IE; IA; IA;]        (1,1,0)
%aux  fadd.d : st.d (1.$1==2.$1)      (7)        /* auxiliary latency for instruction pair */
%move [s_movs] add r, r, r[0]                    /* single reg move, referenced by movd */
          {$1 = $2;}
          [IF; ID; IE; IA; IW;]        (1,1,0)
%move  *movd d, d                                /* *func escape: double move (2-instrs) */
          {$1 = $2;}
          [ ]                          (0,0,0)
%instr  cmp r, r, r                   (int )     /* compare */
          {$1 = $2 :: $3;}
          [IF; ID; IE; IA; IW;]        (1,1,0)
%instr  beq0 r, #rlab                            /* branch if equal */
          {if ($1 == 0) goto $2;}
          [IF; ID; IE;]                (1,2,1)
%glue  r, r                                      /* glue transformation for compare */
          {($1 == $2) ==> (($1 :: $2) == 0);}
```

Figure 3.4: **TOYP instructions**. The (**int**) is a type constraint on the register operands. The expression (in braces) indicates the operation the instruction performs. The resource vector (in brackets) specifies the resources required by the instruction on each cycle. The triple (in parentheses) gives the cost, latency and number of delay slots. The auxiliary latency directive (**%aux**) overrides the normal latency label of a code DAG edge between two instructions, if the constraint is met. The **%move** directives indicate how to move between registers in the same set. A glue transformation expands the left-hand-side subtree into the right-hand-side subtree.

value means that they are executed only if the branch is not taken, which is the same as having no slots at all.

Sometimes operation latency depends on the interaction between two instructions. For these situations the user employs an auxiliary latency directive (**%aux**), which overrides the normal latency associated with the first instruction in the pair. For example, on TOYP, the *fadd.d* instruction usually has a 6-cycle latency. However, if the result is stored to memory then the latency is 7 cycles. The auxiliary latency directive (shown in Figure 3.4) specifies that, if the first operand of the first instruction (1.$1) is the same as the first operand in the second instruction (2.$1), then the latency label on the code DAG edge between an *fadd.d* and a *st.d* is 7.

The user must give the properties of a no-op instruction and, for each general purpose register set, a **%move** directive to indicate how to move between registers of that set. If a **%instr** directive had an expression {$1 = $2;}, then Marion's pattern matcher could loop, continuing to match this pattern. Marion could recognize such patterns and treat them specially; the **%move** directive simply puts this burden on the user. Alternatives for avoiding looping in the pattern matcher have been developed by Glanville [Gla77] and Henry [Hen84].

### 3.3.4 Mapping the IL

A target independent IL rarely maps directly onto a machine's instruction set. Marion supports two mechanisms that enable the user to complete this mapping. The first is the *glue* transformation. Each glue transformation comprises a pattern tree and a replacement tree. In a separate pass prior to code selection, the glue transformer applies each transformation to the IL in a top-down manner; if a subject IL subtree matches a pattern tree, it is replaced by the replacement tree. The glue transformation handles most cases where an IL operator does not map directly onto the target machine. It is generally employed when an IL operator can be mapped to two or three machine instructions without the complication of half-register references. Typical uses are for comparisons, conversions or loading 32-bit immediate values.

For example, on TOYP to compare and branch requires the generic compare *cmp* that puts a condition code into a general purpose register and the branch *beq0* that checks the condition in the register and branches accordingly. (See Figure 3.4.) To map the "==" IL operator onto this two-instruction sequence, a glue transformation expands the subtree ($1 == $2) into the subtree (($1 :: $2) == 0). The new operator "::" (supported by Marion's IL) will match the *cmp* instruction and "==" will match the *beq0* instruction.

The glue transformation, however, is inappropriate for two more complicated mapping situations: transformations that must manipulate register halves and transformations that contain common subexpressions. This is because glue transformations are applied before pseudo-registers are created and because they are tree-to-tree, not tree-to-DAG transformations.

The second mechanism, the **func* escape, handles these cases and, in general, allows the user to effect a complicated IL-to-target mapping without sacrificing opportunities for efficient generated code. The escape mechanism allows the user to write a C function to produce a sequence of individually schedulable instructions. A user-written function can and should comprise only calls to primitives exported by Marion; these primitives create and manipulate operands and generate instructions. A user-written function is called by Marion when the code selector matches an escape function. On TOYP, for example, a move between *d* registers maps into two moves between *r*

```
void movd (sopnds, resopnd)                                                movd
    SOPNDS sopnds;                                   /* operands: dest, src */
    int resopnd;                                     /* which operand is dest */
    {
    SOPNDS low_opnds;                                /* temporaries */
    SOPNDS high_opnds;

    opnd_low(&low_opnds[1], &sopnds[1]);             /* lowhalf(dest) */
    opnd_low(&low_opnds[2], &sopnds[2]);             /* lowhalf(src) */
    gen_instr(s_movs, 2, &low_opnds[1], &low_opnds[2]); /* move low half */
    opnd_high(&high_opnds[1], &sopnds[1]);           /* highhalf(dest) */
    opnd_high(&high_opnds[2], &sopnds[2]);           /* highhalf(src) */
    gen_instr(s_movs, 2, &high_opnds[1], &high_opnds[2]); /* move high half */
    }
```

Figure 3.5: **Example escape function**. User-written code for the **movd* escape function. Each *opnd_low* or *opnd_high* creates an operand that represents the low or high half of a $d$ register. Each *gen_instr* references the $r$ register move via the label *s_movs* and generates a move instruction.

registers. The user-written function for **movd*, shown in Figure 3.5, creates operands to represent the two halves of each $d$ register and generates two move instructions between $r$ registers; after code selection, each move instruction can be scheduled individually. The move between $r$ registers is referenced by the optional label *[s_movs]* that appears in the directive. (See Figure 3.4).

Maril includes several built-in functions for use in instruction expressions and transformations. The built-in functions *high* and *low* are typically used to split a 32-bit immediate into 16-bit halves. The built-in *eval* evaluates a constant expression, typically to negate or decrement an immediate value; it may appear only within glue transformations. Datatypes are used as built-ins to match type conversions in the IL.

Marion supports two additional IL operators. First, the generic compare special operator "::", which is useful for machines that separate compare and branch into two instructions, can be introduced by a glue transformation. Second, the explicit arithmetic right shift ">>>" matches the corresponding IL operator unambiguously. Figure 3.6 shows glue transformations that use some of these built-in functions.

Figure 3.7 presents a sample code fragment for TOYP.

## 3.4 Code Generator Generator

Marion's code generator generator takes a machine description as input and produces a set of target-dependent tables and routines that is used by the code generator for code selection, instruction scheduling and register allocation. The most important table is an array that contains instruction information. Each array element corresponds to an instruction directive given in the description and contains the following: a pattern tree and a replacement symbol derived from the expression given in the directive; an array that indicates, for each instruction, its operand kinds (*e.g.*register) and their locations within the pattern and subject trees; an index into an array of resource vectors; and cost, latency and delay slot data. Other data structures hold declarations, CWVM information,

```
%instr  oru r, r, #uconst16              (char|short|int)
        {$1 = $2 | ($3 << 16);}          /* load constant upper */
        [IF; ID; IE; IA; IW;]            (1,1,0)

/* for loading 32-bit constant */
%glue  #const32                          (int)
        {$1 ==> (low($1) | (high($1) << 16));}

/* for adding negative constant */
%glue  r, #nconst16                      (char|short|int)
        {($1 + $2) ==> ($1 - eval(-$2));}
```

Figure 3.6: **Glue transformations**. An instruction directive and two glue transformations. The first transformation splits a 32-bit constant into halves; it is needed for most RISCs, which do not have 32-bit immediates. The *low* and *high* built-ins take the low and high halves of the constant, respectively. The *oru* instruction matches the subtree created by the transformation that represents the high half. The second transformation is useful for loading a negative constant in one instruction on a machine that zero-extends immediate operands. The built-in *eval* evaluates a constant expression.

C code fragment

```
register double x;
double y;
int a;

a = 0xFFFFF;
x = y;
```

(a)

TOYP assembler fragment

```
                        ; x is allocated register d1
addu   r4,r0,0xFFFF     ; low half manifest constant
oru    r5,r4,0xF        ; high half manifest constant
st     r5,8(fp)         ; store a
ld.d   d2,16(fp)        ; load y
add    r2,r4,r0         ; high(d1) gets high(d2)
add    r1,r3,r0         ; low(d1) gets low(d2)
```

(b)

Figure 3.7: **Sample TOYP code**. Given the C fragment in (a), the unscheduled TOYP code is shown in (b). The 32-bit constant takes two instructions to manifest. The double move also takes two instructions.

classes and elements (described in Chapter 4), auxiliary latencies, escape function addresses and glue transformation patterns. The instruction and glue transformation pattern trees are created at compile time by a target-dependent initialization routine.

The two most significant target-dependent routines are *maptype_decl*, which returns the general purpose register set for the given type, and *reg_overlap*, which returns register overlapping information derived from **%equiv** directives and register sizes. These are used during pseudo-register creation and register allocation.

The code generator generator uses a recursive descent parser to build lists of information that directly correspond to the major Maril constructs, such as declarations and instructions; the lists are then processed and output as initialized data.

Since most instructions use machine resources in the same manner, resource use vectors are shared among instructions. The code generator generator enters each unique vector it encounters in a resource vector table that is output with the other information lists.

Because pattern trees are allowed to match subject trees of various datatypes, the pattern trees are typed bottom-up with a union of types called a *multitype*. The multitype indicates all possible types that a pattern node may match. Since the IL trees presented to the matcher are guaranteed to be correctly typed, the multitype cannot cause an illegal match.

## 3.5 Code Selection

Marion performs code selection before control is given to the code generation strategy. The selector matches the input program's IL using a recursive-descent brute-force tree pattern matcher. Patterns are derived from the machine description by the code generator generator. The matcher examines the patterns in the order given, greedily selecting the first one that matches, and then attempts to match the subtrees. If unsuccessful, it proceeds to the next pattern. If successful, the pattern is attached to the IL node. Figure 3.8 gives the basic matching algorithm. After matching, Marion does a left-to-right bottom-up walk of the IL, generating the code stream according to the patterns attached to the IL nodes.

For example, to match the subject tree shown in Figure 3.9, given the set of patterns in Figure 3.10, the top-down matcher begins with the start symbol "STMT" and examines the pattern "= Addr REG." Since the top-level operator in the subject matches the "=" operator in the pattern, the matcher examines the children, beginning with the left child. The pattern tree's left child "Addr" matches the subject tree's left child "Addr a." The pattern tree's right child is the non-terminal "REG," so the matcher recurses and examines patterns whose replacement symbols are "REG" against the new subject tree rooted at the "+" node. The first symbol in the pattern "+ REG REG" matches the new subject tree's top-level operator, so the matcher examines the children. Since the pattern tree's left child is "REG," the matcher again recurses. The first pattern it examines fails because the operators do not match. The first symbol in the next pattern "↑ Addr" matches the subject tree's top-level operator and the children match as well. At this point the matcher returns *success* and then attempts to match the other subtree. This subtree matches in similar fashion; therefore the entire tree matches.

Two tables are created by the code generator generator to effect more efficient matching. The first, the NTindex table, contains triples *(nt,first,last)*; each maps a replacement symbol *nt* into a

```
Algorithm 1: match
    Input: threaded IL DAG
    Output: attributed threaded IL DAG
    Define: foreach IL subject or pattern node s
        s.op = IL operator; s.types = set of allowable types (multitype)
        s.pat = pattern matched by a subject node
    Define:
        NT = the set of non-terminal replacement symbols
        N.pat = the set of patterns associated with a non-terminal symbol N
    Method:
        /*** Match all statements in procedure ***/
        foreach node s ∈ IL thread
            if matchNT(STMT,s) = true then
                success
            else
                error
            endif
        endfor

    matchNT(N, s)
        /*** Match a non-terminal ***/
        foreach pattern p ∈ N.pat
            if matchpat(p, s) = true then
                return true
            endif
        endfor
        return false

    matchpat(p, s)
        /*** Match a pattern ***/
        if p.op = s.op and s.types ⊂ p.types then
            foreach p.child
                if matchpat(p.child, s.child) = false then
                    return false
                endif
            endfor
            s.pat = p
            return true
        elseif p.op matches some symbol N then
            if matchNT(N, s) = true then
                return true
            endif
        endif
        return false
```

Figure 3.8: **Algorithm for matching the IL.** STMT is the non-terminal start symbol in the grammar derived from the machine description.

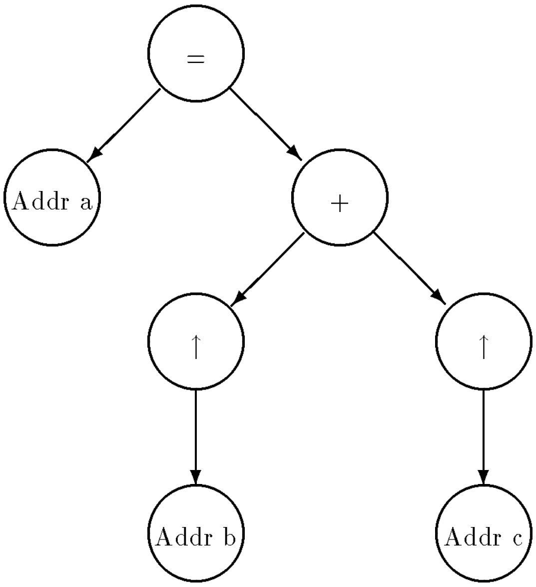



Figure 3.9: **Example IL subject tree** for the statement $a = b + c$. "↑" is an operator that extracts the value at an address.

| Replacement | Pattern | Instruction |
|---|---|---|
| STMT | = Addr REG | st addr,r |
| REG | + REG REG | add r1,r2,r3 |
| REG | ↑ Addr | ld addr,r |

Figure 3.10: **Tree patterns**. Table of tree patterns (in prefix form), replacement symbols and corresponding instructions. STMT is the non-terminal start symbol. REG is a non-terminal that represents a register. "↑" is an operator that extracts the value at an address. Addr is a terminal symbol (leaf) that represents an address.

range in the second table, the OPindex table. The OPindex table contains triples *(op,first,last)*; each maps an IL operator *op* into a range in the instruction pattern array. At any point during matching, the top-down matcher has a replacement symbol (from the containing pattern) and a subject tree. The matcher uses the replacement symbol to index into the NTindex table, yielding a range in the OPindex table. The matcher uses the top-level operator of the subject tree to index into this range, yielding a range of instructions to examine. All instructions in this range have the replacement symbol *nt* and a pattern tree whose top-level operator is *op*. Figure 3.11 shows an example.

During code selection *pseudo-registers* are created for all expression temporaries. Local common subexpressions and user variables that may reside in registers are also represented by pseudo-registers. Pseudo-registers are later mapped to physical registers by the register allocator. *Local*

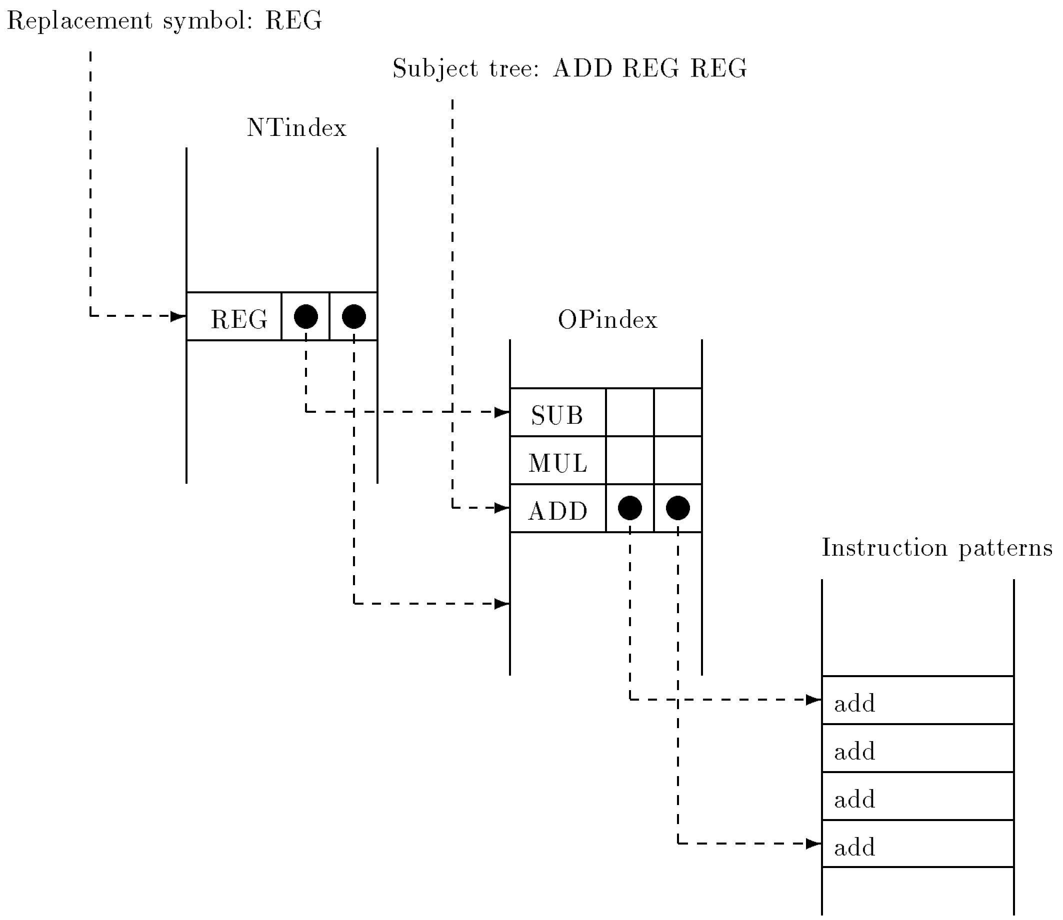



Figure 3.11: **Pattern matcher tables**. An example use of the NTindex and OPindex tables by the pattern matcher. Assume the current replacement symbol is REG and the subject tree is "ADD REG REG" (prefix notation). The replacement symbol REG indexes into NTindex, which gives a range into OPindex. The top-level subject tree operator ADD indexes into the OPindex range, which yields a range of instruction patterns. The pattern matcher examines the patterns in order, attempting to match the subject tree.

pseudo-registers refer to registers that are live in only one basic block. Registers that are live in more than one block are known as *global* pseudo-registers.

A simple code selector that is separate from other code generation phases can yield inefficient code for CISCs. This is because most operations may be performed directly on memory operands and because many IL constructs can be mapped to more than one instruction sequence. On RISCs, however, operands may not reside in memory and most IL constructs can be mapped to only one instruction sequence. Therefore, since the code selector's choices are limited, a simple scheme that is separate from register allocation causes few inefficiences.[2] Addressing expressions do require the selector to make choices, but the choices are simple enough to be managed with a hand-ordered

[2] On some RISCs, notably the i860, there is a greater potential for inefficiency, because the code selector has more instruction sequence choices. Nevertheless, there are still significantly fewer than on most CISCs.

pattern list, using the greedy heuristic.

The input to the code selector is an IL DAG. To accommodate DAGs within a naive tree pattern matcher, an IL node with more than one parent is forced into a register, unless it is a constant that can be subsumed by an addressing mode or an immediate operand.

## 3.6 Code Generator Structure

Marion's modular structure allows it to support different code generation strategies. The strategy module directs post-selection code generation, as shown in Figure 3.12. It is invoked once per source procedure and calls the strategy-independent routines as appropriate. The strategy-independent portion of the back end has three logical components: (1) the code DAG builder constructs a DAG of machine instructions for each basic block (see Section 4.1); (2) the global register allocator determines how physical registers are used; and (3) scheduling support handles low-level scheduling details, such as checking for resource conflicts and maintaining the list of instructions that can be scheduled without causing a delay. The scheduling algorithm is strategy-dependent. Because the strategy is isolated from the rest of the code generator, the strategy can be replaced with a minimum of effort. Quick reconfiguration is useful both to determine the best strategy for a target architecture and to experiment with new strategies. The strategies are discussed in Chapter 6.

## 3.7 Register Allocation

Marion uses a graph coloring global register allocator based on the work of Chaitin [Cha82] and Briggs *et al.* [BCKT89]. Graph coloring models the register allocation problem as an interference graph in which nodes represent pseudo-registers and edges represent interference between them. Two pseudo-registers *interfere* if there is some point in the program at which they are both live. Two pseudo-registers interfere if their lifetimes overlap, meaning they cannot share the same register. By assigning different colors to interfering nodes, coloring the graph is analogous to assigning physical registers to pseudo-registers. The register allocator determines interference using the instruction order that is presented to it.

Chaitin's approach is widely used, produces good results and is general enough to support different code generation strategies. Its disadvantage is that a pseudo-register is either assigned a physical register or is spilled for its entire lifetime. Chow and Hennessy's method [CH84], which "splits" a pseudo-register's lifetime, may be a profitable alternative. The next two sections describe Chaitin-style and Chow-style register allocation in more detail.

### 3.7.1 Chaitin-style Register Allocation

In a Chaitin-style register allocator [CAC+81, Cha82], all expression temporaries, common subexpressions and user variables that may legally reside in registers are given pseudo-registers during code selection. The register allocator considers *all* pseudo-registers, both global and local; it allocates physical registers to some of them and spills the rest to memory, inserting spill code (load and store instructions) where appropriate. In Chaitin's allocator as well as with the modifications of Briggs *et al.* [BCKT89] and Bernstein *et al.* [BGM+89], a pseudo-register either gets a physical register or is spilled to memory for its entire lifetime.

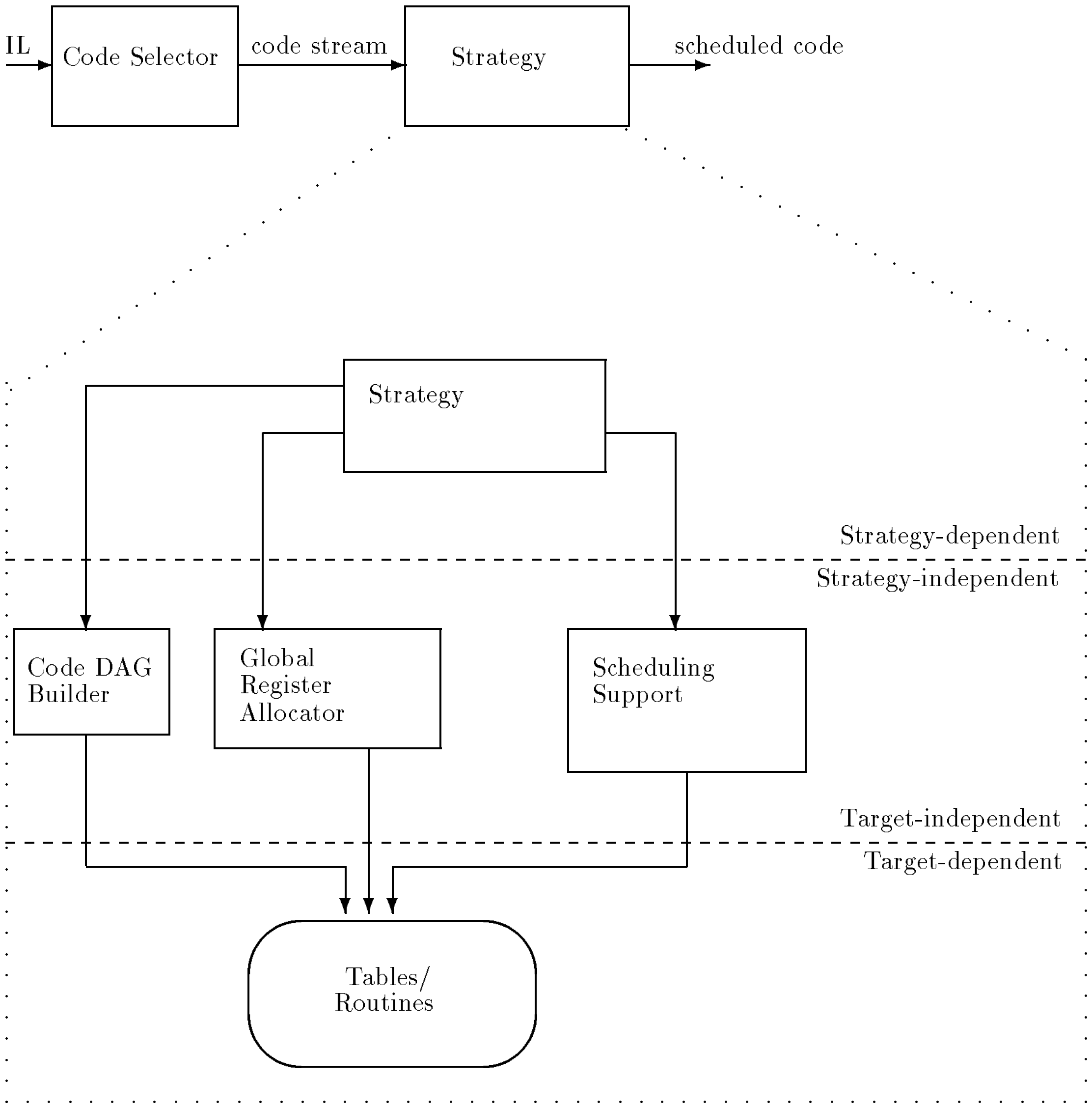



Figure 3.12: **Code generator structure**. (The symbol → means "calls" or "accesses.")

The register allocator comprises four phases: interference graph building, coalescing (or subsumption), coloring and spilling. Chaitin's method iterates over the graph building, coalescing and spilling phases until it determines that the graph is colorable. Briggs's method, which Marion employs, also attempts to color the graph on each iteration.

The coalescing phase combines two pseudo-registers if all of the following are true: (1) there is a move between the last use of one and the definition of the other; (2) the two do not interfere; and (3) the combined pseudo-register node is unconstrained. A pseudo-register node is *unconstrained* if it has fewer neighbors in the interference graph than allocable physical registers. Unconstrained nodes can always be colored.

Constrained nodes are colored in decreasing order of priority; then unconstrained nodes are colored. For each node $p$,

$$priority_p = \frac{regcost_p}{degree_p}. \tag{3.1}$$

$Degree_p$ is the number of neighbors of $p$ in the interference graph. $Regcost_p$ is the cost (in machine cycles) of forcing $p$ into memory. Generally, this is the sum of the cost of each memory reference weighted according to the frequency of use of the reference's block. Specifically, let $p$ be a pseudo-register; then

$$regcost_p = \sum_{r \in refs_p} opcost_r * freq_b, \tag{3.2}$$

where

(1) $opcost_r$ is the number of instructions needed to do a load or store operation;[3]

(2) $b$ is the basic block containing the reference $r$;

(3) $freq_b$ is the number of times $b$ is executed (as calculated by a profiling tool) and

(4) $refs_p$ is the set of all references to $p$.

$Regcost_p$ is divided by $degree_p$ (Equation 3.1) to reflect the fact that if $p$ is assigned a physical register, then that physical register cannot be assigned to any of $p$'s neighbors.

If a pseudo-register cannot be colored, then it is spilled. Each use of the register is preceded by a load and each definition is followed by a store, except that multiple uses of a global pseudo-register within a block are bracketed by only one load and one store. New short-lived pseudo-registers are created to hold the values between loads and uses and between stores and definitions. Therefore, if any spilling occurs, a new interference graph must be constructed; coalescing and coloring are then repeated.

### Marion Register Allocation

Marion register allocation uses Chaitin's method [Cha82] with the modifications of Briggs *et al.* [BCKT89], except that the code generation strategy determines how the register allocator is applied

[3] $Opcost_r$ is 2 for double precision pseudo-registers and 1 for all others. It does not consider load latency, but could be modified to do so.

to local and global pseudo-registers; they may be considered together, which is the standard method, or separately.

Chaitin does not specifically discuss register pairs. In Marion's register allocator nodes in the interference graph that represent register pairs must be colored with two adjacent colors (which correspond to adjacent physical registers); the coloring phase enforces this using a first-fit policy. (It currently assumes even/odd pairs.) The presence of register pairs changes the definition of an *unconstrained* node. A node $p$ is *unconstrained*, if $degree_p * need_p$ is less than the number of allocable registers. $Need_p$ is the number of physical registers required by pseudo-register $p$ (2 for a register pair). The new definition of $degree_p$ is as follows:

$$degree_p = \sum_{q \in neighbors_p} need_q. \tag{3.3}$$

Thus, the degree of a node is the sum of the physical register requirements of its neighbors. The new definition of *unconstrained* considers the case where a register pair cannot be colored because of physical register set fragmentation. Marion's coloring policies reduce fragmentation by packing together pseudo-registers that need single physical registers, but fragmentation can still occur.

Register calling conventions indicate which physical registers are calleesave (preserved by the called procedure), which are callersave (not preserved by the called procedure) and which are used to pass parameters. Marion includes callersave physical registers in the interference graph and makes them interfere with pseudo-registers that are live across a procedure call. This prevents callersave registers from being assigned to those pseudo-registers, as Chaitin recommends.

For each calleesave register, Marion inserts a move instruction in the procedure's prolog[4] from the calleesave register to a new pseudo-register; a move from the pseudo-register back to the calleesave register is inserted in the epilog. The register allocator treats this pseudo-register as it treats all other pseudo-registers. If possible, it is coalesced with the calleesave physical register, thereby eliminating the moves. If the pseudo-register cannot be colored, it is spilled, which in effect saves the calleesave register on the stack. Chow's *shrink-wrapping* [Cho88] may be a useful addition to this scheme.

At a call site, an argument that is passed in a register is moved explicitly into that physical register. Again, the register allocator relies on coalescing to eliminate unnecessary moves.

### 3.7.2 Chow-style Register Allocation

Chow and Hennessy's method [CH84, CH90] differs from Chaitin's in that the register allocator can split a candidate's lifetime. In Chow's original method, global register allocation is performed on a subset of the available physical registers before code selection. The remaining registers are used for local expression temporaries. Larus and Hilfinger's variation [LH86] performs register allocation after code selection. This section discusses the original method, Larus and Hilfinger's variation and describes how a Chow-style allocator could be fit into Marion.

[4]The *prolog* is the sequence of instructions at the beginning of a procedure that precedes the user's code. The *epilog* is the sequence at the end of a procedure that follows the user's code. The prolog and epilog contain code to allocate and deallocate the procedure's activation record and to enforce the calling conventions.

To perform register allocation before code selection, Chow's allocator divides the register set into two fixed-size partitions, one for global register allocation and one for expression temporaries created by the code selector. Each register allocation candidate, including user variables and optimizer-generated temporaries, has a "home" stack location, where it will reside if does not get a physical register. The interference graph is constructed using the basic block as the grain-size. This means that two candidates interfere if there is some block in which both are live, even though one may be live only at the beginning of the block and the other live only at the end. In contrast, Chaitin-style allocators use a finer grain-size, the machine instruction. Coalescing is not applicable in a Chow-style allocator, because machine instructions have not yet been generated.

Like Chaitin-style allocators, Chow's allocator computes a priority for each candidate node; constrained and unconstrained nodes are also distinguished. The colorer iterates until no more nodes can be colored. On each iteration, the highest priority constrained node is colored, each of its neighbors is checked to see if the neighbor should be split, and uncolorable nodes are removed from the interference graph. If splitting occurs, the graph is updated and the constrained and unconstrained sets are recalculated. After coloring, the IL is modified to reflect the locations of the candidates that were colored.

Lifetime splitting is one of the main advantages of Chow's allocator; it allows the allocator the freedom to assign a register to an important *part* of a candidate's lifetime, instead of all or none of it. A second advantage is compile-time efficiency: Chow's allocators consider fewer candidates, because local expression temporaries are left to the the code selector, and the coloring process occurs only once with incremental updates of the interference graph; Chaitin-style allocators iterate over the graph construction and coloring phases.

The primary disadvantage of Chow's allocator is that it partitions the register set, which eliminates the possibility of using registers for global candidates sometimes and for locals other times. Another disadvantage is the larger grain-size, which increases the number of global pseudo-registers that interfere.

Larus and Hilfinger's register allocator [LH86] eliminates register set partitioning by allocating registers after code selection. It is more like Chaitin's allocator, in that all allocation candidates are represented in the code by pseudo-registers. Also, all pseudo-registers (global and local) are placed into the interference graph together. Like Chow's method, however, the grain-size is a basic block. Chow's coloring algorithm, including lifetime splitting, is used to color the graph. Spill code is inserted for candidates that must reside in memory or whose lifetimes are split. Two registers are reserved for use in spill code; this avoids the need for repeating the coloring.

Because Marion allows the code generation strategy to direct register allocation and instruction scheduling, Chow's original method, which performs allocation before code selection, is inappropriate. Larus and Hilfinger's method is compatible with Marion, but the basic block grain-size for interference would be too coarse for workloads that contain large basic blocks.

## 3.8 Summary

This chapter has discussed the machine description language Maril, along with the major components of the Marion System, including register allocation, code selection and the intermediate language. Maril's innovation is the incorporation of instruction scheduling information in a machine description language for code generation. Maril has constructs, such as auxiliary latencies and escape functions, to handle idiosyncrasies that are found even on RISCs. The next chapter continues with an examination of Marion's instruction scheduling.

# Chapter 4

# Instruction Scheduling

Many basic block instruction scheduling algorithms have been used for specific RISCs and microcoded processors. Marion's instruction scheduling infrastructure supports many of the algorithms, which are themselves target-independent. In addition, the data structures that Marion employs are target-independent, but contain target specific information that is derived from the machine description.

Recall that the code generation strategy directs the invocation of and degree of communication between instruction scheduling and register allocation. The strategy also includes the scheduling algorithm, so that the algorithm can be easily replaced. This permits the compiler writer to experiment with different strategies and scheduling algorithms to find the appropriate ones for the target machine and expected workload.

This chapter first describes the primary scheduling data structure, the scheduling algorithms that are supported and Marion's structural hazard checking. Then it describes algorithms to schedule explicitly advanced pipelines and proves them correct. Finally, it discusses scheduling for the Intel i860, which has explicitly advanced pipelines.

## 4.1 Code DAG

The primary data structure that supports basic block instruction scheduling is the *code DAG*, in which nodes represent instructions and directed labeled edges represent dependences between instructions. The label represents the latency between the edge's source and destination nodes. The DAG is threaded by a *code thread*, which forms a topological sort and represents the initial order of the basic block's instructions. A *root* of the DAG is a node that depends on no others (*i.e.*, it has no predecessors). A *leaf* is a node on which no other nodes depend (*i.e.*, it has no successors). An edge $(x, y)$ with label $l$ means that $y$ cannot be scheduled fewer than $l$ cycles after $x$ without either violating the program's semantics or causing a data hazard, a pipeline delay that occurs when data is not yet available. Edges are of three types.[1] A type 1 edge $(x, y)$ represents a true dependence[2] and is introduced if one of $y$'s source operands is $x$'s result. The edge label is $x$'s latency. Type 2 edges ensure that memory references are ordered properly; they can be true, anti- or output dependences. A type 2 edge is introduced between two memory references, one of which is a store, that may reference the same location. (Section 3.2 discusses Marion's memory reference disambiguation algorithm.) Type 3 edges, which are anti-dependences, are used by some

[1] Edge types are distinguished only in the way they are created, not by the schedulers.

[2] A *true* dependence (also called a *flow* dependence) is an edge from a definition to a use. An *anti*-dependence is an edge from a use to a definition. An *output* dependence is an edge between two definitions.

code generation strategies to ensure that separate uses of the same physical register do not overlap. A type 3 edge is introduced between uses and a subsequent definition of a physical register. Type 2 and type 3 edges are given latency labels of 1. The label's value is actually target-dependent, but a value of 1 is sufficient to generate correct schedules for all the architectures considered in this research.

The strategy module controls the inclusion of each edge type, permitting Marion to support strategies that do not require all types; for example, Hennessy and Gross's method [HG83] does not require type 3 edges.

The IL DAG, which is output by the front end, provides an initial template for the code DAG. The code selector matches the IL and creates a corresponding sequence of machine instructions, but does not create the code DAG. Marion separates code DAG construction from code selection, because (1) only type 1 edges correspond to edges in the IL DAG and (2) the code DAG is difficult to keep up-to-date when spill code is inserted by the register allocator.

## 4.2 Scheduling Algorithms

Many processors, such as the CDC 6600 and the IBM 360/90, have performed instruction scheduling dynamically in the hardware [Tom67, Tho70]. For microcoded processors, microcode programmers have employed both hand-coded and compiler-based scheduling (or compaction) techniques. More recently, RISC compilers have used instruction scheduling techniques on machine code.

Except in limited cases, optimal instruction scheduling, even without resource or register constraints, is NP-complete. Consequently, heuristics are typically employed. Lawler *et al.* [LLM+87] give a survey of theoretical results. Bernstein and Gertner present a polynomial-time algorithm for the case where a single pipeline has a maximal delay of one cycle [BG89]. Proebsting and Fischer present a linear-time algorithm that minimizes execution time and local register use, given an expression tree and assuming a machine with a 2-cycle load latency and no other latencies greater than 1 [PF91].

The most common approach to instruction scheduling is *list scheduling* [Tho67, LDSM80, Fis81, HG83, FERN84, GM86, GH88, War90]. Given a code DAG, the scheduler keeps a list of instructions that are ready to be scheduled without causing a delay. On each iteration it uses a heuristic to choose a ready instruction to schedule and then updates the list. In the general case, this approach has worst-case running time of $\mathcal{O}(e)$, where $e$ is the number of edges in the DAG, but the heuristic can increase the complexity.

A list scheduler typically selects the highest priority node in the ready list. A common heuristic for assigning priority is *maximum distance*, the length of the longest path through the code DAG from the instruction node to a leaf node. The length of a path is the sum of all edge labels along the path.[3] The reasoning is that the node farthest from completion is the most critical; less important nodes can be scheduled later. Maximum distance is a good initial heuristic, but there are cases where additional or alternative heuristics are profitable.

For example, given the DAG in Figure 4.1, the schedule "$b; a; c; d; e;$" is optimal. However, using only the maximum distance heuristic, the inferior schedule "$a; c; b; \_; d; e;$" is equally likely, because $a$, $b$ and $c$ have the same priority. The delay "$\_;$" occurs because $b$ has a latency of 2. A second-level

[3] The maximum distance is sometimes called the *height* of the node within the DAG.

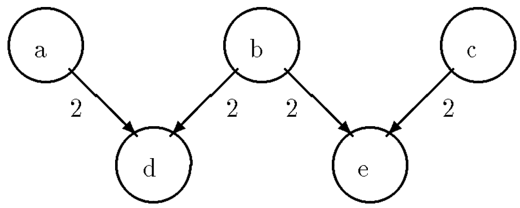



Figure 4.1: **Example DAG**. Nodes represent instructions. Edges represent dependence. Edges are labeled with latencies. The scheduler can schedule either $a$, $b$ or $c$ first; $b$ is the correct choice, because both $d$ and $e$ depend on $b$.

heuristic that gives higher priority to nodes with more successors would choose correctly in this case. The idea behind this heuristic is that scheduling a node with more successors creates more choices for the scheduler on subsequent cycles and allows more nodes to become ready sooner.

Another heuristic, used by Gibbons and Muchnick [GM86], gives a higher priority to a node that has a greater operation latency with one of its successors. By scheduling such a node first, there will be more opportunities to hide the latency with other instructions. Gibbons and Muchnick use the number of successors and the maximum distance as second- and third-level heuristics, respectively.

An alternative to having a second-level priority is Palem and Simons's Rank algorithm [PS90]. This algorithm incorporates maximum distance and the number of successors into a single "rank." First, the algorithm sets the rank of all leaves to some large number (*e.g.*, size of the code DAG plus total of all edge labels). Decreasing ranks are assigned from leaves to roots. To assign node $i$'s rank, each successor is fit into a tentative schedule based on its rank; then, $i$ is given a rank that reflects its position in the schedule relative to its successors. For example, consider the DAG in Figure 4.1. The algorithm would assign $d$ and $e$ a rank of 13. Node $a$ gets a rank of 11, since $d$'s rank is 13 and the latency between $a$ and $d$ is 2. Node $c$ also gets a rank of 11. To assign $b$'s rank, the algorithm creates a schedule containing $d$ and $e$: $d$ at cycle 13 and $e$ at cycle 12. Now, $b$ gets a rank of 10, since it must be scheduled at least 2 cycles before $e$. Thus, $b$ has a higher rank than $a$ or $c$ and, therefore, is scheduled first.

Some list schedulers update priorities during the scheduling process. For example, Warren's scheduler for the IBM RS/6000 [War90] runs before register allocation. To restrain register needs, the scheduler reduces the priorities of load instructions as the number of live registers needed by the schedule increases.

Algorithms other than list scheduling have been used as well. The *critical path* algorithm, described by Landskov *et al.* [LDSM80], schedules all nodes along the longest path from a root of the code DAG to a leaf. Then, other paths are fit around that schedule, stretching it if necessary. The *branch and bound* algorithm, also described by Landskov *et al.*, builds a tree of possible schedules. Each leaf is a complete schedule for the code DAG. An exhaustive search of the tree will find the optimal schedule, but takes exponential time. The search is heuristically bounded by noting the length of the best incomplete path so far and pruning the tree of branches that represent apparently worse incomplete paths. The Bulldog compiler [FERN84] also uses a branch and bound algorithm as an alternative to list scheduling.

A radically different approach by Arya [Ary85] uses integer programming to produce optimal schedules for the Cray-1, albeit exponentially in the number of nodes. Scheduling constraints, including operation latencies and resource requirements, are modeled as inequalities. These are fed to an integer programming package, which finds the lowest cost solution.

All of Marion's code generation strategies use list scheduling algorithms with the maximum distance as the primary heuristic. With minor additions to the scheduling support routines, other scheduling algorithms, such as critical path and branch and bound could be employed.

There may be some interaction between the architecture and scheduling algorithms. For example, for one architecture a particular algorithm may produce the best schedules, whereas a different algorithm is the best for another architecture. Also, a sophisticated algorithm may yield a high benefit on one architecture, but little on another. To date, I have not explored these possibilities beyond contemplating them.

## 4.3 Structural Hazards

A structural hazard occurs when two instructions need the same resource on the same cycle. For example, two instructions executing in separate functional units may require a register write-back bus on the same cycle, or an instruction may need a pipeline stage for more than one cycle, preventing its use by a subsequent instruction. Avoiding these hazards can yield better code.

As discussed in Section 3.3.3, the machine description indicates each instruction's resource needs. From this information Marion constructs a *resource vector* for each instruction. Each element of the resource vector contains all resources needed on a particular cycle. To avoid structural hazards, the scheduler compares a candidate instruction's resource vector with a resource vector that is the composite of the resources required by all currently executing instructions; if the intersection is non-empty, the candidate is not scheduled.

This method is computationally simple and easy to implement, but it restricts the scheduler's knowledge of structural hazards to the current cycle. Information is available to check future hazards; one possibility involves adding special edges to the code DAG to indicate where hazards would result. However, the additional processing time would be high and the likely benefit low.

Marion's structural hazard detection method cannot handle some complicated hazards. For example, the 88000 employs a priority scheme to regulate the use of its register write-back bus. (See Section 2.2.) Integer instructions have the highest priority, followed by floating point instructions and then loads. To accommodate this feature I considered the following:

(1) Allow a priority range to be associated with a resource declaration in the machine description.

(2) Allow an element of a resource vector associated with an instruction to indicate its priority.

(3) Examine priorities when checking for structural hazards and allow schedules to contain structural hazards, if they are caused by prioritized resources.

The main problem is that the cycle on which an instruction uses a resource can shift, causing other resource uses to shift as well. One solution is to have Marion keep individual resource vectors for all currently executing instructions, instead of a composite resource vector. However, checking for structural hazards, which is already time-consuming, would become considerably more expensive.

```
%instr  adds r, r, r
        {$3 = $1 + $2;}
        [ALU; WB]              (1,1,0)
%move   fmov.s f, f            (float)
        {$2 = $1;}
        [FALU; FWB]            (1,1,0)
```

Figure 4.2: **Maril directives for i860 integer add and floating point move instructions.**

Since Marion cannot predict when a load request will actually be serviced by the memory system, I concluded that the potential benefit would not justify the expense. Instead, Marion's view is that the first instruction scheduled gets the resource. This sometimes yields less efficient code, but not incorrect code.

Resources can be used to control multiple instruction issue for some superscalars. The scheduler will schedule as many instructions on a cycle as possible as long as they cause no structural hazards. For example, the i860 allows an integer and a floating point instruction to be issued simultaneously. By using a different set of resources for each of the two instruction types, the user can model two instructions per cycle. Figure 4.2 shows integer add and floating point move instruction directives.[4] Since the resources they use differ, one of each can be scheduled per cycle.

## 4.4 Control Hazards

A control hazard is a pipeline delay caused by a branch instruction if the branch target address is not known when the target instruction needs to be fetched. Most RISCs attempt to avoid these hazards by using a branch delay mechanism. (See Chapter 2.) In Maril, the number of delay slots is specified in the instruction directive. Currently, Marion always fills branch delay slots with no-ops. Gross and Hennessy's algorithm for filling these slots [GH82, Gro83] could easily be included in Marion. This algorithm runs as a separate intra-procedural pass after instruction scheduling. It attempts to fill delay slots with instructions that are before the branch, with instructions that follow the branch target, and with instructions that follow the branch. Gross and Hennessy found that, on a machine whose branches have one delay slot that is always executed, their algorithm filled 90% of the delay slots (counted statically). The effect on overall performance of filling branch delay slots varies with the workload; it will be lower for computation-intensive programs than for decision-intensive programs.

## 4.5 Classes and Clocks

The code DAG and resource vectors provide enough information to schedule instructions for many RISCs, including the R2000 and 88000. In addition, multiple instruction issue can be modeled using machine resources. Some architectures, however, restrict multiple instruction issue or instruction

[4]The actual resources used in the i860 description differ from what is shown, because of explicitly advanced pipelines, discussed in Section 4.5.

word packing,[5] or have features that cannot be expressed using the Maril constructs introduced so far.

The i860 has several such features. First, it supports instructions to perform floating point multiply and add together, but the restrictions on packing these operations together are irregular; not all combinations are legal. Second, it has *chaining* options that feed the results of one floating point pipeline into another. Third, the floating point add and multiply pipelines must be explicitly advanced. An *explicitly advanced pipeline* (EAP) is a pipeline that retains its state until one of a particular set of instructions is executed. An example is a multiply pipeline that advances only when a multiply instruction is issued; after a multiply has been "launched," the pipeline remains in a suspended state until the next multiply is issued.

To model the i860 floating point unit, Marion views it as a "long instruction word" in which each field corresponds to one of the following: three multiplier stages called M1, M2 and M3, three adder stages called A1, A2 and A3, and the floating point write-back bus FWB. The individual pipestage sub-operations that compose a full operation are declared as instructions, as shown in Figure 4.3(b). For example, to perform $d6 \leftarrow d4 * d5$, the code selector produces the sequence,

M1 $d4, d5$;   M2;   M3;   FWB $d6$.

M1 launches the multiply, M2 and M3 advance the pipeline and FWB catches the result. These sub-operations need not be scheduled on consecutive cycles, because the pipeline is explicitly advanced. In addition, since each sub-operation uses only the resource corresponding to one field, it may be packed with others to form a long instruction word.

Two Maril features support the scheduling of sub-operations. First, to handle irregular packing restrictions, the user associates a *class* with a sub-operation. The class is the set of all long instruction words in which that sub-operation may appear. The long instruction words are class *elements*. Two sub-operations may be packed if the intersection of their associated classes is non-empty. The scheduler checks for legality as each sub-operation is packed. For example, the i860 floating point multiply instruction *pfmul* is a class element. Classes associated with M1, M2, M3 and FWB all contain *pfmul*. Therefore, M1 and M3, for example, may be packed together, yielding a *pfmul* instruction.

Second, to allow EAP sub-operations to be separated in the schedule, the user defines a *clock* to keep track of time in a particular EAP. A *temporal register* based on clock $k$ is a register whose value changes when clock $k$ ticks. Temporal registers are typically latches between EAP pipestages. Instructions that advance an EAP, and therefore change the values in its latches, are declared to *affect* the EAP's clock. As shown in Figure 4.3, the i860 multiply pipeline has three temporal registers, $m1$, $m2$ and $m3$, based on clock *clk_m*; each multiply sub-operation affects *clk_m*. Section 4.8 discusses scheduling for the i860 in more detail.

Pipelines that advance on every cycle need not be modeled at the latch level, because the sub-operations performed in the stages may not be temporally separated. Within an EAP, however, the sub-operations may be separated. Doing so sometimes enables an operation to begin earlier than would otherwise be possible; it also facilitates the packing of instructions from more than one pipeline. Separating sub-operations requires the scheduler to keep track of the sub-operation results in temporal registers, a process that is called *temporal scheduling*.

[5]Recall that instruction word *packing* refers to the process of fitting individual operations into a long instruction word.

```
%clock  clk_m;                                        /* multiple pipeline clock */
%reg  m1 (double; clk_m) +temporal;                   /* stage 1 exit latch */
%reg  m2 (double; clk_m) +temporal;                   /* stage 2 exit latch */
%reg  m3 (double; clk_m) +temporal;                   /* stage 3 exit latch */
                (a)
```

```
%instr  M1 d, d          (double; clk_m)              /* launch multiply */
        {m1 = $1 * $2;}
        [M1]             (1,1,0)
%instr  M2               (double; clk_m)              /* advance through stage 2 */
        {m2 = m1;}
        [M2]             (1,1,0)
%instr  M3               (double; clk_m)              /* advance through stage 3 */
        {m3 = m2;}
        [M3]             (1,1,0)
%instr  FWB d            (double; clk_m)              /* catch result */
        {$1 = m3;}
        [FWB]            (1,1,0)
                (b)
```

Figure 4.3: **Maril description for i860 floating point multiply**. (a) Declarations for a multiply clock *clk_m* and its associated temporal registers. Temporal registers represent latches in the multiply pipeline. (b) Instruction directives for i860 floating point multiply. Each multiply sub-operation affects *clk_m*.

A condition code register is, in effect, a temporal register. Like an EAP latch, it is set by some instructions, but unmodified by others. The compiler writer declares the condition code register to be a temporal register and indicates which instructions affect the condition code. Marion schedulers then ensure that the condition code is set and referenced properly.

## 4.6 Temporal Scheduling

Separating pipeline sub-operations can cause a non-backtracking scheduler to deadlock. To avoid this, Marion modifies the code DAG before scheduling. This section describes how Marion represents temporal information in the code DAG and shows how deadlock can occur. It also presents an algorithm to prevent deadlock, given that no chaining (*i.e.*, EAP results fed directly into another EAP) is allowed and proves the algorithm correct. The next section relaxes the restriction on chaining, gives a modified algorithm and proves the modified algorithm correct.

If a code DAG edge represents a true dependence via a temporal register based on clock $k$, then Marion marks it as a *temporal edge* based on $k$. A sequence of nodes connected by temporal edges is called a *temporal sequence*. Each sequence is restricted to have only one head, *seqhead*, and one tail, *seqtail*. In addition, a given node cannot be a member of more than one sequence. These restrictions are discussed in more detail later in this section.

Intuitively, a temporal sequence is a sequence of EAP sub-operations required to perform one operation. The seqhead launches the operation; the seqtail catches the result. More formally, let $G = (N, E)$ be a code DAG, where $N$ is the set of nodes and $E$ is the set of edges between nodes.

**Definition:** Let $k$ be a clock. Let $x \in N$ such that there is an outgoing temporal edge $(x, y) \in E$ that is based on $k$ and there is no incoming a temporal edge $(z, x) \in E$ that is based on $k$. Recursively define $T$ to be a *temporal sequence based on k*:

$$T = \begin{cases} \{x\}, & \\ T \cup \{y\}, & \text{if } \exists\, z \in T \text{ such that } (z, y) \text{ is a temporal edge based on } k \end{cases}$$

**Definition:** Let $T$ be a temporal sequence based on $k$.
*seqhead*$(T) = x$ such that $x \in T$ and there is no $z$ such that $(z, x)$ is a temporal edge based on $k$.
*seqtail*$(T) = x$ such that $x \in T$ and there is no $z$ such that $(x, z)$ is a temporal edge based on $k$.

**Definition:** Let $x \in N$.

$$seqhead(x) = \begin{cases} \text{seqhead}(T), \text{if } x \in \text{a temporal sequence } T \\ x, \text{ otherwise} \end{cases}$$

$$seqtail(x) = \begin{cases} \text{seqtail}(T), \text{if } x \in \text{a temporal sequence } T \\ x, \text{ otherwise} \end{cases}$$

When the source of a temporal edge has been scheduled, but the destination has not, the scheduler is said to be *scheduling across a temporal edge*. The temporal edge indicates that the value it represents is held in a temporal register and therefore is ephemeral; to prevent the loss of that value the scheduler adheres to the following rule:

**Rule 1:** If there is a temporal edge $(x, y)$ based on clock $k$, and $x$ has been scheduled, then an instruction $z \neq y$ that affects $k$ may not be scheduled before $y$, but may be packed with $y$.

If $z$ were scheduled before $y$, the value held in the temporal register would be lost, because $z$ affects clock $k$. For example,

M1 $d4, d5$;   M2;   M3;   FWB $d6$

forms a temporal sequence based on *clk_m*. After "M1 $d4, d5$" is scheduled, the initial sub-operation in a similar sequence (*e.g.*, "M1 $d7, d8$") may not be scheduled before M2, but may be packed with it, since their resources do not overlap.

After the two sub-operations are packed together and scheduled, the scheduler will be scheduling across two temporal edges. To avoid violating Rule 1, all sub-operations that are destinations of temporal edges based on the same clock are bundled into a *temporal group*. Each temporal group is treated as a single instruction to be scheduled. In effect, temporal grouping pre-packs all sub-operations that *must* be packed together. Other sub-operations may then be packed with a temporal group.

Without any code DAG modification, Rule 1 can cause the scheduler to deadlock. Figure 4.4 gives an example. The edge (M2,M3) is a temporal edge based on clock $m$. The edge (FWB,M3) is not a temporal edge. If M2 is scheduled before FWB, the scheduler will deadlock. This is because FWB must precede M3 in the schedule, but FWB affects clock $m$. Scheduling FWB would violate Rule 1, since it would cause M2's result value to be lost. To force a schedule that avoids deadlock,

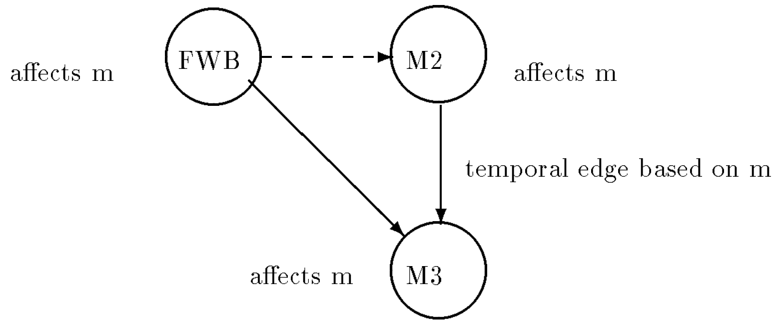



Figure 4.4: **Scheduling deadlock**. Without the dashed edge (FWB,M2), M2 may be scheduled before FWB, causing deadlock.

Marion adds the extra code DAG edge (FWB,M2). (This example has been simplified from actual i860 examples.)

The algorithm to add these edges is driven by *alternate entries* into temporal sequences. In the previous example, (FWB,M3) is an alternate entry.

> **Definition:** Let $T$ be a temporal sequence based on $k$. Let $y \in T$ such that $y \neq$ seqhead($T$). $(x, y) \in E$ is an *alternate entry* into $T$ iff $x \notin T$.

An alternate entry edge is sometimes caused by an anti-dependence between a previous use and subsequent definition of a physical register.

Because of alternate entries, temporal sequences must be protected to avoid scheduling deadlock. Intuitively, to protect a temporal sequence $T$ based on clock $k$, whenever an alternate entry into $T$ is found, Marion searches each path backward from the alternate entry. If an instruction that affects $k$ is found, an edge is added from that instruction (or a member of its temporal sequence) to the seqhead of $T$. This ensures that all ancestors of any node in a temporal sequence are scheduled before the head of the sequence. Given a topological sort of the input DAG, these edges are always forward edges or cross edges, never back edges. Therefore, they cannot cause a cycle in the graph. The actual algorithm differs somewhat.

The remainder of this section describes the algorithm to protect temporal sequences and proves that it avoids scheduling deadlock. This algorithm prohibits chaining between EAPs, which greatly simplifies the process of preventing deadlock. The next section relaxes this restriction, gives a modified algorithm and proves that it avoids deadlock.

Marion places some restrictions on the target machine. The restrictions either reflect the actual hardware or should have little affect on the generated code. The four restrictions are listed, followed by a discussion:

> (1) Although a node may affect more than one clock, each temporal edge is based on only one clock. This means that only a single clock may control the latches in an EAP.
>
> (2) No node may be a member of more than one temporal sequence. This implies that no chaining is allowed and both the number of incoming temporal edges *(temporal fanin)* and outgoing temporal edges *(temporal fanout)* are at most one.

(3) The packing restrictions prohibit two nodes from different temporal sequences from being scheduled together only if both are the $i$th nodes in their respective sequences.

(4) No deadlock can occur due to general purpose register usage.

These are not overly restrictive. One clock is sufficient to model EAP control. Restriction 2 prohibits a common subexpression temporary from being a temporal register; although this is possible on some architectures, its prohibition should have only minor effect on the generated code. Restriction 2 also prohibits chaining, but this is relaxed in the next section. In practice, on the i860, Restriction 3 has been liberalized and no difficulty was observed. However, this could be a problem for other targets. Restriction 4 is met by requiring anti-dependence edges to be added to the code DAG for independent uses of the same physical register [GM86] or by a scheduling algorithm that specifically avoids this kind of deadlock [HG83].

The following definitions are used in Algorithm 2:

**Definition:** Let $x \in N$. $clks(x) = \{k \mid k \text{ is a clock and } x \text{ affects } k\}$.

**Definition:** Let $x, y \in N$. If $x \xrightarrow{*} y$,[6] then $dist(x, y)$ = the number of edges on the path.

**Definition:** Let $x, y \in N$. Let $k$ be a clock. If $x \xrightarrow{*} y$ such that all edges are temporal based on $k$, then $dist_k(x, y)$ = the number of edges on the shortest such path.

**Definition:** Let $x \in N$. $sched_k(x)$ is the schedule time of $x$ according to clock $k$. $sched(x)$ is the schedule time of $x$ according to primary system clock (which defines the processor's cycles).

Algorithm 2, shown in Figure 4.5, walks the code DAG in reverse topological order. Whenever it finds an alternate entry $(y, x)$ into a temporal sequence $T$ based on $k$, it adds $x$ to a list associated with $y$. The lists are propagated backwards through the DAG until a node $z$ is found that affects some clock. Then, an edge is added from $z$ (or a member of $z$'s temporal sequence) to $seqhead(T)$.

The algorithm uses temporal sequence distances to determine where to add edges. For example, in Figure 4.6(a), the temporal sequence $A$ is longer than the sequence $B$; both are based on clock $a$ and all instructions affect $a$. The algorithm adds the dotted edge (A2,B1). If the edge (A1,B1) were added instead, deadlock could occur as follows: A2 and B1 are scheduled together; then, A3 and B2 must be scheduled together; but then, A4 cannot be scheduled without losing B3's operand.

If the source of an alternate entry $(y, x)$ into a sequence $S$ affects a different clock than the $S$'s clock, then the algorithm adds an edge from $y$, instead of seqhead($y$), as shown in Figure 4.6(b). This avoids another pitfall. For example, given the four sequences in Figure 4.6, if sequence $B$ were based on clock $b$ and neither dotted edge were present, then B2 and D2 could be scheduled before either A4 or C4, causing deadlock.

The algorithm has a worst case behavior of $\mathcal{O}(ne)$, where $n = |N|$ and $e = |E|$. Each list $x.list$ is kept in order (according to code thread order) and no duplicates are allowed; therefore, a list has at most $n$ items. This implies that adding a list item takes $\mathcal{O}(n)$, but so does merging two lists. Therefore, the total propagation time over a given DAG is $\mathcal{O}(ne)$. Since the propagation time dominates, the worst case behavior is $\mathcal{O}(ne)$.

---

[6]Notation: $x \xrightarrow{*} y$ means that there is a path in the code DAG of length zero or more from $x$ to $y$. $x \xrightarrow{+} y$ means that there is a path of length one or more.

```
Algorithm 2: protect_temporal_sequence
    Input: code DAG G = (N, E)
    Output: modified code DAG
    Define foreach x ∈ N:
        x.list = an initially empty list of pairs (y, d), such that y ∈ N and d is a distance
    Method:
        foreach x ∈ N (in reverse code thread order)
            if | clks(x) | > 0
                foreach (y, d) ∈ x.list
                    let k = the clock on which y's temporal sequence is based
                    if ∃ j ∈ clks(x) such that j ≠ k then
                        add edge (x,seqhead(y)) to E
                    elseif dist_k(seqhead(x), x) ≤ d then
                        add edge (seqhead(x),seqhead(y)) to E
                    else
                        let w = (dist_k(seqhead(x), x) − d + 1)th node in x's temporal sequence
                        add edge (w,seqhead(y)) to E
                    endif
                    remove (y, d) from x.list
                endfor
            endif
            foreach y ∈ predecessors(x)
                if (y, x) is an alternate entry into a temporal sequence T then
                    add (x, dist_k(seqhead(x), x)) to y.list
                else
                    foreach (z, d) ∈ x.list
                        add (z, d) to y.list
                    endfor
                endif
            endfor
        endfor
```

Figure 4.5: **Algorithm protect_temporal_sequence** to add code DAG edges to protect temporal sequences. The algorithm finds alternate entries and propagates their locations backwards through the DAG until the appropriate node is found to become the source of a new edge.

Now I show that the code DAG output by Algorithm 2 avoids deadlock. During scheduling, a node is either *scheduled* or *unscheduled*. An unscheduled node with no unscheduled predecessors is considered to be *ready*. Operation latency is not considered here, since no-ops can always be scheduled until any *ready* node can be scheduled without causing a delay.

Since all nodes in a temporal sequence are consecutive in the code thread and since for every edge $(x, y)$ added by the algorithm, $x$ precedes $y$ in the code thread, the algorithm creates no cycles in the output graph $G$. Therefore, until every node is scheduled, there must be some ready node. The proof shows that, assuming some DAG node has not been scheduled, the scheduler can always proceed by finding a node to schedule. The focus is on the set $P$ of temporal edges across which the scheduler is scheduling:

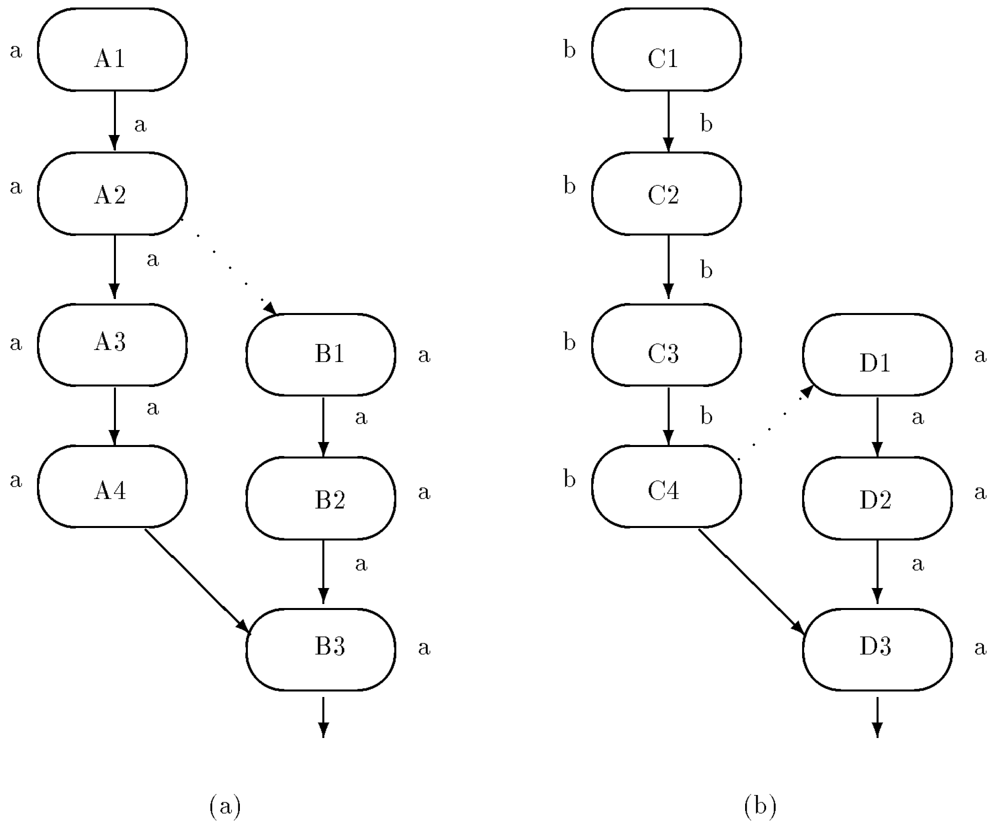



Figure 4.6: **Alternate entries into temporal sequences**. (a) A long temporal sequence $A$ has an alternate entry into a short temporal sequence $B$. (b) Temporal sequence $D$ is based on clock $a$. The source of the alternate entry into $D$ is based on a different clock, clock $b$. Algorithm 2 adds the dotted edges to avoid deadlock.

$$P = \{(x, y) \mid (x, y) \in E \text{ is a temporal edge and } x \text{ is scheduled and } y \text{ is unscheduled}\}.$$

Lemma 1 states that if the destination of every temporal edge in $P$ is ready, then some or all of the nodes can be grouped and scheduled.

**Lemma 1:** If $\forall\, (x, y) \in P$, $y$ is ready, then the scheduler can proceed.

**Proof:** If $|\,P\,| = 0$ then any ready node can be scheduled.

Let $|P| > 0$. Since $\forall\, (x, y) \in P$, $y$ is ready, all such nodes can be grouped together and the group can be scheduled. $\square$

Lemma 2 states that if the destination of a temporal edge $(x, y) \in P$ is not ready, then some ancestor of $y$ can be scheduled. In this case $y$ must be an alternate entry point. The proof partitions all ready ancestors of the alternate entry into two sets: those that affect some clock and those that affect none. Then, it proves by contradiction that the former set is empty; if node $u$ affects some clock, then either it must be an ancestor of seqhead$(x)$, which leads to a contradiction, or it must

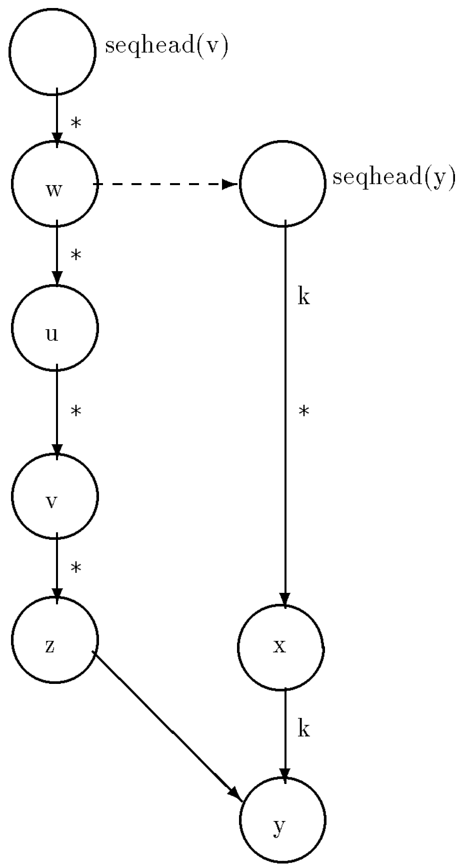



Figure 4.7: **Lemma 2 example**. If $u$ affects some clock, then Algorithm 2 adds the dashed edge $(w, \text{seqhead}(y))$. Since seqhead($y$) and $x$ are scheduled, $u$ is also scheduled. The symbol "*" means there are zero or more edges along this path.

be part of another temporal sequence $T$. In the latter case, $w \in T$ such that $w \xrightarrow{*} \text{seqhead}(y)$, which also leads to a contradiction, because nodes between $w$ and $u$ must be scheduled in lock step with nodes between seqhead($y$) and $y$ and the distance between $w$ and $u$ must be less than or equal to the distance between seqhead($y$) and $y$. (See Figure 4.7.)

**Lemma 2:** If $\exists\,(x, y) \in P$ such that $y$ is not ready and $\exists\,(z, y) \in E$ such that $(z, y)$ is not a temporal edge and $z$ is unscheduled, then $\exists\, u \in N$ such that $u \xrightarrow{*} z$ and $u$ can be scheduled.

**Proof:** Let $T$ be the temporal sequence such that $y \in T$.
Let $A = \{u \mid u \xrightarrow{*} z$ and $u$ is ready and $|\,\text{clks}(u)| = 0\}$. Let $B = \{u \mid u \xrightarrow{*} z$ and $u$ is ready and $|\,\text{clks}(u)| > 0\}$. Clearly $A \cup B \neq \emptyset$.

Show $A \neq \emptyset$ by contradiction. Assume $A = \emptyset$. This implies $B \neq \emptyset$. Let $u \in B$. Let $v \in N$ be the last node that affects some clock on the path $u \xrightarrow{*} v \xrightarrow{*} z$. Since $(z, y)$ is an alternate entry into $T$, then from Algorithm 2, there is an edge $(w,\text{seqhead}(y)) \in E$ such that seqhead$(v) \xrightarrow{*} w \xrightarrow{*} v$. Since seqhead$(y) \xrightarrow{*} x \rightarrow y$ and $x$ is scheduled, seqhead$(y)$ is scheduled. Therefore $w$ is scheduled. If for all temporal sequences $S$, $v \notin S$, then $v = w$. But, since $v$ is unscheduled, $v \neq w$. Therefore, there is a temporal sequence $S$ such that $v, w \in S$. Now, if $T$ and $S$ are based on different clocks, then $v = w$ (from Algorithm 2). Therefore, $T$ and $S$ are based on the same clock $k$. This implies that after seqhead$(y)$ is scheduled, nodes from $T$ and $S$ are scheduled in lock step according to clock $k$. Now, from Algorithm 2,

$$\mathrm{dist}_k(w, v) \leq \mathrm{dist}_k(w, z) \leq \mathrm{dist}_k(\mathit{seqhead}(y), y).$$

Therefore,

$$\begin{aligned}
\mathrm{sched}_k(x) &= \mathrm{sched}_k(\mathrm{seqhead}(y)) + \mathrm{dist}_k(\mathrm{seqhead}(y), y) - 1 \\
&\geq \mathrm{sched}_k(\mathrm{seqhead}(y)) + \mathrm{dist}_k(w, v) - 1 \\
&\geq \mathrm{sched}_k(w) + 1 + \mathrm{dist}_k(w, v) - 1 \\
&\geq \mathrm{sched}_k(w) + \mathrm{dist}_k(w, v) \\
&\geq \mathrm{sched}_k(v)
\end{aligned}$$

This implies that sched$(x) \geq$ sched$(v)$. Since $x$ is scheduled, then $v$ is scheduled. $\bigotimes$[7]

Therefore $A \neq \emptyset$. Since $u \in A$ implies that $u$ affects no clocks, $u$ can be scheduled. □

Theorem 1 uses the two lemmas to show that the scheduler avoids deadlock.

**Theorem 1:** Given the output of Algorithm 2, the scheduler will not deadlock.

**Proof:** Assume $\forall\,(x, y) \in P$, $y$ is ready. By Lemma 1 the scheduler can proceed.

Assume $\exists\,(x, y) \in P$ such that $y$ is not ready. This implies $\exists\,(z, y) \in E$ such that $z$ is unscheduled. Since temporal fanin $\leq 1$, $(z, y)$ is not temporal. Therefore, by Lemma 2, the scheduler can proceed.

Therefore, the scheduler will not deadlock. □

## 4.7 Temporal Scheduling with Chaining

*Chaining* occurs when a pipeline sends its result directly to its own input or to another pipeline's input without using a general purpose register to hold the intermediate value. This section discusses chaining, presents a modified algorithm to protect temporal sequences and proves that the modified algorithm avoids deadlock.

Marion models chaining by introducing sub-operations that explicitly feed values from one pipeline to another. The code selector produces these sub-operations by matching patterns in the order specified by the compiler writer. For example, Figure 4.8 shows a specification of several add pipeline instructions for the i860. The A1m instruction takes one operand from the register set and one from the multiply pipeline latch $m3$. (The multiply instructions are shown in Figure 4.3.) Thus, the expression "$d9 = d6 * d7 + d8$" yields the following code sequence:

M1 $d6, d7$;   M2;   M3;   A1m $d8$;   A2;   A3;   FWBa $d9$.

The sub-operation names and resource names have no implicit meaning. All of the inter-pipestage movement is captured by the temporal registers.

[7] The $\bigotimes$ symbol represents contradiction.

```
%instr Aam d          (double; clk_a, clk_m)      /* Launch add */
       {a1 = a3 + m3;}                            /* operands: from adder,multiplier */
       [A1]           (1,1,0)
%instr A1m d          (double; clk_a, clk_m)      /* Launch add */
       {a1 = $1 + m3;}                            /* operands: from reg,multiplier */
       [A1]           (1,1,0)
%instr A1 d, d        (double; clk_a)             /* Launch add */
       {a1 = $1 + $2;}                            /* operands: from regs */
       [M1]           (1,1,0)
%instr A2             (double; clk_a)             /* Advance add thru stage 2 */
       {a2 = a1;}
       [M2]           (1,1,0)
%instr A3             (double; clk_a)             /* Advance add thru stage 3 */
       {a3 = a2;}
       [M3]           (1,1,0)
%instr FWBa d         (double; clk_a)             /* Catch add result */
       {$1 = a3;}
       [FWB]          (1,1,0)
```

Figure 4.8: **Instructions for the i860 add pipeline**. Instruction A1m takes an operand from the multiply pipeline. Instruction Aam takes one operand from the add pipeline and the other from the multiply pipeline. Temporal registers $a1$, $a2$ and $a3$ represent the latches in the add pipeline.

To support chaining, the restriction that no node may be part of more than one temporal sequence is removed. This allows temporal fanin to be greater than one, but temporal fanout is still restricted to be less than or equal to 1. If a node $x$ has two incoming temporal edges, then it is part of a temporal tree. Intuitively, a *temporal tree* is a subgraph of the code DAG such that all nodes are connected by temporal edges. Since temporal fanout is still at most 1, the subgraph is a tree. More formally,

> **Definition:** Let $R$ and $S$ be temporal sequences. $T = R \cup S$ is a *partial temporal tree* iff $\mathrm{seqtail}(R) \in S$ or $\mathrm{seqtail}(S) \in R$. $R$ is a *temporal tree* iff either (1) $T$ is a partial temporal tree and there is no temporal sequence $V$ such that either $\mathrm{seqtail}(V) \in T$ or $\mathrm{seqtail}(T) \in V$, or (2) $T$ is a temporal sequence and $\exists\, x \in T$ such that $|\mathrm{clks}(x)| > 1$.
>
> **Definition:** Let $T$ be a temporal tree. $\mathit{entries}(T) = \{x \mid x \in T$ and there is no edge $(z,x) \in E$ such that $(z,x)$ is a temporal edge$\}$. Let $y \in T$. $\mathit{entries}(y) = \{x \mid x \in \mathrm{entries}(T)$ and $x \xrightarrow{*} y$ such that all nodes on the path are members of $T\}$.
>
> **Definition:** Let $T$ be a temporal tree. Let $x \in T$. $x$ is a *merge point* iff $x$ has temporal fanin $> 1$ or $|\mathrm{clks}(x)| > 1$ or there are temporal sequences $S$ and $T$ such that $S \neq T$ and $x \in S$ and $x \in T$.
>
> **Definition:** Let $T$ be a temporal tree. $\mathit{clks}(T) = \{k \mid \exists\, x \in T$ such that $k \in \mathrm{clks}(x)\}$.
>
> **Definition:** Let $S$ and $T$ be temporal trees. $S$ *precedes* $T$ iff $\forall x \in S$ and $\forall y \in T$, $x$ precedes $y$ in the code thread.

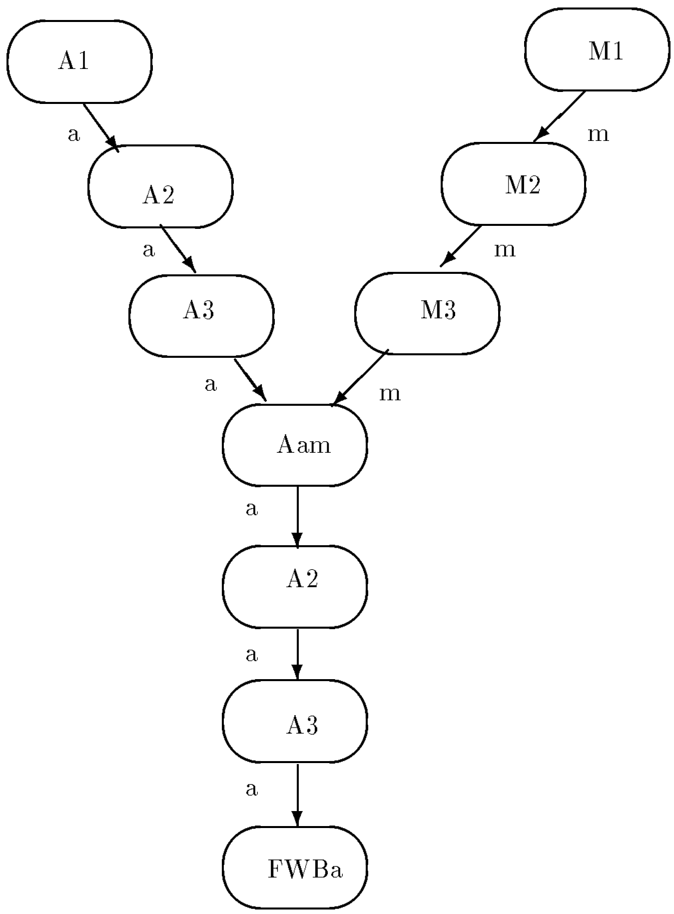



Figure 4.9: **Example temporal tree**. Nodes A1 through FWBa form a temporal sequence based on clock $a$. Nodes M1 through Aam form a temporal sequence based on clock $m$. Together the two temporal sequences form a temporal tree with a merge point Aam. Nodes A1 and M1 are the entries into the temporal tree.

**Definition (modified):** Let $x \in N$. Let $R, S$ be temporal sequences.

$$seqhead(x) = \begin{cases} \text{seqhead}(S), \text{if } x \in R \text{ and } x \in S \text{ and seqtail}(R) \in S \\ \text{seqhead}(R), \text{if } x \in R \\ x, \text{ otherwise} \end{cases}$$

$$seqtail(x) = \begin{cases} \text{seqtail}(R), \text{if } x \in R \text{ and } x \in S \text{ and seqtail}(R) \in S \\ \text{seqtail}(R), \text{if } x \in R \\ x, \text{ otherwise} \end{cases}$$

Figure 4.9 shows a temporal tree that contains two temporal sequences. The node Aam is a merge point; it takes one operand directly from the add pipeline and the other directly from the multiply pipeline. The edge (M3,Aam) is an alternate entry into the temporal sequence that is based on clock $a$.

Overlapping the execution of temporal trees is difficult. One method I considered is the following: require every entry of a temporal tree to be scheduled before any entry of the next tree. However, this will not prevent deadlock if temporal sequences based on a given clock differ in length. A merge point in one tree may be waiting for nodes dependent on clock $j$ to be scheduled, while clock $k$ is not permitted to tick; another tree may be waiting for nodes dependent on clock $k$ to be scheduled, while clock $j$ is not permitted to tick.

Therefore, I chose to force any two temporal trees that have some clock in common to be *well ordered.* This means that all merge points of the first tree must be scheduled before any entry of the second tree. Marion enforces this by adding code DAG edges as follows:

> Let $S$ and $T$ be two temporal trees such that,
>
> (1) $\mathrm{clks}(S) \cap \mathrm{clks}(T) \neq \emptyset$,
> (2) $S$ precedes $T$, and
> (3) there is no temporal tree $R$ such that $S$ precedes $R$ and $R$ precedes $T$ and either $\mathrm{clks}(S) \subseteq \mathrm{clks}(R)$ or $\mathrm{clks}(R) \supseteq \mathrm{clks}(T)$.
>
> Let $x$ be the last merge point in $S$. Then $\forall\ y \in \mathrm{entries}(T)$, add edge $(x, y)$ to $E$.

To prevent these additional edges from creating cycles in the graph, the code selector ensures that temporal trees are not nested by evaluating all subexpressions of temporal tree entries before any part of the temporal tree itself.

Temporal trees complicate the protection of temporal sequences. Algorithm 3, shown in Figure 4.10, is a modification of Algorithm 2. It accommodates alternate entries into a temporal sequence that is part of a temporal tree. Other temporal sequences are handled as in Algorithm 2. Within a temporal tree each new edge that is added because of an alternate entry $(z, y)$ has a destination that is a member of the set of entries(seqhead($y$)), instead of seqhead($y$). If the alternate entry is itself part of the same temporal tree, the source of the new edge is an ancestor of that sequence. The worst case behavior of Algorithm 3 is the same as the that of Algorithm 2, $\mathcal{O}(ne)$.

Now I show that, given a code DAG $G = (N, E)$ output by Algorithm 3, the scheduler avoids deadlock. Again, the proof focuses on the set $P$ of edges across which the scheduler is scheduling.

$$P = \{(x, y) \mid (x, y) \in E \text{ is a temporal edge and } x \text{ is scheduled and } y \text{ is unscheduled}\}.$$

Both Lemmas 1 and 2 still hold, because they consider cases that are unaffected by the modifications to the algorithm or by the presence of temporal trees. Lemma 3 considers edges whose destinations are temporal tree merge points. It argues that if two such edges are in $P$, their associated trees may not affect any clock in common. Two such edges in $P$ cannot be part of the same temporal tree, because merge points are also alternate entry points; Algorithm 3 in effect orders alternate entry points within the same temporal tree. Two such edges cannot be in separate trees, if those trees affect some clock in common, because temporal trees that affect some clock in common are well ordered.

> **Lemma 3:** Let $Q = \{(x, y) \in P \mid$ there is a temporal tree $T$ such that $y \in T$ and $y$ is a merge point$\}$. Let $S$ and $T$ be temporal trees. If $(x, y), (x', y') \in Q$ and $y \in S$ and $y' \in T$, then $S \neq T$ and $\mathrm{clks}(S) \cap \mathrm{clks}(T) = \emptyset$.
>
> **Proof:** Let $(x, y), (x', y') \in Q$ and $y \in S$ and $y' \in T$.

**Algorithm 3:** *protect_temporal_sequence*

```
Input: code DAG G = (N, E); Output: modified code DAG G
Define foreach x ∈ N: x.list = list of tuples (h, k, d, S): ∃z, y such that x -*-> z → y
    and z → y is an alternate entry into a sequence based on k;
    h is seqhead(y); d = dist_k(h, y); S is the temporal
    sequence that contains z and y, if such a sequence exists.
    x.order = the topological order number.
Method:
    foreach x ∈ N (in reverse topological order)
        if | clks(x) | > 0
            foreach (h, k, d, S) ∈ x.list
                if x.order < h.order and (x is a merge point or x ∉ S)
                and (x, h ∉ the same temporal tree or x is not an ancestor of h) then
                    if ∃ j ∈ clks(x) such that j ≠ k or x is a merge point then
                        let w = x
                    elseif dist_k(seqhead(x), x) ≤ d then
                        let w = seqhead(x)
                    else
                        let w = (dist_k(seqhead(x), x) − d + 1)th node in x's sequence
                    endif
                    if y ∈ a temporal tree T then
                        foreach z ∈ entries(h), add edge (w, z) to E; endfor
                    else add edge (w, h) to E
                    endif
                    remove (h, k, d, S) from x.list
                endif
            endfor
        endif
        foreach y ∈ predecessors(x)
            if (y, x) is an alternate entry into a temporal sequence T based on k then
                if y, x ∉ the same temporal tree or ¬(y -+-> v), where v ∈ T, v → x then
                    if ∃ a temporal sequence S such that y, x ∈ S then
                        add (seqhead(x), k, dist_k(seqhead(x), x), S) to y.list
                    else
                        add (seqhead(x), k, dist_k(seqhead(x), x), ∅) to y.list
                    endif
                endif
            else
                foreach (z, k, d, S) ∈ x.list
                    if |clks(x)| = 0 or y ∈ S then add (z, k, d, S) to y.list
                    else add (z, k, max(0, d − 1), S) to y.list
                    endif
                endfor
            endif
        endfor
    endfor
```

Figure 4.10: **Modified algorithm protect_temporal_sequence.**

First, show that $S \neq T$ by contradiction. Assume $S = T$. Assume, without loss of generality, that $y$ precedes $y'$ in the code thread.

Case 1: Let $y \xrightarrow{+} y'$. $y$ cannot be an ancestor of $x'$, because $x'$ is scheduled and $y$ is not scheduled. Therefore, $\exists\, z \in N$ such that $y \xrightarrow{+} z \rightarrow y'$ and $z \rightarrow y'$ is an alternate entry into a temporal sequence $R$, where $x', y' \in R$. But from Algorithm 3, since $y$ is a merge point, $y \xrightarrow{+} x'$. $\otimes$

Case 2: $\exists\, s \in S$ such that $y \xrightarrow{*} s$ and $y' \xrightarrow{*} s$. Let $r, r' \in S$ such that $y \xrightarrow{*} r \rightarrow s$ and $y' \xrightarrow{*} r' \rightarrow s$. $(r, s)$ is an alternate entry into some temporal sequence $R$ such that $r', s \in R$. Therefore, from Algorithm 3, $\exists\, w \in N$ such that $w \xrightarrow{*} r$ and $w \xrightarrow{*} x'$. Since $y$ is a merge point, $y \xrightarrow{*} w \xrightarrow{*} r$. Therefore, $y \xrightarrow{*} x'$. Since $x'$ is scheduled, $y$ is scheduled. $\otimes$

Therefore, $S \neq T$.

Now, show $\text{clks}(S) \cap \text{clks}(T) = \emptyset$ by contradiction. Assume $\text{clks}(S) \cap \text{clks}(T) \neq \emptyset$. Let $S$ precede $T$. $y$ is a merge point of $S$. Since temporal trees whose clocks overlap are well ordered, $\forall v \in \text{entires}(T)$, $y \xrightarrow{+} v$. Therefore, $y \xrightarrow{+} x'$. Since $x'$ is scheduled, $y$ is scheduled. $\otimes$

Therefore, $\text{clks}(S) \cap \text{clks}(T) = \emptyset$. $\square$

Theorem 2 shows that the scheduler will not deadlock, given the output of Algorithm 3. First, the cases covered by Lemmas 1 and 2 are considered. Then, the case is considered where for all $(x, y) \in P$ such that $y$ is not ready, $y$ is a merge point of a temporal tree $T$. There must be some unscheduled $z$ such that $z \rightarrow y$ and some ancestor $v$ of $z$ must be ready. There are two subcases. First, if $v$ is in $z$'s temporal sequence, then $v$ can be grouped with other nodes and scheduled, because for all $(x', y') \in P$, either $y'$ is ready or $y'$ is a merge point of a temporal tree none of whose clocks overlap with $T$'s clocks. Second, if $v \xrightarrow{+} \text{seqhead}(z)$, then $v$ affects no clocks and, therefore, can be scheduled. This argument is similar to the proof of Lemma 2.

**Theorem 2:** Given the output of Algorithm 3, the scheduler will not deadlock.

**Proof:** If $\forall\, (x, y) \in P, y$ is ready, then by Lemma 1, the scheduler can proceed.

Let $(x, y) \in P$ such that $y$ is not ready. Then $\exists\, (z, y) \in E$ such that $z$ is unscheduled. Let $(z, y)$ be non-temporal. By Lemma 2, $\exists\, u \in N$ such that $u \xrightarrow{*} z$ and $u$ can be scheduled.

Now, $\forall\, (x, y) \in P$ such that $y$ is not ready, let $(z, y) \in E$ be temporal. Therefore, $y$ is a merge point. Let $T$ be a temporal tree such that $x, y, z \in T$. There are two cases.

Case 1: $\exists\, v \in N$ such that $v$ is ready and $\text{seqhead}(z) \xrightarrow{*} v \xrightarrow{*} z$. Let $k$ be the clock on which $(x, y)$ is based. $(z, y)$ must be based on some clock $j \neq k$. By Lemma 3, $z$ cannot be a merge point. Therefore, there is a temporal sequence $R$ based on $j$ such that $v, z \in R$ and $k \notin \text{clks}(v)$. By Lemma 3, there is no edge $(x', y') \in P$ such that $y'$ is a merge point of a temporal tree $S$ and $\text{clks}(S) \cap \text{clks}(T) \neq \emptyset$. Furthermore, $\forall\, (x', y') \in P$, either $y'$ is ready and not a merge point or $y'$ is a member of a temporal tree $S$ such that $\text{clks}(S) \cap \text{clks}(T) = \emptyset$. Therefore, $\forall\, (x', y') \in P$ such that $\text{clks}(y') \cap \text{clks}(v) \neq \emptyset, y'$ can be grouped with $v$ and the group can be scheduled.

Case 2: $\exists\, v$ such that $v$ is ready and $v \xrightarrow{+} \text{seqhead}(z)$. If $v \in T$, then seqhead$(z)$ is a merge point. Therefore, by Algorithm 3, $\text{seqhead}(z) \xrightarrow{*} x$. But seqhead$(z)$ is unscheduled and $x$ is scheduled. Therefore, $v \notin T$. Let $k$ be the clock on which $(x, y)$ is based. Let $S$ be a temporal sequence such that $x, y \in S$. If $\exists\, j \in \text{clks}(v)$ such that $j \neq k$, then, from Algorithm 3, $v \xrightarrow{+} x$, because $(z, y)$ is an alternate entry into $S$. But $v$ is unscheduled and $x$ is scheduled. Therefore, either $|\text{clks}(v)| = 0$ or $\text{clks}(v) = \{k\}$.

Show $\text{clks}(v) \neq \{k\}$ by contradiction. Assume $\text{clks}(v) = \{k\}$. Since $k \in \text{clks}(v), \exists\, w \in N$ such that $\text{seqhead}(v) \xrightarrow{*} w \xrightarrow{*} v$ and $w \xrightarrow{*} x$ (because Algorithm 3 adds an edge from $w$ to an ancestor of seqhead$(y)$). If $v = w$, then $w$ is unscheduled. But $x$ is scheduled. Therefore, $v \neq w$. This implies that there is a temporal sequence $R$ such that $w, v \in R$ and $R$ and $S$ are both based on $k$. This implies that after seqhead$(y)$ is scheduled, nodes from $S$ and $T$ are scheduled in lock step according to $k$. Now, from Algorithm 3,

$$\text{dist}_k(w, v) \leq \text{dist}_k(w, \text{seqtail}(v)) \leq \text{dist}_k(\text{seqhead}(y), y).$$

Therefore,

$$\begin{aligned}
\text{sched}_k(x) &= \text{sched}_k(\text{seqhead}(y)) + \text{dist}_k(\text{seqhead}(y), y) - 1 \\
&\geq \text{sched}_k(\text{seqhead}(y)) + \text{dist}_k(w, v) - 1 \\
&\geq \text{sched}_k(w) + 1 + \text{dist}_k(w, v) - 1 \\
&\geq \text{sched}_k(w) + \text{dist}_k(w, v) \\
&\geq \text{sched}_k(v)
\end{aligned}$$

This implies that $\text{sched}(x) \geq \text{sched}(v)$. Since $x$ is scheduled, then $v$ is scheduled. $\otimes$

Therefore, $|\,\text{clks}(v)| = 0$. This implies that $v$ can be scheduled.

Therefore, the scheduler will not deadlock. $\square$

## 4.8 Scheduling for the i860

The machine description for the i860 uses all of the features described, including classes, class elements and clocks. (See Appendix D for the complete machine description.) As discussed in Section 4.5, Marion models the i860 FPU as a long instruction word processor. Eight fields in the long instruction word correspond to the FPU pipestages: three for the adder, three for the multiplier, one for the floating point write-back bus and one for the special T register, which holds a value in between the multiplier and the adder.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| M1 | M2 | M3 | A1 | A2 | A3 | FWB | T |

Figure 4.11 shows the simplified i860 FPU datapath diagram on which the machine description is based. Both the add and multiply pipelines have three stages, which are represented by boxes; the stages are separated by latches, which are represented by ovals. The dashed lines show the clocks that control the latches. To launch both an add and a multiply with one instruction, four source and two destination operands are needed, but only three operands, *src1*, *src2* and *rdest*, are general purpose registers; the others come from chaining the pipelines together or from special registers. Marion supports the special T register, which feeds the adder, but not the KR and KI special

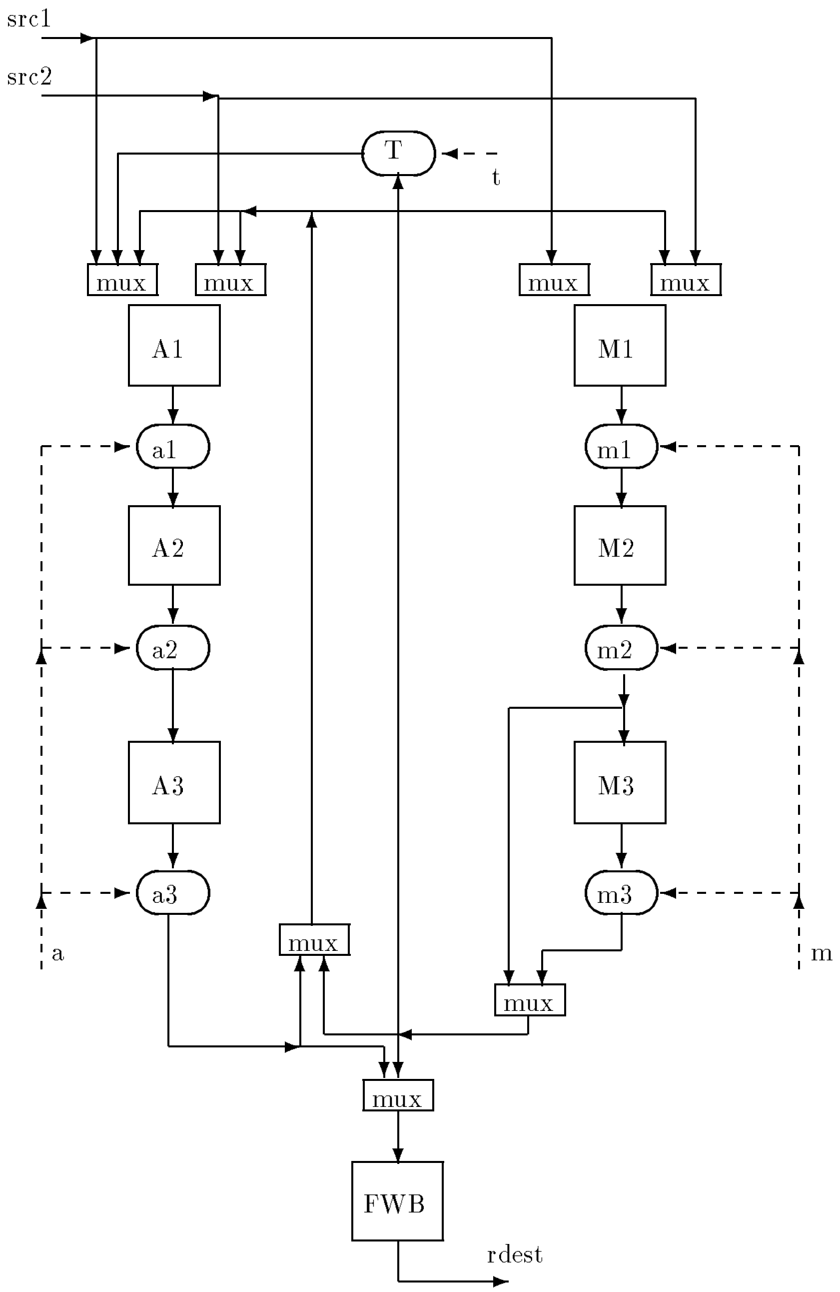



Figure 4.11: **Marion's view of the i860 FPU data path**. The dashed lines show the clocks that control the explicitly advanced pipelines. The T register is a latch that is independent of both pipelines; consequently, it has its own clock. Marion does not use the KR and KI registers, which are not shown is this diagram.

registers, which feed the multiplier. The T register is loaded by the multiplier result; therefore, if an expression adds the result of two multiplies, using the T register is clearly advantageous. The KR and KI registers, however, are loaded only from general purpose floating point registers; therefore, the profitability of using them depends on the usage context. It may be possible for the code selector to identify cases that use KR or KI, but I have not investigated these cases. An example where KR or KI could be used is to hold a loop invariant value that is multiplied by an array element on each iteration of the loop.

Not all of the combinations of add and multiply sources and destinations are supported by the i860 architecture. To enforce the necessary restrictions, all legal long instruction words are defined to be class elements. Associated with each individual sub-operation, such as A1 or M2, is a class; the class contains the elements (long instruction words) of which the sub-operation could be a part. Figure 4.12 shows the i860 machine description class and element declarations. The instructions *pfadd* and *pfmul* are long instruction words that advance a single pipeline. The "X" mnemonics represent long instruction words that advance both pipelines.[8] (See Figure 4.12 for the "X" mnemonic key.) After the "X" the first two character positions specify the multiplier's inputs, the next two the adder's inputs. The next character position indicates the adder's operation ('p' means add, 's' means subtract). The next position indicates which pipeline result is written to the result register. The last two characters are optional and specify that one or more of the special registers is loaded. For example, consider the X12tapat instruction: source registers *src1* and *src2* are fed to the multiplier; an add operation is initiated with inputs from T and the adder result $a3$; $a3$ is sent to the result register *rdest*; and T is loaded with the output of the multiplier.

The i860 machine description declares the adder and multiplier latches ($a1 - a3$ and $m1 - m3$) and the T register to be temporal registers. The sub-operations that compose EAP operations are declared as individual instructions. (See Figures 4.3 and 4.8.) The code selector produces these sub-operations, from which a code DAG containing temporal sequences and temporal trees is constructed. Marion schedules instructions using Algorithm 3 for temporal scheduling with chaining. The restrictions discussed in Section 4.7 all hold, except that there are additional packing restrictions, due to contention for FWB. Fortunately, this does not cause deadlock, because if FWB is needed by two instructions, then neither can be a merge point (since the write-back stage takes only one input) and they must be dependent on different clocks (otherwise they would be ordered by Algorithm 3).

Figure 4.13 shows the code produced by the Marion i860 compiler, using a Postpass strategy, for a sample C program fragment. The first instruction launches the multiply. The second advances the multiply pipeline and launches an add. The third advances both pipelines and launches another add. The fourth instruction advances the add pipeline while the multiply pipeline waits. The fifth launches another add, taking inputs directly from the add and multiply pipeline results. The remaining instructions drain the add pipeline.

## 4.9 Evaluation of Temporal Scheduling

Why does Marion support EAPs with temporal scheduling? An EAP could be treated as a "normal" pipeline, so that once an operation is launched, it advances through the pipeline on every cycle.

[8]These mnemonics are close to those given by Intel [Int89], but are easier to interpret than Intel's.

```
%element    pfadd       (clk_a);                /* add */
%element    pfmul       (clk_m);                /* multiply */
                        /* multiply and add/sub combinations */
%element    Xr21mpa     (clk_a, clk_m);
%element    Xr2tmpar    (clk_a, clk_m);
%element    Xr21apat    (clk_a, clk_m, clk_t);
%element    Xr2tapart   (clk_a, clk_m, clk_t);
%element    Xra12part   (clk_a, clk_m, clk_t);
%element    X12ampa     (clk_a, clk_m);
%element    Xra12par    (clk_a, clk_m);
%element    X12tapat    (clk_a, clk_m, clk_t);
%element    X12tmpa     (clk_a, clk_m);
%element    X12tapa     (clk_a, clk_m);
%element    Xr21mpm     (clk_a, clk_m);
%element    Xr2tmpmr    (clk_a, clk_m);
%element    Xr21mpmt    (clk_a, clk_m, clk_t);
%element    Xr2tmpmrt   (clk_a, clk_m, clk_t);
%element    Xrm12pmt    (clk_a, clk_m, clk_t);
%element    X12mmpm     (clk_a, clk_m);
%element    Xrm12pm     (clk_a, clk_m);
%element    X12tmpmt    (clk_a, clk_m, clk_t);
%element    X12tmpm     (clk_a, clk_m);

%class M1 {pfmul, X12@@@@*};    /* m1 = $1 * $2 */
%class M2 {pfmul, X*};          /* m2 = m1 */
%class M3 {pfmul, X*};          /* m3 = m2 */
%class Rm {pfmul, X@@@@@m*};    /* $3 = m3 */
%class A1 {pfadd, X@@12p@*};    /* a1 = $1 + $2 */
%class A1m {X@@1mp@*};          /* a1 = $1 + m3 */
%class Atm {X@@tmp@*};          /* a1 = t + m3 */
%class A2 {pfadd, X*};          /* a2 = a1 */
%class A3 {pfadd, X*};          /* a3 = a2 */
%class Ra {pfadd, X@@@@@a*};    /* $3 = a3 */
%class Tld {X@@@@@@*t};         /* t = m3 */
```

| X mnemonic key | | | |
|---|---|---|---|
| Position | Meaning | Character | Meaning |
| 1 | Mul opnd 1 | r | KR (placeholder only) |
| 2 | Mul opnd 2 | t | T |
| 3 | Add opnd 1 | 1 | src1 |
| 4 | Add opnd 2 | 2 | src2 |
| 5 | Adder operation | p | add |
| 6 | rdest from | s | subtract |
| 7-8 | Special reg loads | a | from Adder |
| | | m | from Multiplier |

Figure 4.12: **Element and class declarations for the i860**. An element directive indicates which clocks the element affects. Class members take simple regular expressions, where “@” matches any single character and “*” matches any string of zero or more characters.

C fragment

| |
|---|
| $a = (x + b) + (a * x);$ |
| $\text{return}(y + x);$ |

| Cycle | i860 instruction | Remarks |
|---|---|---|
| 0 | pfmul.s a,x,f0 | $m1 \leftarrow a * x$ |
| 1 | rat1p2.s x,b,f0 | $a1 \leftarrow x + b$ |
| | | $m2 \leftarrow m1$ |
| 2 | rat1p2.s y,x,f0 | $a2 \leftarrow a1$ |
| | | $m3 \leftarrow m2$ |
| | | $a1 \leftarrow y + x$ |
| 4 | pfadd.s f0,f0,f0 | $a3 \leftarrow a2$ |
| | | $a2 \leftarrow a1$ |
| 5 | m12apm.s f0,f0,f0 | $a1 \leftarrow a3 + m3$ |
| | | $a3 \leftarrow a2$ |
| 6 | pfadd.s f0,f0,r | $r \leftarrow a3$ |
| | | $a2 \leftarrow a1$ |
| 7 | pfadd.s f0,f0,f0 | $a3 \leftarrow a2$ |
| 8 | pfadd.s f0,f0,a | $a \leftarrow a3$ |

Figure 4.13: **Code produced by the Marion i860 Postpass compiler**. The *remarks* column shows the operations that each instruction performs. The variables $a$, $b$, $x$ and $y$ are in floating point registers. The variable $r$ represents the return value register. Temporal registers *a1*-*a3* and *m1*-*m3* represent latches in the floating point add and multiply pipelines, respectively. On cycle 5 the add pipeline takes inputs from the multiply and add pipeline outputs.

However, this reduces scheduling opportunities, because sub-operations can be scheduled where complete operations cannot. In addition, chaining sometimes requires one pipeline to “wait” for another to catch up.

Given sub-operations, some form of temporal scheduling is necessary to keep track of intermediate values in the pipeline. Marion's approach is to protect temporal sequences before scheduling, so that deadlock cannot occur. This avoids backtracking and reliance on the register allocator, and removes the burden of temporal scheduling from the scheduling algorithm and code generation strategy.

An alternative to temporal scheduling with chaining is to allow the scheduler to chain “on the fly” whenever possible. The register that represents the “link” in the chain can be freed and reused. Touzeau uses this approach in the FPS-164 Fortran Compiler [Tou84], but because it requires the *scheduler* to manipulate chains and to perform local register allocation, it is not flexible enough for Marion's different code generation strategies.

Cohn *et al.* [CGLT89] faced a similar problem with *implicit registers* on the iWarp processor. Implicit registers are like Marion's temporal registers: they represent latches at the heads or tails of memory queues and allow values from memory to be fed directly into floating point operations and vice versa. The iWarp compiler groups nodes in the code DAG that are connected by implicit registers; the DAG is modified so that the set of edges to/from the node group is the union of the edges to/from the individual nodes in the group. If this causes a cycle, then the group is broken and operations are added to copy the implicit register into an explicit register. To schedule a group, the scheduler finds a legal schedule for all nodes in the group.

Davidson, Landskov *et al.* [LDSM80, DLSM81] allow two micro-operations (a producer and a consumer) to be connected by a *delay constraint* via a resource; this means that the consumer micro-operation must be scheduled at least *delay* cycles after the producer and the resource must not be used by another micro-operation. The resource is analogous to a temporal register. They do not indicate how to handle a sequence of such operations.

Backward scheduling, in which instructions are scheduled from the end of the basic block to the beginning, could be advantageous for temporal scheduling with chaining. Because temporal fanout is at most one, merge points effectively become fork points, which would not cause scheduling problems. However, the scheduler would still have to handle alternate entries. I have not investigated this approach.

Temporal scheduling allows sub-operations to be separated in the schedule and supports chaining between EAPs. However, to ensure correctness, a non-backtracking scheduler must be constrained by additional code DAG edges. I have not compared temporal scheduling with the methods used by the iWarp compiler or the FPS-164 Fortran compiler. The additional code DAG edges required for temporal scheduling constrain the scheduler, but the separation of sub-operations gives it more freedom. More study is needed to understand how much benefit temporal scheduling provides relative to its complexity.

## 4.10 Summary

This chapter has described Marion's instruction scheduling mechanisms. A labeled code DAG incorporates operation latencies, memory reference order requirements and, for some code generation strategies, restrictions on register use order. Resource vectors enable the scheduler to avoid structural hazards.

Temporal scheduling supports explicitly advanced pipelines, but additional DAG edges are required to avoid scheduling deadlock. Adding DAG edges separates temporal scheduling support from the scheduling algorithm, enabling Marion to embrace many algorithms. Temporal scheduling is also useful for machines with condition code registers.

# Chapter 5

# Methodology

This chapter describes the methodology used to compare code generation strategies and to examine the effect of architectural features on performance. Both studies use the same workload and measurement criteria. Methodological aspects that are specific to only one of the studies are discussed with that study.

## 5.1 Workload

The workload is composed primarily of floating point intensive benchmarks and applications, but also includes integer code. Many of the programs contain large basic blocks and an extensive use of multi-cycle operations. Because large blocks have a high degree of instruction-level parallelism and multi-cycle operations can cause long pipeline delays, this workload readily exposes differences among code generation strategies and architectural features that affect instruction scheduling. This workload is important because (1) many computation-intensive scientific programs fall into this category and (2) techniques that increase instruction-level parallelism, either by increasing basic block size, such as loop unrolling [DJ79] and trace scheduling [Fis81], or overlapping the execution of different loop iterations, such as software pipelining [Cha81, Lam88, DHB89], are becoming widely used. These techniques are profitable, even on scalar processors [WS87].

The programs in the workload are categorized into five groups:

(1) *Livermore* contains the first fourteen Livermore Loop kernels [McM72].

(2) *Nasker* contains the seven NAS kernel benchmarks [BB84]. BTR is a block tridiagonal solver; CHO performs Cholesky decomposition; EMI is a vortices emission simulation; FFT performs a 2-dimensional fast Fourier transform; GMT does a Gaussian elimination of matrix wall influence coefficients; MXM performs 4-way unrolled matrix multiply; and VPE does pentadiagonals inversion.

(3) *Perfect* includes eight programs from the PERFECT Club Benchmark Suite [BCK+89]. ARC2D models supersonic reentry; BDNA is a nucleic acid simulation; DYFESM models structural dynamics; FLO52 models transonic flow; MDG is a water molecule simulation; MG3D performs seismic migration; QCD is a lattice gauge; and TRACK performs missile tracking.

(4) *Misc* contains miscellaneous computation-intensive applications. NEURAL is a neural net simulator that trains a multi-layer neural network using backpropagation.[1] Three programs from the Rice Compiler Evaluation Suite (RiCEPS)[2] include BOAST,

[1] NEURAL was supplied by by David Cohn.

[2] RiCEPS programs are courtesy of Rice University.

which is a reservoir simulation, SPHOT, which is a photon transport problem solver, and WANAL1, which models wave equation boundary control. COSTSC is a network flow analyzer.[3]

(5) *Int* contains integer programs, including the benchmarks DHRYSTONE [Wei84], INTEGER and KNIGHT, LCC (the Lcc front end) and SPARSE, a sparse matrix equation solver. Sparse performs some floating point operations, but is dominated by data structure manipulation.

Both the Perfect and Misc groups include programs with very large basic blocks; many contain hundreds of instructions and some have more than one thousand. In particular, execution time in ARC2D, BDNA and MG3D is dominated by those large blocks. No loop unrolling was performed on any of the programs, but some of the NAS kernels were written with explicit unrolling.

## 5.2 Measurement Criteria

To compare code generation strategies or different architecture variations Marion computes, for each program,

$$total\_cycles = \sum_{b \in blocks} cycles_b * freq_b. \tag{5.1}$$

$Cycles_b$ is the number of machine cycles that would be required to execute block $b$ to completion. It is computed by the scheduler and takes into account all structural, control and data hazards. $Freq_b$ is the number of times block $b$ was executed when run by a separate profiling tool prior to the experiments. *Total_cycles* is an estimate of CPU execution time; it does not consider delays outside the processor, such as cache misses.

All data presented in the next two chapters is in terms of speedups or normalized execution times. Each program is normalized to a particular strategy or architecture, depending on the parameter of interest. In effect this gives equal weight to each program in the workload and avoids allowing a few of the large floating point programs to dominate the others. Programs in a group are summarized using the harmonic mean.

Code produced for the R2000 has been executed. For the Livermore group, on the average the execution time is 6% greater than the estimated CPU execution time. Section 8.2 discusses this slow down in more detail.

[3]COSTSC was supplied by Jim Larus. It was not used in the code generation strategies comparison.

# Chapter 6

# Integrating Register Allocation and Instruction Scheduling

Many compilers for RISCs perform register allocation and instruction scheduling separately, with each phase ignorant of the requirements of the other. In reality each can impose constraints on the other, sometimes producing inefficient code.

Code inefficiencies are caused by an excess of memory references or pipeline delays. When register allocation is performed first, the same physical register may be assigned to independent expression temporaries, reducing the potential instruction-level parallelism. This, in turn, decreases the scheduler's opportunities to mask pipeline delays. When scheduling precedes register allocation, the number of simultaneously live values may be increased, creating more spills to memory.

The major unresolved issue is whether this lack of communication between register allocation and instruction scheduling produces inefficient schedules; if it does, then to what extent must the two functions be integrated to improve the generated code? This chapter investigates this issue by comparing the performance of three strategies for register allocation and instruction scheduling:

(1) a simple Postpass strategy, in which global register allocation is performed before instruction scheduling [HG83, GM86];

(2) a variation of Integrated Prepass Scheduling (IPS) [GH88], in which the scheduler is invoked before register allocation and must schedule within a local register limit; and

(3) a new technique that I developed called RASE (Register Allocation with Schedule Estimates), which integrates register allocation and instruction scheduling by giving the register allocator cost estimates that quantify the effect of its allocation choices on the subsequently generated schedule.

The first strategy serves as a base for the experiments, since it cleanly separates register allocation and instruction scheduling. This approach is the choice of retargetable compilers that include instruction scheduling, such as GCC [Sta89]. RASE represents the other end of the spectrum of alternatives. It provides a vehicle through which the register allocator and the instruction scheduler can communicate their register requirements. My version of IPS occupies the middle ground; technically it separates the two functions, but it emulates communication by encouraging the scheduler to keep register pressure at a level conducive to good register allocation.

The experiments were performed using benchmarks and applications on compilers targeted to three RISCs, the MIPS R2000, the Motorola 88000 and the Intel i860.

IPS and RASE each produce code that is on the average 12% faster than Postpass, supporting the hypothesis that the lack of communication between register allocation and instruction scheduling produces inefficient schedules. On the other hand, the performance of code generated by IPS and RASE is almost identical; RASE surpasses IPS only on a few programs that contain very large basic blocks. For these particular architectures the close coupling of instruction scheduling and register allocation found in RASE is not necessary; therefore, IPS is a better choice, because it is conceptually simpler and less expensive at compile time. Nevertheless, strategies such as RASE are worth further exploration for other architectures (see Chapter 7) or if programs with large blocks become more common due to techniques such as loop unrolling and trace scheduling.

The rest of this chapter is organized as follows: Section 6.1 explains the problem with the lack of communication between instruction scheduling and register allocation in more detail; Sections 6.2 describes the three code generation strategies; Section 6.3 gives the results of the experiments; and Section 6.4 summarizes.

## 6.1 The Problem

Code efficiency is a function of the execution time of the program. To increase efficiency, a register allocator reduces memory references; an instruction scheduler reduces pipeline delays by masking the latency of multi-cycle operations. However, reducing one may increase the other. Even if a compiler is given profiling information that indicates which parts of the program are most important, a compiler that separates register allocation and instruction scheduling may be unable to find the right mix of pipeline delays and memory references, because of the lack of communication between the two phases.

Many compilers perform global register allocation before instruction scheduling [HG83, GM86]. The problem with this approach is that the same physical register may be assigned to independent pseudo-registers. If they are referenced within the same basic block, the scheduler cannot overlap operations that use them, because doing so would yield incorrect code.

Figure 6.1(a) shows a sample machine code fragment that contains two independent multiply operations. Global pseudo-registers $pr1$ and $pr2$ have lifetimes that span the code fragment. Local pseudo-registers $pr3$ and $pr4$ have lifetimes that do not overlap each other. Assume that another global pseudo-register $pr5$ is live across the code fragment, but not referenced in it. Figure 6.1(b) illustrates pseudo-register lifetimes; pseudo-registers whose lifetimes overlap interfere and cannot be assigned the same register. Therefore, given four physical registers, the register allocator is faced with a choice: force $pr3$ and $pr4$ to share a physical register or spill $pr5$ (see Figure 6.1(c)). If the register allocator makes the first choice, then the scheduler cannot overlap the two multiplies, yielding a schedule that takes 10 cycles. With the second choice, however, the scheduler can produce a schedule that takes only 6 cycles, as shown in Figure 6.1(d). If $pr5$ is rarely used and the code fragment is executed frequently, the second choice would yield better overall performance. If multiply were a single-cycle operation, however, then the two schedules would be equivalent; therefore, better performance could be achieved with the first choice, which avoids the spill. Which allocation is better? It depends on the number and size of the latencies and the degree of instruction-level parallelism within the basic block, information that is outside the register allocator's context.

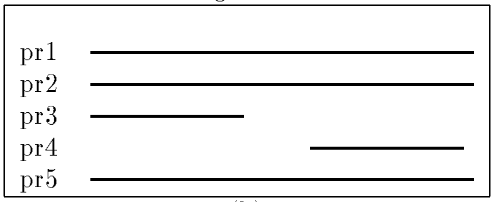
Similar problems occur if scheduling precedes register allocation. Since the scheduler's goal is to exploit instruction-level parallelism, it may increase local register usage without bound. This in turn may reduce the register allocator's opportunities to avoid memory references. Figure 6.2 illustrates this case. Two schedules for a series of four loads and two adds are shown. Assuming no

Code Fragment

| |
|---|
| $pr3 \leftarrow pr1 * pr1$ |
| store $pr3$ |
| $pr4 \leftarrow pr1 * pr2$ |
| store $pr4$ |

(a)

Pseudo-register Lifetimes




(b)

Register Allocator

| | choice 1 | choice 2 |
|---|---|---|
| pr1 | r1 | r1 |
| pr2 | r2 | r2 |
| pr3 | r3 | r3 |
| pr4 | r3 | r4 |
| pr5 | r4 | spill |

(c)

Instruction Scheduler

| Code with choice 1 | | Code with choice 2 | |
|---|---|---|---|
| Cycle | Instr | Cycle | Instr |
| 0 | $r3 \leftarrow r1 * r1$ | 0 | $r3 \leftarrow r1 * r1$ |
| 4 | store $r3$ | 1 | $r4 \leftarrow r1 * r2$ |
| 5 | $r3 \leftarrow r1 * r2$ | 4 | store $r3$ |
| 9 | store $r3$ | 5 | store $r4$ |

(d)

Figure 6.1: **Example: register allocation precedes instruction scheduling**. The code fragment is shown in (a). Pseudo-register lifetimes are shown in (b). Lifetimes that overlap cannot be assigned to the same physical register. Given 4 physical registers: if the same physical register is assigned to $pr3$ and $pr4$ (choice 1), then the multiplies cannot be overlapped; if $pr5$ is spilled (choice 2), then they can be overlapped.

cache misses and a 2-cycle load latency, the first schedule executes in one less cycle and therefore is preferred by the scheduler. However, the second sequence requires one less register and could result in a faster overall program, if using that register to hold a global pseudo-register saves more than one cycle. Again, which schedule is better depends on information outside of the scheduler's context. Because spill code introduced by the register allocator must still be scheduled, scheduling before register allocation does not completely avoid the first problem discussed in this section.

The magnitude of the problem of lack of communication between register allocation and instruction scheduling depends on the interaction between a program's instruction mix and the target processor's characteristics, such as operation latencies and register set size. On most RISCs the load latency on a cache hit is short (2 or 3 cycles) and most integer operation latencies are 1. Therefore, with programs that contain mostly integer operations, this lack of communication causes little performance degradation. With scientific workloads, however, the problem is significant for three

| Cycle | Instr |
|---|---|
| 0 | load $r1, 12(fp)$ |
| 1 | load $r2, 16(fp)$ |
| 2 | load $r3, 40(fp)$ |
| 3 | load $r4, 44(fp)$ |
| 4 | $r1 \leftarrow r1 + r2$ |
| 5 | $r2 \leftarrow r3 + r4$ |

(a)

| Cycle | Instr |
|---|---|
| 0 | load $r1, 12(fp)$ |
| 1 | load $r2, 16(fp)$ |
| 2 | load $r3, 40(fp)$ |
| 3 | $r1 \leftarrow r1 + r2$ |
| 4 | load $r2, 44(fp)$ |
| 6 | $r2 \leftarrow r3 + r2$ |

(b)

Figure 6.2: **Two schedules for independent loads and adds**. Schedule (a) takes 6 cycles and requires 4 registers. Schedule (b) takes 7 cycles and requires 3 registers. Which is better depends on how the extra register could be used.

reasons. First, these workloads tend to have a high degree of instruction-level parallelism, because they contain large basic blocks with many independent expressions. Second, they use many multi-cycle floating point operations, which can cause long pipeline delays. Avoiding these delays requires exploiting that parallelism. Third, the large basic blocks contain many local pseudo-registers, as well as global pseudo-registers; the increased number of register allocation choices exacerbates the lack of coordination in a simple code generation strategy.

## 6.2 Code Generation Strategies

A number of code generation strategies separate register allocation from instruction scheduling. Hennessy and Gross's method [HG83, Gro83] and Gibbons and Muchnick's method [GM86] perform register allocation before basic block instruction scheduling. Both schedulers use list scheduling algorithms with heuristics to choose among scheduling candidates. The primary difference between the two is in dealing with a physical register that is used for two independent computations within a block. Gibbons and Muchnick's method adds edges to the code DAG to force an ordering between the two computations; the scheduler itself is $\mathcal{O}(n^2)$, where $n$ is the number of nodes in the code DAG. Hennessy and Gross's method allows the scheduler to reorder the computations, which sometimes yields better code; however, this increases the scheduling complexity to $\mathcal{O}(n^4)$.

A method by Goodman and Hsu [GH88, Hsu87] performs DAG-driven local register allocation before instruction scheduling. The register allocator examines the code DAG during allocation. If two pseudo-registers must share the same physical register, extra edges are added to the DAG, as in Gibbons and Muchnick's method. The difference is that the register allocator attempts to limit its effect on the scheduler: when a pseudo-register is to be assigned a physical register, the register allocator first attempts to choose a register such that any additional DAG edges are redundant; failing that, it chooses a register that will minimize the increase in the height of the DAG. (The *height* of a DAG is maximum distance along a path from any root to any leaf.)

Goodman and Hsu [GH88, Hsu87] also developed an approach called Integrated Prepass Scheduling, which performs scheduling before local register allocation. Section 6.2.2 describes this approach in more detail.

Auslander and Hopkins [AH82] perform instruction scheduling first, followed by register allocation and another scheduling pass. Warren [War90] builds upon this method with some heuristics for limiting the instruction scheduler's effect on the register allocator. In particular, Warren's method decreases the priority of loads as register pressure increases.

### 6.2.1 Postpass Scheduling

The Postpass strategy cleanly separates register allocation and instruction scheduling. Global register allocation is performed first, using a Chaitin-style graph coloring allocator [Cha82, BCKT89]. Instruction scheduling follows (hence the name Postpass), using an $\mathcal{O}(n^2)$ list scheduling algorithm, similar to Gibbons and Muchnick's [GM86]. The list scheduler builds the code DAG and keeps two lists: the *ready list* contains instructions that can be scheduled without causing a delay; the *leader list* contains instructions whose predecessors have been scheduled, but would cause a data hazard if scheduled. Instructions are scheduled from the ready list according to the following heuristics:

(1) choose the ready instruction with the maximum distance; if not unique, then

(2) choose from that set the instruction with the maximum number of successors; if not unique, then

(3) choose from that set the instruction with the greatest latency; if not unique, then

(4) choose an arbitrary instruction from that set.

Section 4.2 provides a more detailed description of list scheduling algorithms and Section 3.7 discusses global register allocation.

#### Register Coloring Policies

Hennessy and Gross [HG83] suggested that a round-robin register allocation policy would have less impact on a postpass scheduler than a first-fit policy. Goodman and Hsu's results [GH88] confirm this. In both compilers round-robin was used only for local register allocation.

I compared four policies to determine the effects of a global register allocator's coloring policy on the Postpass scheduler.

(1) *First-fit* chooses the first free register (or register pair).

(2) *Round-robin* begins searching for a free register from the point where the last successful search ended.

(3) *Mixture* combines the first two policies: Round-robin is used over the first half of the register set; if a free register is not found, First-fit is used over the second half.

(4) With *Basic-block*, the idea is to avoid using the same physical register more than once within a basic block, thereby attempting to minimize introduced dependences. For pseudo-register $p$, *range*$(p)$ is defined to be the set of all basic blocks in which $p$ is live. For color $c$, *range*$(c)$ is the set of the blocks in which the color is used. *Range*$(c)$ is updated as pseudo-registers are colored. When a pseudo-register $p$ is to be colored, the policy chooses the color $c$ such that $|\text{range}(p) \cap \text{range}(c)|$ is the minimum over all free colors.

The Round-robin policy tends to spread out register usage, but, since the order in which pseudo-registers are colored is based on priority, not proximity, pseudo-registers in the same vicinity may

Table 6.1: **Comparison of register coloring policies**. All values are harmonic means of the execution times, normalized to First-fit, of the programs within each group. The best policy choice for each group is shown in bold. (Smaller values are better.)

| | Test | First-fit | Round-robin | Mixture | Basic-block |
|---|---|---|---|---|---|
| 88000 | Livermore | 1.00 | **0.79** | 0.80 | 0.80 |
| | Nasker | 1.00 | **0.89** | 0.91 | 0.97 |
| | Perfect | 1.00 | **0.91** | 0.92 | 0.96 |
| | Misc | 1.00 | 0.96 | **0.93** | 0.97 |
| | Integer | **1.00** | 1.13 | 1.02 | 1.01 |
| R2000 | Livermore | 1.00 | 0.96 | **0.92** | 0.93 |
| | Nasker | 1.00 | **0.94** | 0.95 | 0.97 |
| | Perfect | 1.00 | 0.93 | **0.92** | 0.95 |
| | Misc | 1.00 | 0.95 | 0.92 | **0.91** |
| | Integer | **1.00** | 1.14 | 1.05 | 1.01 |
| i860 | Livermore | 1.00 | **0.84** | 0.90 | 0.88 |
| | Nasker | 1.00 | **0.83** | 0.88 | 0.86 |
| | Perfect | 1.00 | **0.86** | 0.88 | 0.90 |
| | Misc | 1.00 | 0.90 | 0.90 | **0.87** |
| | Integer | **1.00** | 1.10 | 1.01 | 1.01 |

still be assigned the same physical register. The Basic-block policy more precisely spreads out register usage. However, after all colors have been used once in a large block, the policy in effect reverts to First-fit. Therefore, it does not work well on basic blocks in which many pseudo-registers are live.

Table 6.1 shows the results of the comparison in terms of execution time normalized to First-fit. Each value in the table is the harmonic mean of the normalized execution times for a particular program group.

Except on the Integer group, First-fit is consistently worse than the other policies. First-fit is the best on the Integer group; the other policies use more registers, which increases procedure call overhead, but provides little scheduling benefit.

On the four floating-point intensive program groups, both Round-robin and Mixture are better than Basic-block. Mixture is the best on the R2000, whereas Round-robin is the best on the i860. On the 88000 Round-robin is the best on three of the four groups, but it is better than Mixture on those groups by only 1%.

Considering both floating point and integer programs, Mixture is the best choice. Nevertheless, I use Round-robin in the Postpass strategy in all further experiments. The main reason is that I added Mixture to the comparison after many of the experiments were conducted. Since Mixture has an advantage over Round-robin only on integer programs, I decided not to rerun the experiments.

### 6.2.2 Integrated Prepass Scheduling

Marion's IPS is a variant of Goodman and Hsu's original Integrated Prepass Scheduling [GH88, Hsu87]. In their version the scheduler precedes the register allocator, but attempts to restrict the number of concurrently live local pseudo-registers by giving each basic block a register limit. A *register limit* places an upper bound on the number of live local pseudo-registers, thus limiting the amount of instruction-level parallelism that the scheduler may exploit. The scheduler selects instructions to minimize pipeline delays, in a manner similar to the Postpass scheduler, unless the number of live local pseudo-registers is greater than or equal to the given limit. As long as the scheduler is at the limit, it attempts to schedule instructions that free registers; if it cannot, it may exceed the limit. After scheduling, the register allocator assigns physical registers to the local pseudo-registers within the block; spills to memory are inserted if the limit could not be met. In their study, Goodman and Hsu assumed that "important" global pseudo-registers, such as induction variables and loop invariants, are assigned physical registers before scheduling, so that their IPS considered only local register allocation.

Goodman and Hsu compared their version of IPS to a number of methods including a version of Postpass. All methods assumed that global pseudo-registers received physical registers *a priori*. When given 15 registers for local register allocation, on a workload consisting of the Livermore Loops, IPS achieved speedups of roughly 1.6 over Postpass with a first-fit allocation policy and 1.3 over Postpass with a round-robin policy, on a long pipeline machine (based on the Cray-1). The speedups were 1.2 and 1.1, respectively, on a medium pipeline machine. There was no speedup over the strategies other than Postpass for 15 local registers. Given 30 local registers, IPS exhibited virtually no speedup over any strategy.

The context of Goodman and Hsu's experiment was limited by several factors. First, they allocated global pseudo-registers (essentially by hand) before running IPS and performed only local register allocation after scheduling. For large programs, global register allocation is necessary to effectively use registers over many basic blocks. Second, spills were scheduled by the local register allocator, using a simple scheme. The scheme does not reorder instructions, but instead attempts to place a load enough instructions before the reference to the loaded register to avoid the load latency; it also attempts to place a store enough instructions after the reference to avoid any operation latency. When register pressure is high, this scheme will have difficulty avoiding that latency. Third, structural hazards were not considered. On some RISCs there may be contention for a non-pipelined functional unit or a register write-back bus. Fourth, their workload was limited to 12 Livermore Loops, with some loop unrolling.

Marion's version of IPS addresses all of these limitations. First, to model reserving physical registers for important global pseudo-registers, Marion's IPS sets the local register limit for each block as follows:

$$limit = \max(3, avail - global\_used), \tag{6.1}$$

where *avail* is the maximum number of registers available to the register allocator and *globals_used* is the number of unique global pseudo-registers referenced within the block. After scheduling, the new instruction order (produced by the scheduler) is used by the register allocator to compute interferences. The register allocator is identical to the Postpass allocator, except that the First-fit coloring policy is used, since scheduling precedes register allocation. Second, the Postpass scheduler is invoked after register allocation to ensure that spill code is scheduled as well as possible. Third, the scheduler takes structural hazards into account when scheduling code. Fourth, the experiments are performed on a workload that includes complete programs, in addition to the Livermore Loops and the NAS kernels.

The IPS strategy represents an intermediate level of integration between register allocation and instruction scheduling. In effect it "reserves" registers for global pseudo-registers by reducing the register limit by the number of globals referenced in the block. In addition, it restricts local register use by imposing register limits on scheduling. Together these impose constraints on the scheduler that mimic those that the register allocator must face.

### 6.2.3 RASE Code Generation

RASE provides its register allocator with schedule cost estimates that allow the allocator to quantify the effect of its choices on the scheduler. This is in contrast to IPS, which uses a heuristic to attempt to minimize the scheduler's impact on the register allocator. There are two aspects to the solution used in RASE: (1) a pre-scheduler (PRESCHED) is invoked before the global register allocator (GRA) to gather schedule cost information that is then used by GRA; and (2) the responsibility for allocating registers is distributed; the instruction scheduler (FINALSCHED) allocates registers to local pseudo-registers and GRA allocates registers only to global pseudo-registers.

As with all of the strategies, a simple code selector generates straightforward machine code that defines and uses pseudo-registers. RASE's three phases then follow. (See Figure 6.3.)

First, PRESCHED computes a schedule cost function for each basic block. Given a register limit, the schedule cost function returns a schedule cost estimate. A *schedule cost estimate* is the estimated number of machine cycles required to execute the instructions in a basic block, while remaining within the register limit. In contrast to IPS's register limits, RASE's limits must not be exceeded.

In the second phase, GRA uses the schedule cost estimates to compute the incremental cost of reducing each register limit. The *incremental cost* reflects the additional cycles required to schedule a block with a reduced upper bound on the parallelism. The incremental costs are in the same units (machine cycles) as the costs associated with spilling global pseudo-registers to memory. GRA uses both the incremental and spill costs to assign physical registers to global pseudo-registers and to decide what register limit to impose on each block.

The third phase, FINALSCHED, schedules each block, assigning physical registers to local pseudo-registers within the register limit for that block. Both PRESCHED and FINALSCHED use an underlying scheduler (SCHED).

This approach addresses the code generation communication problem by giving GRA responsibility for determining how many registers to use, both for exploiting instruction-level parallelism and for reducing memory references. In essence, for each basic block, GRA partitions the register set into two portions. One portion is given to the instruction scheduler, in the form of the register limit, to exploit instruction-level parallelism. GRA itself assigns the registers in the other portion to the procedure's most important global pseudo-registers, thereby reducing memory references. Based on both the spill and incremental costs, GRA determines the appropriate balance between these two competing needs for registers.

#### Schedule Cost Estimates

For each block $b$, PRESCHED constructs a schedule cost function,

$$schedcost_b(x) = c + d/x^2, \tag{6.2}$$

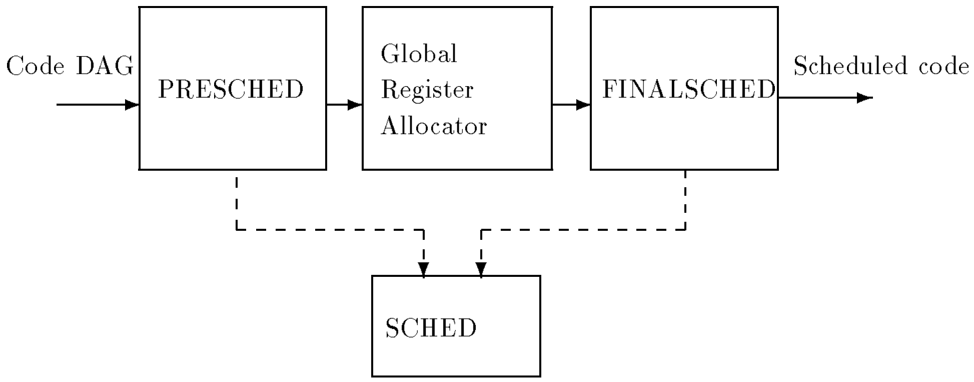



Figure 6.3: **RASE structure**. PRESCHED calls SCHED to gather schedule cost information. The global register allocator assigns registers to global pseudo-registers and establishes register limits. FINALSCHED calls SCHED to perform actual scheduling within the register limits.

that returns the estimated number of machine cycles needed to execute block $b$ with a register limit of $x$. PRESCHED constructs the function by twice invoking the underlying scheduler (SCHED) to obtain two points on the function's curve and then uses the points to compute the coefficients $c$ and $d$. On the first invocation, SCHED is given a small register limit, typically 3. On the second invocation SCHED is given the maximum register limit, usually the total number of registers. SCHED returns an ordered pair *(regs,schedcost)*, where *schedcost* is the schedule cost, in machine cycles, and *regs* is the maximum number of live local pseudo-registers at any point within the basic block. The value, *regs*, gives PRESCHED an upper bound on the instruction-level parallelism within a block. It will be less than the given register limit if SCHED was unable to exploit enough parallelism to reach the limit, but can never be greater. *Schedcost* takes into consideration aspects of the target architecture that are known at compile time, such as operation latencies and structural hazards.

If the architecture has more than one register set, a schedule cost function is constructed for each set. A total of $1 + k$ invocations of SCHED is required, where $k$ is the number of distinct general purpose register sets in the target architecture, one for the minimum plus a maximum for each set. For most RISCs, $k$ is 1 or 2. These calls to SCHED do not change the instruction order.

RASE constructs a schedule cost function for two reasons: (1) calling SCHED for each register limit would be prohibitively expensive and (2) since GRA may introduce spill code for global pseudo-registers, additional calls to SCHED would not necessarily yield more precise estimates. I conjectured that increasing the register limit would decrease the schedule cost rapidly near the minimum limit and slowly near the maximum. To test this hypothesis, the cost estimates were computed for each point for each block in the NAS kernels on the 88000. The average correlation coefficient between the points and the inverse quadratic function was 0.88. In most cases the schedule cost function decreases more rapidly than the curve of computed cost estimates. Additional tuning could yield a better function. Figure 6.4 shows the curves for the first two large blocks in the first kernel.

Figure 6.4: **Example schedule cost function** for a basic block returns the estimated number of machine cycles to execute the block, given a register limit. This graph shows the schedule costs for two basic blocks, computed by calling SCHED for each register limit, and the schedule cost function for Block 2. The schedule cost function for Block 1 is nearly identical, so it is not shown.

**Global Register Allocation**

RASE's global register allocator differs from the Chaitin-style allocator [Cha82] used by Postpass and IPS. Chaitin's register allocator includes nodes in the interference graph that represent both global and local pseudo-registers. In RASE, the nodes for local pseudo-registers are replaced with nodes representing basic block pseudo-registers. (See Figure 6.5.) The number of basic block pseudo-registers for each block is the maximum number of live local pseudo-registers at any point (as determined by PRESCHED). For each basic block the number of basic block pseudo-registers that get colored becomes the register limit for that block. In effect, the responsibility for local register allocation is replaced by the responsibility for determining the register limits.

As in Chaitin's register allocator, the RASE allocator colors nodes in decreasing order of priority. For each node $p$,

$$priority_p = \frac{regcost_p}{degree_p}. \tag{6.3}$$

(See Section 3.7 for a detailed description of Chaitin's allocator.) For global pseudo-register nodes $regcost_p$ is the cost of spilling $p$ to memory. RASE calculates this in the same manner as Chaitin's allocator. For basic block pseudo-register nodes, $regcost_p$ represents the incremental cost of reducing the register limit by one. This is calculated as follows. First, for each block $b$, let $n$ be the number of basic block pseudo-register nodes for $b$. Each such node $p$ is given a value between 1 and $n$ that represents a possible register limit for $b$. Then, for each node $p$, with value $x$, $regcost_p$ is the additional cost of the schedule that would result if the block's register limit were reduced from $x$ to $x-1$. Thus,

$$regcost_p = (schedcost_b(x-1) - schedcost_b(x)) * freq_b, \tag{6.4}$$

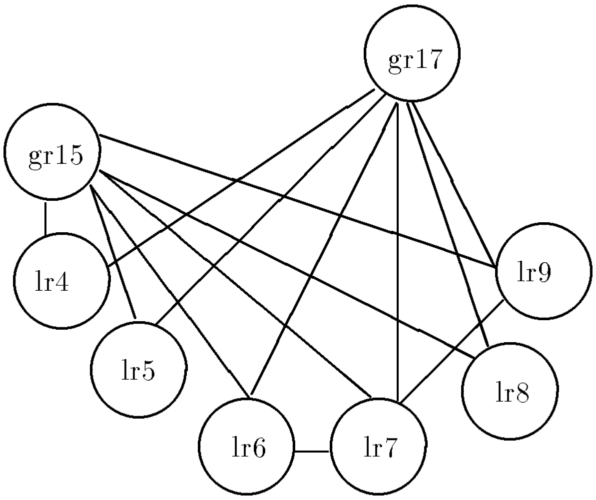
where $schedcost_b(x)$ is the schedule cost function for $b$ and $freq_b$ is the frequency with which $b$ is executed.

To force GRA to color a minimum number of basic block pseudo-register nodes for each block, $regcost_p$ is set to infinity for those nodes. For each block, this minimum is the maximum number of pseudo-register source operands in any single instruction in the block.

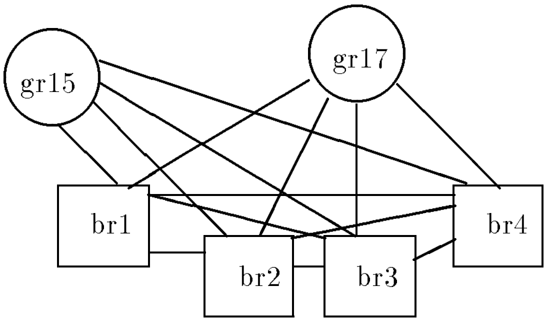
After coloring the graph, GRA counts the number of basic block pseudo-register nodes that were colored, for each basic block. This number becomes the register limit for that block.

Since RASE's register allocation still precedes scheduling, the First-fit coloring policy may constrain the scheduler. The Round-robin policy alleviates the problem somewhat, but may still cause scheduling constraints. The Basic-block policy did not work well with Postpass, because it was overloaded by local pseudo-registers. The RASE allocator, however, allocates registers only to global pseudo-registers, which enables the Basic-block policy to perform better than Round-robin. Consequently, the RASE register allocator uses the Basic-block coloring policy.

As discussed in Section 3.7.1, the Marion global register allocator constructs interference graph nodes that represent register pairs and requires them to be colored with adjacent physical registers. This applies to basic block pseudo-registers as well. In particular, within the group of basic block pseudo-registers that do not have infinite cost, all of the nodes representing pairs precede those representing single registers. Thus, the pairs have a higher *regcost* than the singles and, therefore, will be candidates for coloring first.




(a) The graph in a Chaitin-style register allocator contains a node for each global and local pseudo-register. Some local pseudo-registers may interfere, while others may not. The instruction order presented to the register allocator determines the interference.




(b) The graph in the RASE register allocator contains 4 basic block pseudo-register nodes, because out of the 6 local pseudo-registers, at most 4 are live at any point in the block. All such nodes conflict, so that no two get the same color. The number that do get colored becomes the register limit for this block.

Figure 6.5: **Interference graphs for a basic block in Chaitin-style and RASE register allocators.** Assume that (1) global pseudo-registers $gr15$ and $gr17$ are live throughout the block, (2) the basic block contains 6 local pseudo-registers and (3) of those 6, at most 4 are live at any point in the block. (lr = local pseudo-register, br = basic block pseudo-register, gr = global pseudo-register).

The two groups of nodes representing basic block pseudo-register pairs and singles, respectively, are further subdivided into calleesave and non-calleesave subgroups. Within each group the calleesave subgroup precedes the non-calleesave subgroup. PRESCHED indicates how many local pseudo-registers (both pairs and singles) are live across a call; these become the size of the calleesave groups. Nodes in these groups interfere with callersave physical registers, forcing them to be colored with calleesave physical registers (if at all). If calleesave physical registers are not available, but callersave register are, it is sometimes profitable to give callersave registers to a basic block, even though it would prefer calleesave. To enable this to happen, for each calleesave basic block pseudo-register, an extra non-calleesave basic block pseudo-register is added. These extra pseudo-registers are given low *regcosts*, beginning at $.001 * freq$ and decreasing for each subsequent one.

### The Underlying Scheduler (SCHED)

The algorithm for SCHED, shown in Figure 6.6, is $\mathcal{O}(n^2)$ and is similar to the IPS scheduler's algorithm. Significant differences are that SCHED performs local register allocation while scheduling and strictly adheres to the register limit, spilling if necessary. In contrast, the IPS scheduler may exceed the limit, since it precedes register allocation.

When a register limit is reached, SCHED, like IPS, attempts to schedule an instruction that frees registers in the limited register set or at least creates no more registers than it frees. However, if no such instruction exists, SCHED spills the local pseudo-register $r$ from the limited set such that

$$h(r) \leq h(s),\ \forall s \in \text{regs},\ \text{where}\ h(r) = \max_{u \in use(r)} (c_u) \tag{6.5}$$

and $c_u$ is the maximum cumulative cost of $u$. In other words, SCHED chooses the register whose farthest use from the end of the block is closer to the end of the block than any other register's. When spilling, SCHED schedules the store of the spilled register $r$ and then adds the corresponding load instruction to the code DAG, ensuring that it precedes all unscheduled uses of $r$. Spilling is sometimes necessary because SCHED performs local register allocation, along with scheduling.

SCHED's heuristics for choosing a ready instruction are a superset of Postpass's heuristics:

(1) choose the ready instruction whose maximum distance[1] is within *tolerance* of the greatest maximum distance of any instruction; if not unique, then

(2) choose from that set the instruction that is not a local load instruction; if not unique, then

(3) choose from that set the instruction with the maximum distance; if not unique, then

(4) choose from that set the instruction with the maximum number of successors; if not unique, then

(5) choose from that set the instruction with the greatest latency; if not unique, then

(6) choose an arbitrary instruction from that set.

[1] *Maximum distance* is defined in Section 4.2.

**Algorithm 4:** *schedule_within_limit*

```
Input: code DAG; foreach register set s, a register limit, limit(s)
Output: schedule
Define:
    c is the current cycle, initially 0
    R is the set of ready nodes, initially = { DAG roots }
    L is the set of leader nodes (ready except for outstanding latency), initially empty
    foreach register set s, use(s) is current number of registers in use, initially 0
    foreach node x and foreach register set s,
        net(x, s) is the net number of s registers created by n
    foreach node x ∈ L, lat(x) is the remaining latency of x
Method:
    while R ≠ ∅ and L ≠ ∅
        if R ≠ ∅ then
            y = ⊥
            if foreach s, use(s) < limit(s) then
                y = choose x ∈ R, using heuristics
            else
                let s be a register set such that use(s) ≥ limit(s)
                if ∃ x ∈ R such that net(x, s) < 0 then
                    y = choose best such x, using heuristics
                elseif ∃ x ∈ R such that net(x, s) = 0 then
                    y = choose best such x, using heuristics
                elseif ∃ x ∈ L such that net(x, s) <= 0 then
                    y = NOP
                else spill a register from s
                endif
            endif
        else y = NOP
        endif
        if y ≠ ⊥ then
            schedule y
            foreach successor x of y
                if all predecessors of x are scheduled then
                    L = L ∪ {x}; lat(x) = remaining latency of x's predecessors
                endif
            endfor
        endif
        c = c + 1
        foreach x ∈ L
            decrement lat(x)
            if lat(x) = 0 then L = L − x; R = R ∪ x; endif
        endfor
    endwhile
```

Figure 6.6: **Algorithm for the RASE scheduler SCHED.**

The tolerance is at most 8 and is a function of the number of currently live local pseudo-registers; as the number approaches the limit, the tolerance increases. The tolerance used in the first heuristic, along with the second heuristic, allows the scheduler more freedom to find an instruction that does not increase the number of live local pseudo-registers. The latter four heuristics are the same as those use by Postpass.

Allocating local registers on-the-fly creates two problems that complicate SCHED and sometimes cause it to spill unnecessarily. First, considerable effort is needed to avoid reloading a spilled register when it cannot be used. To accomplish this, SCHED marks each reload and will not schedule it unless some successor has no predecessors other than reloads. This avoids reloading too early, but sometimes forces reloading to occur too late. For example, in a large basic block with many local common subexpressions, those subexpressions may be spilled after their initial uses. Then, the latter part of the schedule may contain many load delays, because only reloads and their dependent operations are left to schedule. It is fairly common to find a small number of these reloads in a large basic block; the problem is serious only in a few large blocks.

The second problem concerns function calls. During pre-scheduling, the scheduler notes how many local pseudo-registers are live across each call site in a basic block. The global register allocator uses this information to try to give the basic block enough calleesave registers to meet its needs. However, until it schedules a call, SCHED does not know which pseudo-registers will be live across the call.

An alternative that may solve the second problem is to delay register assignment until after scheduling. Since the scheduler remains within the register limit, spilling should not be necessary; some moves between registers may be needed, however, because of specific register needs for parameter passing, calleesave and callersave registers, and register pairs.

A potential solution to both problems is to allow SCHED to exceed the register limit, like the IPS scheduler, and then do local register allocation after scheduling. Of course, spills would have to be inserted, requiring them to be scheduled. Nevertheless, this would simplify SCHED and could improve both compile-time performance and the schedules produced.

The presence of two or more registers sets causes another problem for SCHED (and for the IPS scheduler as well). For example, if one set (*e.g.*, floating point) is at its register limit, but there are ready instructions that use only the other set (*e.g.*, integer), SCHED will schedule them. Eventually a floating point register may have to be spilled anyway. If the integer instructions could have been overlapped with floating point instructions, the resulting schedule is inferior. Preventing this is difficult, because some of those integer instructions may be predecessors of floating point instructions; therefore, scheduling the integer instructions may avoid the floating point register spill. Fortunately, this problem is rare; the only significant case observed was on the Nasker program VPE on the i860. Section 6.3 discusses this problem further.

## 6.3 Results

The comparison of code generation strategies was performed on three commercially available RISC architectures, the MIPS R2000, the Motorola 88000 and the Intel i860. These architectures are discussed in Chapter 2.

All data given in this section are speedups. (Larger numbers are better.) The speedup of

Table 6.2: **Speedup of IPS and RASE over Postpass for the Livermore group** on three architectures. (Larger numbers are better.)

| | Speedup over Postpass | | | | | |
|---|---|---|---|---|---|---|
| | 88000 | | R2000 | | i860 | |
| Test | IPS | RASE | IPS | RASE | IPS | RASE |
| Ker1 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 |
| Ker2 | 1.19 | 1.19 | 1.06 | 1.06 | 0.99 | 1.00 |
| Ker3 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 |
| Ker4 | 1.01 | 1.01 | 1.01 | 1.01 | 1.01 | 1.01 |
| Ker5 | 1.00 | 1.00 | 1.00 | 1.00 | 0.96 | 1.00 |
| Ker6 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 |
| Ker7 | 1.59 | 1.57 | 1.32 | 1.36 | 1.00 | 1.00 |
| Ker8 | 1.26 | 1.24 | 1.16 | 1.17 | 1.08 | 1.10 |
| Ker9 | 1.21 | 1.23 | 1.17 | 1.16 | 1.02 | 1.02 |
| Ker10 | 1.19 | 1.19 | 1.17 | 1.17 | 1.19 | 1.19 |
| Ker11 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 |
| Ker12 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 |
| Ker13 | 1.16 | 1.16 | 1.08 | 1.08 | 1.01 | 1.14 |
| Ker14 | 1.07 | 1.08 | 1.08 | 1.08 | 0.96 | 1.04 |
| HMn | 1.10 | 1.10 | 1.07 | 1.07 | 1.01 | 1.03 |

strategy $x$ (IPS or RASE) over Postpass is,

$$Speedup = \frac{total\_cycles(Postpass)}{total\_cycles(x)}. \tag{6.6}$$

Table 6.2 shows the speedups of IPS and RASE over Postpass on the three architectures, for the Livermore group. In virtually all cases, IPS and RASE produce faster code. On the 88000, both IPS and RASE achieve speedups over Postpass of up to 1.6. On the R2000, the speedups are smaller, but on four loops they exceed 1.15. The speedups for the Nasker group are shown in Table 6.3. On the 88000, both IPS and RASE achieve speedups averaging 1.2 (for the unoptimized kernels). Again the speedups are smaller on the R2000.

The greater speedups of IPS and RASE over Postpass on the 88000 versus the R2000 are caused by three differences between the architectures. First, the 88000 has only 32 registers, while the R2000 has 32 integer registers and 16 floating point registers. Given more registers, the register allocator is less likely to constrain the Postpass scheduler. Second, the 88000 floating point operation latency is roughly twice that of the R2000. Since hiding shorter latencies requires less instruction-level parallelism, scheduling constraints imposed by the register allocator have a smaller impact. Third, a double precision load/store takes 2 instructions on the R2000, but only 1 on the 88000 (with a 3-cycle load latency). The additional instructions give schedulers more possibilities for hiding latencies, thereby reducing the impact of scheduling constraints.

Livermore kernel 10 is an interesting case. It contains only double precision subtracts, loads and stores. Since double subtract latency is only two on the R2000, we expected IPS and RASE to exhibit little speedup over Postpass on that architecture. Postpass's Round-robin coloring policy, however, forces an important value that is used in the inner loop to be spilled, yielding an inferior schedule. Although the Round-robin policy yields better code than First-fit for most programs, Round-robin can sometimes produce poor results.

The average speedup of IPS and RASE over Postpass is smaller on the i860 than either the 88000 or R2000, because the i860 has more registers (32 integer and 32 single precision floating point). Nevertheless, the results are consistent with those from the other architectures: IPS and RASE are better than Postpass. The speedups vary widely and in some cases IPS and RASE produce worse code than Postpass. Much of the variation is attributed to the difficulty of scheduling the explicitly advanced pipelines on the i860; each of the schedulers must be restricted to ensure that intermediate values in the pipelines are not lost. Modifications to the RASE scheduler might reduce the variation; this area needs further study.

On the i860, the NAS kernel VPE exhibits the difficultly of scheduling with register limits on a machine with two register sets. Both IPS and RASE produce worse code than Postpass. Each reaches the floating point register limit and then continues to schedule integer instructions that could have been overlapped with floating point instructions, if a floating point register had been spilled or the limit surpassed.

No global optimizations are performed by the Marion's Lcc front end. The lack of these optimizations can affect the results. To gauge this effect, I hand-optimized the Nasker program group. In general, while execution time improved over the unoptimized code, the relative performance across code generation strategies remained comparable to that produced by the unoptimized code.

Table 6.3 contrasts the unoptimized and hand-optimized Nasker group. The hand-optimized kernels were hand-optimized using loop induction variable analysis and strength reduction and loop invariant removal. These optimizations change a program's instruction mix, generally reducing integer addressing code, including integer multiplies, but increasing integer pseudo-registers to hold compiler-created strength-reduced induction variables. On the optimized kernels, IPS and RASE exhibit speedups over Postpass comparable to those on the unoptimized code. On the optimized programs, the speedups are somewhat lower on the 88000, but higher on the R2000 and i860. The lower speedups on the 88000 result from fewer integer multiplies in the optimized code; integer multiplies contend with floating point instructions for the FPU. Fewer integer multiplies are also the reason for the higher speedups on the R2000. Because the R2000 integer multiply has a long latency and is not pipelined, the integer multiply/divide unit is the critical resource on the unoptimized code; when optimizations reduce those multiplies, the registers become the critical resource. Because IPS and RASE more effectively use registers, they produce better schedules than Postpass on the optimized code.

Tables 6.4 and 6.5 show the speedups for the Perfect and Misc groups, respectively. On these programs both IPS and RASE exhibit smaller speedups over Postpass than on the kernels: IPS speedups ranged from 1.0 to 1.2; for RASE they ranged from 1.0 to nearly 1.3. In most of the programs, execution time is distributed over many basic blocks, as opposed to the Livermore and Nasker groups. Since the level of scheduling difficulty varies considerably between blocks, the potential speedup over Postpass is smaller for the programs than for the kernels. Nevertheless, some applications exhibit significant speedups on the 88000 and good speedups on the R2000. In both ARC2D and MG3D, which exhibit some of the largest speedups, the dominating blocks contain large numbers of floating point operations of all kinds.

On the 88000, RASE exhibits speedups over IPS of 7% to 13% on four of the Perfect group programs. This occurs because IPS decreases a block's local register limit by the number of global

Table 6.3: **Speedup of IPS and RASE over Postpass for the Nasker group** (a) unoptimized and (b) optimized, on three architectures. No global optimizations were performed on the unoptimized kernels, (the same as on all other programs). Loop induction variable analysis and strength reduction and invariant removal were performed by hand on the optimized kernels.

| | Speedup over Postpass | | | | | |
|---|---|---|---|---|---|---|
| | 88000 | | R2000 | | i860 | |
| Test | IPS | RASE | IPS | RASE | IPS | RASE |
| BTR | 1.24 | 1.25 | 1.23 | 1.24 | 1.17 | 1.20 |
| CHO | 1.18 | 1.18 | 1.03 | 1.03 | 1.05 | 1.03 |
| EMI | 1.17 | 1.18 | 1.02 | 1.03 | 1.06 | 1.08 |
| FFT | 1.33 | 1.33 | 1.04 | 1.03 | 0.96 | 1.01 |
| GMT | 1.10 | 1.10 | 1.00 | 1.00 | 1.00 | 0.97 |
| MXM | 1.31 | 1.33 | 1.06 | 1.06 | 1.05 | 1.13 |
| VPE | 1.17 | 1.17 | 1.08 | 1.08 | 0.97 | 0.91 |
| HMn | 1.21 | 1.21 | 1.06 | 1.07 | 1.03 | 1.04 |

(a) Unoptimized Nasker group

| | Speedup over Postpass | | | | | |
|---|---|---|---|---|---|---|
| | 88000 | | R2000 | | i860 | |
| Test | IPS | RASE | IPS | RASE | IPS | RASE |
| BTR | 1.20 | 1.21 | 1.19 | 1.21 | 1.14 | 1.15 |
| CHO | 1.07 | 1.08 | 1.08 | 1.08 | 1.08 | 1.08 |
| EMI | 1.09 | 1.12 | 1.09 | 1.10 | 1.01 | 1.02 |
| FFT | 1.22 | 1.23 | 1.01 | 1.01 | 1.09 | 1.11 |
| GMT | 1.04 | 1.04 | 1.00 | 1.00 | 1.00 | 1.00 |
| MXM | 1.29 | 1.29 | 1.19 | 1.21 | 1.11 | 1.11 |
| VPE | 1.14 | 1.14 | 1.08 | 1.08 | 0.97 | 0.91 |
| HMn | 1.14 | 1.15 | 1.08 | 1.09 | 1.05 | 1.05 |

(b) Hand-optimized Nasker group

pseudo-registers referenced within the block, which in effect favors global pseudo-registers at the expense of locals. Most of the time this heuristic is effective. However, for blocks that contain more global pseudo-registers than physical registers (which is the case with these four programs), the register limit is reduced to the minimum, preventing the scheduler from exploiting much instruction-level parallelism. RASE, on the other hand, finds a better balance between using registers for scheduling and using them to reduce memory references. This results in more physical registers being used locally for scheduling, which yields better schedules for these large blocks.

On workloads that have small basic blocks or mostly integer operations, IPS and RASE both perform better than Postpass, as shown in Table 6.6. The speedups result primarily from Postpass's Round-robin register allocation policy, which causes a higher than necessary procedure call overhead. If Postpass used the Mixture policy, the speedups would be significantly smaller.

Table 6.4: **Speedup of IPS and RASE over Postpass for the Perfect group** on three architectures.

| | Speedup over Postpass | | | | | |
|---|---|---|---|---|---|---|
| | 88000 | | R2000 | | i860 | |
| Test | IPS | RASE | IPS | RASE | IPS | RASE |
| ARC2D | 1.16 | 1.25 | 1.06 | 1.06 | 1.13 | 1.16 |
| BDNA | 1.04 | 1.18 | 1.11 | 1.15 | 0.99 | 1.01 |
| DYFESM | 1.11 | 1.11 | 1.03 | 1.03 | 1.06 | 1.06 |
| FLO52 | 1.11 | 1.11 | 1.04 | 1.04 | 1.08 | 1.12 |
| MDG | 1.05 | 1.05 | 1.02 | 1.01 | 1.04 | 1.03 |
| MG3D | 1.18 | 1.28 | 1.05 | 1.06 | 1.05 | 1.08 |
| QCD | 1.02 | 1.10 | 1.04 | 1.02 | 1.05 | 1.04 |
| TRACK | 1.02 | 1.06 | 1.02 | 1.05 | 1.03 | 1.02 |
| HMn | 1.08 | 1.14 | 1.05 | 1.05 | 1.05 | 1.06 |

Table 6.5: **Speedup of IPS and RASE over Postpass for the Misc group** on three architectures.

| | Speedup over Postpass | | | | | |
|---|---|---|---|---|---|---|
| | 88000 | | R2000 | | i860 | |
| Test | IPS | RASE | IPS | RASE | IPS | RASE |
| BOAST | 1.12 | 1.13 | 1.10 | 1.10 | 1.08 | 1.11 |
| SPHOT | 1.09 | 1.10 | 1.10 | 1.08 | 1.10 | 1.11 |
| WANAL1 | 1.15 | 1.17 | 1.08 | 1.08 | 1.05 | 1.06 |
| NEURAL | 1.24 | 1.24 | 1.15 | 1.14 | 1.11 | 1.11 |
| HMn | 1.15 | 1.16 | 1.11 | 1.10 | 1.09 | 1.10 |

Table 6.6: **Speedup of IPS and RASE over Postpass for the Int group** on three architectures.

| | Speedup over Postpass | | | | | |
|---|---|---|---|---|---|---|
| | 88000 | | R2000 | | i860 | |
| Test | IPS | RASE | IPS | RASE | IPS | RASE |
| DHRYSTONE | 1.07 | 1.04 | 1.12 | 1.05 | 1.07 | 1.03 |
| INTEGER | 1.07 | 1.07 | 1.05 | 1.05 | 1.12 | 1.09 |
| KNIGHT | 1.25 | 1.24 | 1.17 | 1.16 | 1.16 | 1.17 |
| LCC | 1.39 | 1.35 | 1.39 | 1.30 | 1.35 | 1.29 |
| SPARSE | 1.23 | 1.21 | 1.24 | 1.19 | 1.22 | 1.19 |
| HMn | 1.19 | 1.17 | 1.18 | 1.14 | 1.18 | 1.15 |

Table 6.7: **Average speedup over the four floating point program groups of IPS and RASE over Postpass** on three architectures.

| Speedup over Postpass | | |
|---|---|---|
| Architecture | IPS | RASE |
| 88000 | 1.14 | 1.15 |
| R2000 | 1.07 | 1.07 |
| i860 | 1.05 | 1.06 |

## 6.4 Summary

This chapter presents a study of code generation strategies to test the hypothesis that separating instruction scheduling and register allocation produces inefficient schedules. Three strategies were examined: Postpass separates the two phases completely, performing register allocation first; IPS schedules before register allocation, using a heuristic that emulates the register allocation constraints; RASE most closely couples the two phases by computing schedule cost estimates that are used by the register allocator.

Both RASE and IPS are considerably better than Postpass (see Table 6.7), supporting the hypothesis that integrating the two phases is necessary. IPS is better than Postpass for two reasons. First, by limiting local register usage, the IPS scheduler emulates the constraints under which the register allocator must operate. The register limit takes into consideration the global pseudo-registers referenced within a block. This restrains the growth of register pressure, which reduces overall spilling and, in particular, avoids significant spilling in important blocks. Second, because IPS schedules before register allocation, the scheduler is free from specific register restrictions and also presents the register allocator with an ordering that is closer to the final schedule (modulo spill code), which gives the allocator a better picture of the interference between pseudo-registers.

RASE is better than Postpass for two reasons. First, the schedule cost estimates let the register allocator find a balance between local and global register needs, which gives each basic block sufficient registers to exploit instruction-level parallelism, but not enough to cause unnecessary spilling elsewhere. Second, it delays local register allocation until final scheduling, which reduces specific register restrictions on the scheduler.

On the three architectures examined in this study, RASE achieves little speedup over IPS (1% on the average), except on programs with large basic blocks. On architectures with fewer registers or longer latencies, RASE achieves greater speedups. This is discussed in the next chapter.

Three modifications to RASE may improve the code it produces. First, RASE could present the global register allocator with an ordering computed by PRESCHED. The prepass reordering would make RASE more like IPS, but RASE would still retain the more precise heuristic for determining the local register limit. Second, RASE could separate local register allocation from FINALSCHED, which may eliminate unnecessary local spills caused by the difficulty of performing these two functions together. Third, a global register allocator with live-range splitting might reduce overall spilling. (This might also improve the other strategies.)

Both IPS and RASE take more compilation time than Postpass. IPS takes roughly twice as much time as Postpass, because it schedules each block twice and because the scheduler must check register limits. RASE is roughly three times slower than IPS, because it schedules each block three or four times (depending on the number of register sets) and because the scheduling algorithm is more complicated than IPS's. More detail is given in Section 8.2.

In summary, the results demonstrate that some level of integration is necessary to produce efficient schedules, but that the implementation and compilation expense of strategies that closely couple the two phases, such as RASE, is unnecessary at present. Nevertheless, given the trend toward techniques that increase instruction-level parallelism, RASE's better performance on programs with very large basic blocks makes it a good candidate for further examination.

# Chapter 7

# Register Sets and Latency Versus Code Generation Strategy

This chapter discusses experiments that examine the effect on RISC processor performance of register set size and organization, and operation and load latency, using different code generation strategies. The strategies used in these studies are those discussed in the previous chapter. The architectures that are examined incorporate many features of two single-processor RISCs, the Motorola 88000 and the MIPS R2000, although none is an exact model of either.

Naturally, more registers yield better overall performance, but at some point the additional benefit diminishes. In general, code generators that integrate register allocation and instruction scheduling yield more efficient schedules, but their relative advantage decreases as the register set size increases.

The results show that, for a smaller number of registers, register pressure causes the architecture with a shared register organization to execute faster than a split organization; for a larger number of registers, the write-back bus to the shared register set becomes the bottleneck, making a split organization better. The split organization's advantage can be offset by double-porting the shared register file.

The results also indicate that a machine with a floating point coprocessor does not always execute faster than one with a slower on-chip implementation, if the coprocessor does not also perform expensive integer operations, such as multiply and divide. The problem can be solved by transferring operands to the FPU, performing the operation there, and then shipping the data back to the CPU.

The rest of this chapter is organized as follows: Section 7.1 describes the methodology that is specific to these studies; Section 7.2 discusses the experiments and results; Section 7.3 outlines related work; and Section 7.4 summarizes the results of these experiments.

## 7.1 Methodology

Most architectural features used in these studies are adopted from the Motorola 88000 chip and the MIPS R2000/R2010 chip set. The aim is not to model these machines exactly, but to choose components that would constitute a "reasonable" base model for RISCs, and then vary features of that model. When varying a particular feature, other aspects of the architecture remain fixed.

The architectures examined include several common features. First, the CPU contains a 4-stage instruction pipeline that handles all integer operations, except multiply and divide. Second, the FPU contains separate add and multiply pipelines that share initial and final stages. Divide

Table 7.1: **Longerop and Shorterop operation latencies** used in the experiments.

| | Latency (in cycles) | |
|---|---|---|
| | Longerop | Shorterop |
| fadd.s | 5 | 2 |
| fmul.s | 6 | 4 |
| fdiv.s | 30 | 11 |
| fadd.d | 6 | 2 |
| fmul.d | 9 | 5 |
| fdiv.d | 60 | 18 |
| imul | 4 | 12 |
| idiv | 38 | 35 |

Table 7.2: **Longload and Shortload load latencies** used in the experiments.

| | Latency (in cycles) | |
|---|---|---|
| | Longload | Shortload |
| ld | 11 | 2 |
| ld.d | 12 | 3 |

operations iterate within the first add pipe stage. Third, a 32-bit bus connects the CPU to a data cache and memory.[1] Fourth, load operations use a 2-stage load pipeline. Integer and single precision floating point loads have a 2-cycle latency; double precision loads take 3 cycles.

Five independent architectural parameters are examined: (1) the code generation strategy; (2) the number of 32-bit registers (16 to 128);[2] (3) the organization of the register file; (4) the operation latencies; and (5) the load latency.

The experiments compare two sets of operation latencies: *Longerop*, which models 88000 latencies, and *Shorterop*, which models R2000 latencies. (See Table 7.1.) These particular latencies were chosen because they represent those of two alternative chip designs, single-chip and dual-chip. The floating point latencies of the dual-chip design are roughly half those of the single-chip, because more of the dual-chip area can be devoted to floating point operations. In the dual-chip design, integer multiply is done on the CPU chip; the latency is three times that of the single-chip, which uses the FPU to perform this operation.

The studies compare two sets of load latencies: *Longload*, which models architectures without a cache (such as the Cray-1), and *Shortload*, which models architectures with a cache. (See Table 7.2.) Only the shorter load latencies are used in most of the experiments; the longer latencies are used only in the load latency study and the code generation strategy study.

Both split and shared register organizations are examined. In a split organization registers are usually divided into two equal partitions for integer and floating point values, respectively. This is the logical choice for an architecture with a separate floating point coprocessor; therefore, in our studies the split organization is usually coupled with the shorter floating point operation latencies.

[1]The memory system is not modeled; instead all references are assumed to be cache hits.

[2]The instruction word format is not altered to take register set size into consideration.

A shared organization is most appropriate for a single chip implementation; therefore, it is paired with the longer floating point latencies. The shared organization is also used with shorter latencies in some of the experiments, to test the effect of changing only the register organization. In all studies, double precision floating point values use 64-bit even-odd register pairs.

The code generation strategies used in these experiments are described in Chapter 6. All of them perform global (procedure-wide) register allocation. Interprocedural or link-time register allocation was not considered; With such a register allocation technique, a greater number of registers could be used effectively, mostly to reduce procedure call overhead [Wal86, Cho88]. Techniques to increase basic block size, such as loop unrolling, trace scheduling and software pipelining, were not considered; these techniques could also increase the need for more registers. No global or loop optimizations were performed, except as noted.

## 7.2 Experiments and Results

### 7.2.1 Register Set Size

This experiment examines the effect on performance of register set sizes ranging from 16 to 128 registers, across three code generation strategies and three architectures. The architectures differ in register organization and operation latency. *A88sh* has a shared register set and Longerop operation latencies; *Ar2sh* has the same register organization but with Shorterop latencies; *Ar2sp* also has Shorterop latencies, but registers are split into two equal partitions, except for 32/16 and 64/32, in which the floating point partition is half the size of the integer partition. For each architecture, the execution time of each program is normalized to the execution time for the same strategy and same architecture with 32 registers. Tables 7.3 and 7.4 show the results. Figures 7.1 through 7.3 at the end of the chapter summarize the data in the tables.

With the IPS and RASE code generation strategies, 32 registers is a good choice for all three architectures. Given only 16 registers, the degradation (from 32 registers) averages 12% for A88sh and Ar2sh and 26% for Ar2sp. Above 32 registers, execution speed improves only 1%, on the average, for A88sh and Ar2sh and 4% for Ar2sp. Although the slope of the execution speed curve is steeper (as register set size increases) for the split register organization, Ar2sp, than the shared organizations, there is little improvement above 32 registers. The Ar2sp slope is steeper because many programs require more registers of one type than the other. In addition, those needs may vary over different parts of a program.

The same general trends apply to Postpass, except that the best register set size shifts to 64 registers. Over all three architectures, the improvement between 32 and 64 registers averages 9%. As with RASE and IPS, when using Postpass, the split register organization is more constrained with fewer registers, because of the varying needs of different programs. This is evident in the Perfect and Misc groups, which contain the larger programs. In general, Postpass uses registers less efficiently than the other two strategies, because there is no communication between register allocation and instruction scheduling.

The benefit of using an integrated code generator, such as IPS or RASE, is that architects can more comfortably choose a smaller register set size. In this experiment both strategies brought large gains when varying register set size from 16 to 32, and both limited the penalty of not using 64 or more. A smaller register set size has several advantages: faster access time, less chip area,

Table 7.3: **Effect of register set size across three machine organizations** (shared register set with longer operation latencies (A88sh), shared with shorter latencies (Ar2sh) and split with shorter latencies (Ar2sp)). Register sizes 32/16 and 64/32 were not used with the shared register sets. The execution times for an architecture are normalized to the same machine organization with 32 registers. Each figure is the harmonic mean of all programs within a particular workload group. The data shown here is for the code generation strategies Postpass and IPS. The data for RASE is in the next table.

| | | Prog Group | Register Set Size 16 | 32 | 32/16 | 64 | 64/32 | 128 |
|---|---|---|---|---|---|---|---|---|
| Postpass | A88sh | Liverm | 1.12 | 1.00 | | 0.91 | | 0.90 |
| | | Nasker | 1.18 | 1.00 | | 0.86 | | 0.84 |
| | | Perfect | 1.17 | 1.00 | | 0.94 | | 0.91 |
| | | Misc | 1.13 | 1.00 | | 0.93 | | 0.90 |
| | | Int | 0.98 | 1.00 | | 1.03 | | 1.05 |
| | Ar2sh | Liverm | 1.12 | 1.00 | | 0.92 | | 0.91 |
| | | Nasker | 1.17 | 1.00 | | 0.88 | | 0.86 |
| | | Perfect | 1.15 | 1.00 | | 0.94 | | 0.91 |
| | | Misc | 1.14 | 1.00 | | 0.92 | | 0.90 |
| | | Int | 0.98 | 1.00 | | 1.03 | | 1.04 |
| | Ar2sp | Liverm | 1.39 | 1.00 | 0.92 | 0.91 | 0.87 | 0.87 |
| | | Nasker | 1.29 | 1.00 | 0.88 | 0.88 | 0.83 | 0.82 |
| | | Perfect | 1.30 | 1.00 | 0.94 | 0.91 | 0.88 | 0.87 |
| | | Misc | 1.20 | 1.00 | 0.92 | 0.88 | 0.87 | 0.86 |
| | | Int | 1.04 | 1.00 | 1.02 | 1.02 | 1.04 | 1.04 |
| IPS | A88sh | Liverm | 1.04 | 1.00 | | 1.00 | | 1.00 |
| | | Nasker | 1.17 | 1.00 | | 0.99 | | 1.00 |
| | | Perfect | 1.15 | 1.00 | | 0.96 | | 0.94 |
| | | Misc | 1.19 | 1.00 | | 0.99 | | 0.99 |
| | | Int | 1.00 | 1.00 | | 1.00 | | 1.00 |
| | Ar2sh | Liverm | 1.04 | 1.00 | | 1.00 | | 1.00 |
| | | Nasker | 1.13 | 1.00 | | 0.99 | | 1.00 |
| | | Perfect | 1.16 | 1.00 | | 0.97 | | 0.96 |
| | | Misc | 1.23 | 1.00 | | 1.00 | | 0.99 |
| | | Int | 1.00 | 1.00 | | 1.00 | | 1.00 |
| | Ar2sp | Liverm | 1.28 | 1.00 | 0.99 | 0.99 | 0.98 | 0.99 |
| | | Nasker | 1.22 | 1.00 | 0.94 | 0.93 | 0.94 | 0.94 |
| | | Perfect | 1.33 | 1.00 | 0.96 | 0.93 | 0.92 | 0.91 |
| | | Misc | 1.31 | 1.00 | 0.96 | 0.94 | 0.94 | 0.94 |
| | | Int | 1.07 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 |

Table 7.4: **Effect of register set size across three machine organizations** (shared register set with longer operation latencies (A88sh), shared with shorter latencies (Ar2sh) and split with shorter latencies (Ar2sp)). Register sizes 32/16 and 64/32 were not used with the shared register sets. The execution times for an architecture are normalized to the same machine organization with 32 registers. Each figure is the harmonic mean of all programs within a particular workload group. The data shown here is for the RASE code generation strategy. The data for Postpass and IPS is in the previous table.

| | | Prog | Register Set Size | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | Group | 16 | 32 | 32/16 | 64 | 64/32 | 128 |
| RASE | A88sh | Liverm | 1.01 | 1.00 | | 1.00 | | 1.00 |
| | | Nasker | 1.16 | 1.00 | | 1.00 | | 1.00 |
| | | Perfect | 1.13 | 1.00 | | 0.96 | | 0.95 |
| | | Misc | 1.13 | 1.00 | | 0.98 | | 0.98 |
| | | Int | 1.00 | 1.00 | | 1.00 | | 1.00 |
| | Ar2sh | Liverm | 1.01 | 1.00 | | 1.00 | | 1.00 |
| | | Nasker | 1.15 | 1.00 | | 1.00 | | 1.01 |
| | | Perfect | 1.17 | 1.00 | | 0.97 | | 0.97 |
| | | Misc | 1.18 | 1.00 | | 0.98 | | 0.98 |
| | | Int | 1.00 | 1.00 | | 1.00 | | 1.00 |
| | Ar2sp | Liverm | 1.20 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 |
| | | Nasker | 1.33 | 1.00 | 0.95 | 0.95 | 0.95 | 0.95 |
| | | Perfect | 1.32 | 1.00 | 0.98 | 0.96 | 0.95 | 0.94 |
| | | Misc | 1.22 | 1.00 | 0.99 | 0.97 | 0.97 | 0.97 |
| | | Int | 1.10 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 |

and perhaps most important, one fewer bit per operand in the instruction format. If techniques that increase basic block size, such as loop unrolling and trace scheduling, become more widely used, the need for more registers will increase; sophisticated strategies will then have even more utility.

When using Postpass, the Int group performance *decreases* as register set size increases. This is because the Round-robin register allocation policy uses more registers than necessary for integer code, increasing the number of registers that must be preserved over procedure calls.

Many of the Postpass results show that adding additional integer registers is more important than adding floating point registers. This is because these programs contain many integer address calculations. If loop optimizations, such as induction variable analysis and loop invariant removal, were performed, the amount of integer computation would be reduced, but integer register pressure may be increased by the additional registers needed to hold induction variables and invariants. Therefore, even with loop optimizations, adding integer registers remains more important than adding floating point registers. Section 7.2.6 discusses loop optimizations in greater detail.

### 7.2.2 Register Set Organization

Table 7.5 shows the effect of changing register set organization. (Figures 7.4 through 7.6 at the end of the chapter summarize the data in the table.) The architectures examined are identical

Table 7.5: **Effect of register set organization** (shared (Ar2sh), split (Ar2sp) and shared with an additional write-back bus (Ar2shb)) across different register set sizes and code generation strategies. The execution times for an architecture are normalized to Ar2sh with the same number of registers. Each figure is the harmonic mean of all programs within a particular workload group. The register set size of choice for a particular code generation strategy is in bold face type.

| Sz | Prog Group | Postpass Ar2sh | Postpass Ar2sp | Postpass Ar2shb | IPS Ar2sh | IPS Ar2sp | IPS Ar2shb | RASE Ar2sh | RASE Ar2sp | RASE Ar2shb |
|---|---|---|---|---|---|---|---|---|---|---|
| 16 | Lmore | 1.00 | 1.19 | 0.93 | 1.00 | 1.14 | 0.91 | 1.00 | 1.09 | 0.91 |
| | Nasker | 1.00 | 1.10 | 0.97 | 1.00 | 1.09 | 0.96 | 1.00 | 1.16 | 0.94 |
| | Perfect | 1.00 | 1.09 | 0.95 | 1.00 | 1.15 | 0.97 | 1.00 | 1.12 | 0.97 |
| | Misc | 1.00 | 1.01 | 0.94 | 1.00 | 1.00 | 0.92 | 1.00 | 1.01 | 0.94 |
| | Int | 1.00 | 1.03 | 1.00 | 1.00 | 1.06 | 1.00 | 1.00 | 1.11 | 1.00 |
| 32 | Lmore | 1.00 | 0.96 | 0.91 | **1.00** | **0.92** | **0.91** | **1.00** | **0.92** | **0.91** |
| | Nasker | 1.00 | 1.01 | 0.95 | **1.00** | **1.00** | **0.95** | **1.00** | **1.00** | **0.95** |
| | Perfect | 1.00 | 0.96 | 0.94 | **1.00** | **1.01** | **0.98** | **1.00** | **0.98** | **0.97** |
| | Misc | 1.00 | 0.96 | 0.91 | **1.00** | **0.94** | **0.91** | **1.00** | **0.92** | **0.92** |
| | Int | 1.00 | 0.98 | 1.00 | **1.00** | **0.99** | **1.00** | **1.00** | **1.01** | **1.00** |
| 64 | Lmore | **1.00** | **0.95** | **0.91** | 1.00 | 0.91 | 0.91 | 1.00 | 0.92 | 0.91 |
| | Nasker | **1.00** | **0.99** | **0.94** | 1.00 | 0.95 | 0.95 | 1.00 | 0.95 | 0.96 |
| | Perfect | **1.00** | **0.94** | **0.93** | 1.00 | 0.97 | 0.96 | 1.00 | 0.96 | 0.95 |
| | Misc | **1.00** | **0.93** | **0.92** | 1.00 | 0.89 | 0.90 | 1.00 | 0.90 | 0.91 |
| | Int | **1.00** | **0.97** | **1.00** | 1.00 | 0.99 | 1.00 | 1.00 | 1.01 | 1.00 |
| 128 | Lmore | 1.00 | 0.92 | 0.91 | 1.00 | 0.91 | 0.91 | 1.00 | 0.92 | 0.91 |
| | Nasker | 1.00 | 0.97 | 0.95 | 1.00 | 0.94 | 0.95 | 1.00 | 0.95 | 0.95 |
| | Perfect | 1.00 | 0.93 | 0.94 | 1.00 | 0.95 | 0.95 | 1.00 | 0.95 | 0.94 |
| | Misc | 1.00 | 0.92 | 0.92 | 1.00 | 0.89 | 0.90 | 1.00 | 0.90 | 0.91 |
| | Int | 1.00 | 0.98 | 1.00 | 1.00 | 0.99 | 1.00 | 1.00 | 1.01 | 1.00 |

except for this feature. All have the longer operation latencies. Ar2sh has a shared register file. Ar2sp has registers split into integer and floating point partitions. *Ar2shb* is the same as Ar2sh, but with an additional write-back bus and register port, giving it the same number of ports as Ar2sp. The execution times for each architecture are normalized to Ar2sh with the same strategy and number of registers.

Surprisingly, for register set sizes greater than 16 registers, the split organization is better than the shared organization in most cases. This is counterintuitive when considering register set organization alone; the performance of the shared organization should be at least as good as the split organization, because a program would have access to more registers when executing in either a floating point or integer intensive section of code.

One hypothesis is that the crucial performance factor is *access* to the register file rather than its organization. Split organizations use a separate write-back bus for each register partition; the shared organization architecture, Ar2sh, has only one write-back bus. To test this hypothesis, a second write-back bus and corresponding register port were added to Ar2sh. (See the Ar2shb column.) The additional bus enabled programs to execute at least as fast as the split organization in nearly all cases. In particular, for a register set size of 32, the second bus brought an improvement over Ar2sp averaging 2% across the three code generation strategies. There were cases where Ar2sp performed better than Ar2shb. These fall into two categories. First, on the Int group with at least 32 registers, the Postpass uses more registers with Ar2shb than Ar2sp, because of the shared organization. This increases procedure call overhead. Second, using IPS or RASE, when there are enough registers (64 or more), Ar2sp and Ar2shb perform equally well (always within 1%); the variations are due primarily to the use of heuristics in the algorithms.

For small register set sizes (*e.g.*, 16) the shared organization is better than the split. The smaller number of registers is a greater bottleneck for Ar2sp than the write-back bus is for Ar2sh. With 32 or more registers, the opposite is true.

In conclusion, a shared organization, coupled with an extra write-back bus and port, is better than either a single-ported shared file or a split organization.

### 7.2.3 Operation Latency

An architecture with a separate floating point coprocessor, such as the R2000, has the chip area to implement floating point operations efficiently; consequently, it has shorter floating point operation latencies. Therefore, given a floating point intensive workload, one would expect that Ar2sp, which has shorter latencies, would have lower execution times than A88sh, which models a single-chip implementation and has longer latencies. Consider, however, columns A88sh and Ar2sp of Table 7.6. For nearly 50% of the program groups, Ar2sp has slower execution times than A88sh. The slower times are primarily due to integer multiplies; on Ar2sp integer multiply is done in the CPU and has a longer latency than floating point multiply. Array address calculations are responsible for most of the multiplies; many of them would be eliminated by loop optimizations. Nevertheless, it is interesting to examine the implications of programs with numerous long integer operations.

One advantage of combining both integer and floating point operations on the same chip is that a faster floating point unit can easily be used for integer multiplies, as occurs on the 88000. This improves the execution speed of programs that are dominated by integer operations. The R2000, on the other hand, does integer multiply within the main processor, instead of the floating point coprocessor. The operation latency for the CPU integer multiply is large compared to the floating point latencies.

To test whether the Ar2sp's slower integer multiply is a bottleneck, we created a variation of Ar2sp called *Ar2spf*. To perform integer multiply or divide on Ar2spf, the operands are transferred to floating point registers, the operation is executed in the FPU and the result is transmitted back. A transfer between register sets has a 2-cycle latency.

Ar2spf shows improved performance for all program groups where Ar2sp is slower than A88sh. Even in those cases in which the Ar2sp had been faster, doing integer multiplies and divides in the floating point unit resulted in better or comparable performance. The only exception is the Livermore group, on which Ar2spf exhibited a slight degradation from Ar2sp. These programs have enough floating point operations to hide most integer multiplies on Ar2sp; issuing floating point instructions is the bottleneck. Therefore, the additional transfer instructions and floating point multiply instructions required by Ar2spf cause a reduction in the floating point instruction issue rate and thus a performance degradation. In general, however, the effect of using the floating point coprocessor to execute integer multiplies improves execution time. Figures 7.7 through 7.9 at the end of the chapter summarize the data in Table 7.6.

Table 7.6: **Effect of operation latencies** (longer latencies (A88sh), shorter latencies (Ar2sp) and shorter latencies with integer multiply and divide done in the floating point coprocessor (Ar2spf)) across register set sizes and code generation strategies. The execution times for each architecture are normalized to A88sh with the same number of registers. Each figure is the harmonic mean of all programs within a particular workload group. The register set size of choice for a particular code generation strategy is in bold face type.

| Sz | Prog Group | Postpass A88sh | Postpass Ar2sp | Postpass Ar2spf | IPS A88sh | IPS Ar2sp | IPS Ar2spf | RASE A88sh | RASE Ar2sp | RASE Ar2spf |
|---|---|---|---|---|---|---|---|---|---|---|
| 16 | Lmore | 1.00 | 1.10 | 1.09 | 1.00 | 1.08 | 1.07 | 1.00 | 1.03 | 1.02 |
| | Nasker | 1.00 | 1.20 | 1.12 | 1.00 | 1.20 | 1.10 | 1.00 | 1.33 | 1.17 |
| | Perfect | 1.00 | 1.04 | 1.01 | 1.00 | 1.06 | 0.96 | 1.00 | 1.08 | 1.01 |
| | Misc | 1.00 | 0.90 | 0.89 | 1.00 | 0.91 | 0.87 | 1.00 | 0.92 | 0.88 |
| | Int | 1.00 | 1.06 | 0.98 | 1.00 | 1.07 | 1.00 | 1.00 | 1.12 | 1.03 |
| 32 | Lmore | 1.00 | 0.90 | 0.90 | **1.00** | **0.88** | **0.89** | **1.00** | **0.87** | **0.88** |
| | Nasker | 1.00 | 1.09 | 0.99 | **1.00** | **1.14** | **0.98** | **1.00** | **1.15** | **1.02** |
| | Perfect | 1.00 | 0.93 | 0.88 | **1.00** | **0.93** | **0.86** | **1.00** | **0.91** | **0.86** |
| | Misc | 1.00 | 0.85 | 0.83 | **1.00** | **0.82** | **0.79** | **1.00** | **0.79** | **0.79** |
| | Int | 1.00 | 1.00 | 0.93 | **1.00** | **1.01** | **0.93** | **1.00** | **1.03** | **0.95** |
| 64 | Lmore | **1.00** | **0.90** | **0.90** | 1.00 | 0.87 | 0.88 | 1.00 | 0.87 | 0.88 |
| | Nasker | **1.00** | **1.13** | **1.00** | 1.00 | 1.08 | 0.97 | 1.00 | 1.10 | 0.99 |
| | Perfect | **1.00** | **0.90** | **0.85** | 1.00 | 0.91 | 0.85 | 1.00 | 0.91 | 0.85 |
| | Misc | **1.00** | **0.81** | **0.80** | 1.00 | 0.78 | 0.77 | 1.00 | 0.78 | 0.78 |
| | Int | **1.00** | **0.99** | **0.91** | 1.00 | 1.01 | 0.93 | 1.00 | 1.03 | 0.95 |
| 128 | Lmore | 1.00 | 0.88 | 0.87 | 1.00 | 0.87 | 0.88 | 1.00 | 0.87 | 0.88 |
| | Nasker | 1.00 | 1.08 | 0.97 | 1.00 | 1.07 | 0.96 | 1.00 | 1.09 | 0.98 |
| | Perfect | 1.00 | 0.89 | 0.84 | 1.00 | 0.90 | 0.84 | 1.00 | 0.90 | 0.85 |
| | Misc | 1.00 | 0.81 | 0.81 | 1.00 | 0.78 | 0.77 | 1.00 | 0.78 | 0.78 |
| | Int | 1.00 | 0.99 | 0.92 | 1.00 | 1.01 | 0.93 | 1.00 | 1.03 | 0.95 |

### 7.2.4 Load Latency

If a processor has no cache, the load latency will be considerably longer than one with a cache. This experiment examines the effect of register set size on two architectures with long load latencies. The architectures are the same as two of those examined in the first experiment, in Section 7.2.1, except for load latency. *A88shL* has the Longerop operation latencies and a shared register file. *Ar2spL* has the Shorterop operation latencies and a split register file.

The results, shown in Table 7.7 exhibit the same trends as the first experiment, except that the point of diminishing returns is shifted to 64 registers with IPS and RASE. With Postpass, 128 registers provide up to a 5% performance increase for A88shL and 10% on Ar2spL. Also, additional registers are much more important for the split register file organization, than the shared, with all three strategies. Figures 7.10 through 7.12 at the end of the chapter summarize the data in the table.

Table 7.7: **Effect of register set size across two machine organizations, each with long load latencies**, and three code generation strategies. The execution time of each program is normalized to the same architecture with 32 registers. Each figure is the harmonic mean of all programs within a particular workload group.

| | Machine | Program Group | Register Set Size 16 | 32 | 64 | 128 |
|---|---|---|---|---|---|---|
| Postpass | A88shL | Livermore | 1.19 | 1.00 | 0.78 | 0.77 |
| | | Nasker | 1.47 | 1.00 | 0.79 | 0.73 |
| | | Perfect | 1.29 | 1.00 | 0.87 | 0.82 |
| | | Misc | 1.25 | 1.00 | 0.88 | 0.86 |
| | | Int | 1.07 | 1.00 | 0.94 | 0.93 |
| | Ar2spL | Livermore | 1.58 | 1.00 | 0.89 | 0.83 |
| | | Nasker | 1.62 | 1.00 | 0.79 | 0.68 |
| | | Perfect | 1.47 | 1.00 | 0.85 | 0.80 |
| | | Misc | 1.32 | 1.00 | 0.84 | 0.81 |
| | | Int | 1.17 | 1.00 | 0.94 | 0.87 |
| IPS | A88shL | Livermore | 1.20 | 1.00 | 1.00 | 1.00 |
| | | Nasker | 1.63 | 1.00 | 0.99 | 0.99 |
| | | Perfect | 1.33 | 1.00 | 0.88 | 0.84 |
| | | Misc | 1.44 | 1.00 | 0.96 | 0.96 |
| | | Int | 1.01 | 1.00 | 1.00 | 1.00 |
| | Ar2spL | Livermore | 1.89 | 1.00 | 0.90 | 0.90 |
| | | Nasker | 2.23 | 1.00 | 0.86 | 0.87 |
| | | Perfect | 1.66 | 1.00 | 0.83 | 0.77 |
| | | Misc | 1.70 | 1.00 | 0.87 | 0.87 |
| | | Int | 1.27 | 1.00 | 1.00 | 1.00 |
| RASE | A88shL | Livermore | 1.06 | 1.00 | 1.00 | 1.00 |
| | | Nasker | 1.38 | 1.00 | 0.99 | 1.00 |
| | | Perfect | 1.30 | 1.00 | 0.95 | 0.94 |
| | | Misc | 1.29 | 1.00 | 0.96 | 0.96 |
| | | Int | 1.01 | 1.00 | 1.00 | 1.00 |
| | Ar2spL | Livermore | 1.13 | 1.00 | 0.89 | 0.89 |
| | | Nasker | 1.67 | 1.00 | 0.86 | 0.87 |
| | | Perfect | 1.44 | 1.00 | 0.84 | 0.82 |
| | | Misc | 1.62 | 1.00 | 0.92 | 0.92 |
| | | Int | 1.35 | 1.00 | 1.00 | 1.00 |

Table 7.8: **Effect of code generation strategy across machine organizations and register set sizes.** This table compares the same architectures as Table 7.6, except the execution times for each strategy are normalized to Postpass with the same architecture and register set size. Each figure is the harmonic mean of all programs within a particular workload group.

| Sz | Program Group | A88sh PP | A88sh IPS | A88sh RASE | Ar2sh PP | Ar2sh IPS | Ar2sh RASE | Ar2sp PP | Ar2sp IPS | Ar2sp RASE |
|---|---|---|---|---|---|---|---|---|---|---|
| 16 | Livermore | 1.00 | 0.83 | 0.81 | 1.00 | 0.85 | 0.82 | 1.00 | 0.82 | 0.76 |
| | Nasker | 1.00 | 0.83 | 0.80 | 1.00 | 0.85 | 0.83 | 1.00 | 0.84 | 0.87 |
| | Perfect | 1.00 | 0.91 | 0.87 | 1.00 | 0.90 | 0.88 | 1.00 | 0.95 | 0.91 |
| | Misc | 1.00 | 0.90 | 0.85 | 1.00 | 0.91 | 0.87 | 1.00 | 0.90 | 0.86 |
| | Int | 1.00 | 0.86 | 0.88 | 1.00 | 0.85 | 0.87 | 1.00 | 0.87 | 0.93 |
| 32 | Livermore | 1.00 | 0.90 | 0.90 | 1.00 | 0.92 | 0.91 | 1.00 | 0.88 | 0.87 |
| | Nasker | 1.00 | 0.83 | 0.83 | 1.00 | 0.89 | 0.87 | 1.00 | 0.88 | 0.86 |
| | Perfect | 1.00 | 0.92 | 0.88 | 1.00 | 0.89 | 0.87 | 1.00 | 0.93 | 0.89 |
| | Misc | 1.00 | 0.84 | 0.85 | 1.00 | 0.84 | 0.84 | 1.00 | 0.82 | 0.80 |
| | Int | 1.00 | 0.83 | 0.85 | 1.00 | 0.83 | 0.84 | 1.00 | 0.84 | 0.88 |
| 64 | Livermore | 1.00 | 0.99 | 0.99 | 1.00 | 1.00 | 0.99 | 1.00 | 0.95 | 0.95 |
| | Nasker | 1.00 | 0.95 | 0.95 | 1.00 | 0.97 | 0.96 | 1.00 | 0.93 | 0.92 |
| | Perfect | 1.00 | 0.94 | 0.93 | 1.00 | 0.92 | 0.91 | 1.00 | 0.94 | 0.93 |
| | Misc | 1.00 | 0.91 | 0.91 | 1.00 | 0.91 | 0.90 | 1.00 | 0.86 | 0.87 |
| | Int | 1.00 | 0.80 | 0.82 | 1.00 | 0.79 | 0.81 | 1.00 | 0.82 | 0.85 |
| 128 | Livermore | 1.00 | 1.00 | 1.00 | 1.00 | 1.01 | 1.00 | 1.00 | 1.00 | 1.00 |
| | Nasker | 1.00 | 0.98 | 0.98 | 1.00 | 1.01 | 0.99 | 1.00 | 0.98 | 0.98 |
| | Perfect | 1.00 | 0.95 | 0.94 | 1.00 | 0.94 | 0.94 | 1.00 | 0.96 | 0.95 |
| | Misc | 1.00 | 0.92 | 0.92 | 1.00 | 0.92 | 0.91 | 1.00 | 0.88 | 0.88 |
| | Int | 1.00 | 0.77 | 0.79 | 1.00 | 0.77 | 0.79 | 1.00 | 0.79 | 0.82 |

### 7.2.5 Code Generation Strategy

In all experiments the IPS and RASE code generation strategies generate more efficient code than Postpass. As shown in Table 7.8, for 32 registers IPS and RASE are 15% better than Postpass, on the average, over all three architectures. These two strategies generate more efficient code, because they integrate register allocation and instruction scheduling. (Figures 7.13 and 7.14 at the end of the chapter summarize the data in this section.)

As register set size increases, the benefit of IPS or RASE over Postpass diminishes, because Postpass has enough registers to avoid constraining the scheduler. This result applies for all architectures and all program groups, except for the Int group. For integer code, the benefit of IPS or RASE over Postpass *increases* with register set size, because of Postpass's Round-robin register allocation policy. Recall (from Section 6.2.1) that Round-robin was chosen over First-fit, because it yields better results for floating point programs.

RASE is slightly better than IPS, given 32 registers or less. With 16 registers, RASE is an average of 2% better than IPS over all three architectures. Given 32 registers, the average is 1%. However, on the Perfect group, RASE is an average of nearly 4% better, with both 16 and 32 registers. On large basic blocks, which many Perfect group programs contain, IPS uses too many

Table 7.9: **Effect of code generation strategy across machine organizations and register set sizes.** Both architectures have long load latencies. The execution times for each strategy are normalized to Postpass with the same architecture and register set size. Each figure is the harmonic mean of all programs within a particular workload group.

| Sz | Program Group | A88shL PP | A88shL IPS | A88shL RASE | Ar2spL PP | Ar2spL IPS | Ar2spL RASE |
|---|---|---|---|---|---|---|---|
| 16 | Livermore | 1.00 | 0.81 | 0.69 | 1.00 | 1.02 | 0.61 |
| | Nasker | 1.00 | 0.81 | 0.66 | 1.00 | 1.04 | 0.78 |
| | Perfect | 1.00 | 0.98 | 0.85 | 1.00 | 1.12 | 0.89 |
| | Misc | 1.00 | 0.93 | 0.83 | 1.00 | 1.01 | 0.93 |
| | Int | 1.00 | 0.75 | 0.76 | 1.00 | 0.84 | 0.91 |
| 32 | Livermore | 1.00 | 0.77 | 0.77 | 1.00 | 0.86 | 0.85 |
| | Nasker | 1.00 | 0.71 | 0.71 | 1.00 | 0.75 | 0.76 |
| | Perfect | 1.00 | 0.94 | 0.84 | 1.00 | 0.98 | 0.90 |
| | Misc | 1.00 | 0.81 | 0.81 | 1.00 | 0.78 | 0.75 |
| | Int | 1.00 | 0.80 | 0.81 | 1.00 | 0.75 | 0.76 |
| 64 | Livermore | 1.00 | 0.99 | 0.99 | 1.00 | 0.89 | 0.89 |
| | Nasker | 1.00 | 0.89 | 0.88 | 1.00 | 0.83 | 0.84 |
| | Perfect | 1.00 | 0.95 | 0.91 | 1.00 | 0.95 | 0.91 |
| | Misc | 1.00 | 0.88 | 0.88 | 1.00 | 0.80 | 0.81 |
| | Int | 1.00 | 0.85 | 0.85 | 1.00 | 0.80 | 0.81 |
| 128 | Livermore | 1.00 | 1.00 | 1.00 | 1.00 | 0.94 | 0.94 |
| | Nasker | 1.00 | 0.96 | 0.96 | 1.00 | 0.96 | 0.97 |
| | Perfect | 1.00 | 0.98 | 0.96 | 1.00 | 0.96 | 0.95 |
| | Misc | 1.00 | 0.90 | 0.90 | 1.00 | 0.83 | 0.84 |
| | Int | 1.00 | 0.86 | 0.87 | 1.00 | 0.85 | 0.86 |

registers to avoid memory references and not enough to exploit instruction-level parallelism. RASE is able to find a better balance.

On machines with a long load latency, RASE exhibits a greater performance gain over Postpass than IPS. (See Table 7.9.) This is particularly evident on the Perfect group, which contains applications with large basic blocks; with 32 registers RASE is 10% better than IPS on A88shL and 8% better on Ar2spL. In addition, on A88shL, IPS exhibits less performance gain over Postpass with 16 registers than with 32 registers, and on Ar2spL with 16 registers, IPS performs worse than Postpass. This is because IPS sets the local register limit too low to handle the more constrained architectures, those with long load latencies and 16 registers.

In summary, this study reinforces the results of comparing the strategies from Chapter 6. Integrating instruction scheduling and register allocation produces more efficient schedules. Both IPS and RASE produce better code than Postpass. RASE performs better than IPS on programs with large basic blocks. In addition, RASE is much less sensitive to architectural changes, such as long load latencies and a small number of registers. Since RASE has a greater compile-time expense, it is probably not necessary for many architectures. Nevertheless, on architectures with long load latencies and/or small register set sizes, RASE is necessary to take advantage of the architecture's performance potential. Also, if the trend in compiler technology continues in the direction of greater loop unrolling and trace scheduling, using a strategy such as RASE will be advantageous.

### 7.2.6 Effect of Loop Optimizations

As explained in Section 7.1, Marion compilers do not perform global optimizations. To gauge the effect of loop optimizations on our previous results, induction variable analysis, strength reduction and loop invariant removal were done by hand on the Nasker group. The optimizations eliminated most integer multiplies. Table 7.10 shows the effect of register set size across register set organizations, operation latencies and code generation strategies, for both hand-optimized and unoptimized code. (The unoptimized numbers are the same as those in Table 7.3.) The execution time for each register set size (within a code generation strategy and optimization level) is normalized to the same architecture, with 32 registers.

Naturally, for all program groups and machine organizations the optimizations result in lower execution times. The savings differ considerably between machine organizations, but little between register set sizes. (See Table 7.11.) For A88sh, the savings average 11%. For the three architectures with the slower integer multiply, Ar2sh, Ar2shb and Ar2sp, the savings average 38%. For Ar2spf, where integer multiply uses the FPU, the savings average 26%.

Although the loop optimizations yield lower execution times, they do not change the general trends we have observed. For example, variations in register set size produce similar relative performance figures for both optimized and unoptimized code. Thirty-two registers is still the register set size of choice when using IPS and RASE; 64 registers are needed when using Postpass.[3] On the optimized code, for each code generation strategy, Ar2sp and Ar2spf performance are almost identical; also, the trend as register set size increases is close to the trend of the unoptimized code on Ar2spf. This makes sense, since the optimizations reduce the effect of a slow integer multiplier.

When comparing across machine organizations (Table 7.12), while holding the register set size fixed, the hand-optimized Nasker kernels follow the same trends as the unoptimized kernels: for 32 or more registers the write-back bus is still the bottleneck in the shared organization; IPS and RASE still perform significantly better than Postpass. One exception is that, for machine organizations Ar2sh, Ar2shb and Ar2sp, which have a slow integer multiply, the unoptimized code is slower, relative to A88sh; the optimized code is faster. Additionally, for Ar2spf, which uses the FPU for integer multiply, the unoptimized code is marginally faster, relative to A88sh; the optimized code is significantly faster. Both results are because the unoptimized code pays a high penalty for either the slow integer multiply or the additional transfers to and from the FPU.

## 7.3 Related Work

Hwu and Chang evaluated micro-architectural parallelism, including multiple instruction issue, multiple result distribution buses, multiple execution units and pipelined execution units [HC88]. All of their architecture variations had a split register organization with 32 integer and 32 floating point registers. Code was generated with a prepass code generation strategy, in which instruction scheduling precedes register allocation. Using the Livermore Loops and Linpack subroutines

[3]It remains to be seen whether this particular result will hold for programs with larger basic blocks, such as the Perfect Club benchmarks.

Table 7.10: **Effect of register set size across machine organizations and code generation strategies for unoptimized and hand-optimized Nasker group**. The execution times for each architecture are normalized to the same machine organization with 32 registers. Each figure is the harmonic mean of all programs in the Nasker group.

| | Mach | | Register Set Size 16 | 32 | 64 | 128 |
|---|---|---|---|---|---|---|
| Postpass | A88sh | Unopt | 1.18 | 1.00 | 0.86 | 0.84 |
| | | Opt | 1.23 | 1.00 | 0.88 | 0.87 |
| | Ar2sh | Unopt | 1.17 | 1.00 | 0.88 | 0.86 |
| | | Opt | 1.22 | 1.00 | 0.85 | 0.83 |
| | Ar2shb | Unopt | 1.19 | 1.00 | 0.89 | 0.87 |
| | | Opt | 1.25 | 1.00 | 0.86 | 0.85 |
| | Ar2sp | Unopt | 1.29 | 1.00 | 0.88 | 0.82 |
| | | Opt | 1.33 | 1.00 | 0.86 | 0.81 |
| | Ar2spf | Unopt | 1.33 | 1.00 | 0.87 | 0.82 |
| | | Opt | 1.30 | 1.00 | 0.85 | 0.80 |
| IPS | A88sh | Unopt | 1.17 | 1.00 | 0.99 | 1.00 |
| | | Opt | 1.14 | 1.00 | 0.99 | 1.00 |
| | Ar2sh | Unopt | 1.13 | 1.00 | 0.99 | 1.00 |
| | | Opt | 1.14 | 1.00 | 1.00 | 1.00 |
| | Ar2shb | Unopt | 1.14 | 1.00 | 0.99 | 0.99 |
| | | Opt | 1.18 | 1.00 | 1.00 | 1.00 |
| | Ar2sp | Unopt | 1.22 | 1.00 | 0.93 | 0.94 |
| | | Opt | 1.36 | 1.00 | 0.96 | 0.96 |
| | Ar2spf | Unopt | 1.32 | 1.00 | 0.98 | 0.98 |
| | | Opt | 1.34 | 1.00 | 0.96 | 0.95 |
| RASE | A88sh | Unopt | 1.16 | 1.00 | 1.00 | 1.00 |
| | | Opt | 1.13 | 1.00 | 1.00 | 1.00 |
| | Ar2sh | Unopt | 1.15 | 1.00 | 1.00 | 1.01 |
| | | Opt | 1.15 | 1.00 | 1.00 | 1.00 |
| | Ar2shb | Unopt | 1.16 | 1.00 | 1.00 | 1.00 |
| | | Opt | 1.15 | 1.00 | 1.00 | 1.00 |
| | Ar2sp | Unopt | 1.33 | 1.00 | 0.95 | 0.95 |
| | | Opt | 1.37 | 1.00 | 0.96 | 0.96 |
| | Ar2spf | Unopt | 1.34 | 1.00 | 0.97 | 0.96 |
| | | Opt | 1.38 | 1.00 | 0.96 | 0.96 |

Table 7.11: **Effect of loop optimizations on the Nasker group** for four register set sizes, five machine organizations and three code generation strategies. The table shows the performance improvement of the hand-optimized over the unoptimized Nasker group. Each figure is the harmonic mean of all programs in the Nasker group.

| | | Register Set Size 16 | 32 | 64 | 128 |
|---|---|---|---|---|---|
| Postpass | A88sh | 0.88 | 0.86 | 0.89 | 0.89 |
| | Ar2sh | 0.68 | 0.64 | 0.61 | 0.62 |
| | Ar2shb | 0.66 | 0.63 | 0.60 | 0.61 |
| | Ar2sp | 0.67 | 0.63 | 0.61 | 0.61 |
| | Ar2spf | 0.73 | 0.74 | 0.73 | 0.73 |
| IPS | A88sh | 0.88 | 0.90 | 0.91 | 0.90 |
| | Ar2sh | 0.63 | 0.61 | 0.62 | 0.62 |
| | Ar2shb | 0.62 | 0.60 | 0.61 | 0.61 |
| | Ar2sp | 0.68 | 0.60 | 0.61 | 0.61 |
| | Ar2spf | 0.78 | 0.76 | 0.74 | 0.74 |
| RASE | A88sh | 0.88 | 0.90 | 0.90 | 0.90 |
| | Ar2sh | 0.64 | 0.62 | 0.62 | 0.62 |
| | Ar2shb | 0.64 | 0.61 | 0.61 | 0.61 |
| | Ar2sp | 0.64 | 0.61 | 0.61 | 0.61 |
| | Ar2spf | 0.77 | 0.75 | 0.74 | 0.74 |

as benchmarks, Hwu and Chang concluded that, when used together, multiple instruction issue and pipelined execution units produced a speedup greater than the sum of the speedups of each taken separately. In addition, floating point pipelining was more important than pipelining the cache. They also found that issuing more than two instructions per cycle produced little additional speedup, even though loop unrolling was done by the compiler.

Others have examined multiple instruction issue, operation and memory access latencies and pipelined versus non-pipelined architectures, but with no variation in the number of registers or code generation strategy. Pleszkun and Sohi [PS88] studied variations of the Cray-1, which has eight 64-bit scalar registers. They found that multiple functional units improve performance, but pipelining functional units that execute in parallel does not significantly improve scalar code. They also found that performance is enhanced by issuing up to four instructions, when memory latencies are short. They did not describe their code generator.

Sohi and Vajapeyam [SV89] examined variations of a VLIW with unlimited registers. They performed loop unrolling and instruction scheduling. They found that as operation latencies increase, the need for multiple instruction issue decreases. They also concluded that given an unrestricted horizontal instruction format, a split register file is superior to a shared one; however, with a restricted instruction format, neither is significantly better than the other.

Goodman and Hsu [GH88] implemented the original Integrated Prepass Scheduler (IPS). They compared it to a number of strategies, including a version of Postpass. Their experiment is discussed in more detail in Section 6.2.2.

Table 7.12: **Effect of machine organization across register set sizes and code generation strategies for unoptimized and hand-optimized Nasker group**. The execution times for each architecture are normalized to A88sh with the same register set size. Each figure is the harmonic mean of all programs in the Nasker group.

| | Size | | A88sh | Ar2sh | Ar2shb | Ar2sp | Ar2spf |
|---|---|---|---|---|---|---|---|
| Postpass | 16 | Unopt | 1.00 | 1.10 | 1.05 | 1.20 | 1.12 |
| | | Opt | 1.00 | 0.90 | 0.85 | 0.96 | 0.95 |
| | 32 | Unopt | 1.00 | 1.11 | 1.04 | 1.09 | 0.99 |
| | | Opt | 1.00 | 0.91 | 0.84 | 0.89 | 0.90 |
| | 64 | Unopt | 1.00 | 1.14 | 1.07 | 1.13 | 1.00 |
| | | Opt | 1.00 | 0.88 | 0.82 | 0.86 | 0.86 |
| | 128 | Unopt | 1.00 | 1.14 | 1.07 | 1.08 | 0.97 |
| | | Opt | 1.00 | 0.88 | 0.83 | 0.83 | 0.83 |
| IPS | 16 | Unopt | 1.00 | 1.11 | 1.06 | 1.20 | 1.10 |
| | | Opt | 1.00 | 0.87 | 0.84 | 1.00 | 0.99 |
| | 32 | Unopt | 1.00 | 1.16 | 1.09 | 1.14 | 0.98 |
| | | Opt | 1.00 | 0.87 | 0.81 | 0.85 | 0.85 |
| | 64 | Unopt | 1.00 | 1.16 | 1.09 | 1.08 | 0.97 |
| | | Opt | 1.00 | 0.87 | 0.82 | 0.82 | 0.82 |
| | 128 | Unopt | 1.00 | 1.15 | 1.08 | 1.07 | 0.96 |
| | | Opt | 1.00 | 0.87 | 0.81 | 0.81 | 0.81 |
| RASE | 16 | Unopt | 1.00 | 1.14 | 1.09 | 1.33 | 1.17 |
| | | Opt | 1.00 | 0.90 | 0.85 | 1.06 | 1.07 |
| | 32 | Unopt | 1.00 | 1.15 | 1.09 | 1.15 | 1.02 |
| | | Opt | 1.00 | 0.89 | 0.84 | 0.88 | 0.88 |
| | 64 | Unopt | 1.00 | 1.15 | 1.10 | 1.10 | 0.99 |
| | | Opt | 1.00 | 0.89 | 0.84 | 0.84 | 0.84 |
| | 128 | Unopt | 1.00 | 1.15 | 1.09 | 1.09 | 0.98 |
| | | Opt | 1.00 | 0.89 | 0.84 | 0.84 | 0.84 |

All four studies measured only floating point intensive kernels, such as the Livermore Loops and Linpack subroutines, rather than complete applications, as was done in this study.

## 7.4 Summary

This study has examined the effect on RISC architecture performance of register set size and organization, operation and load latencies, and strategies for register allocation and instruction scheduling. The results show that a sophisticated code generation strategy reduces the number of registers needed to achieve good performance. By using such a strategy, architects can more comfortably use a smaller register set size, which can lower the register access time, use fewer bits in the instruction format and allow more logic to be spent on other components.

The results also show that changing the register set size has roughly the same effect on both of the architectures with the shared register organization, even though the operation latencies differed.

However, the change in register set size has a much greater impact on the architecture with the split organization.

Given a reasonable number of registers, the crucial factor (for register files) is the number of register write-back buses rather than the register set organization. In particular, the split organization, which inherently has a second write-back bus, yields faster execution times than the shared organization with the same register set size. Adding a second bus to the shared organization tips the balance the other way. Given only 16 registers, the split register organization yields slower execution times than the shared organization with either one or two write-back buses, because large programs can usually use more than 8 of either integer or floating point registers.

Shorter operation latencies usually result in faster execution. However, if most operation latencies are reduced at the expense of making a few latencies longer, then the instruction mix of a program determines whether there is an improvement. For example, programs dominated by long integer operations, such as multiply and divide, will execute more slowly if the latency of these operations is longer. The results show that the performance of these programs can be improved by using the floating point coprocessor, even with the additional expense of transferring the operands and result between chips. If the integer code is dominated by multiplies stemming from array address calculations, then global optimizations can reduce the number of such operations and a machine that has a separate (but slower) integer multiplier should perform as well as a machine that uses the FPU for integer multiply.

Among the variations examined, the choice of code generation strategy has the greatest effect on performance. Adding additional registers has a significant effect when using the Postpass strategy, but not when using the other strategies.

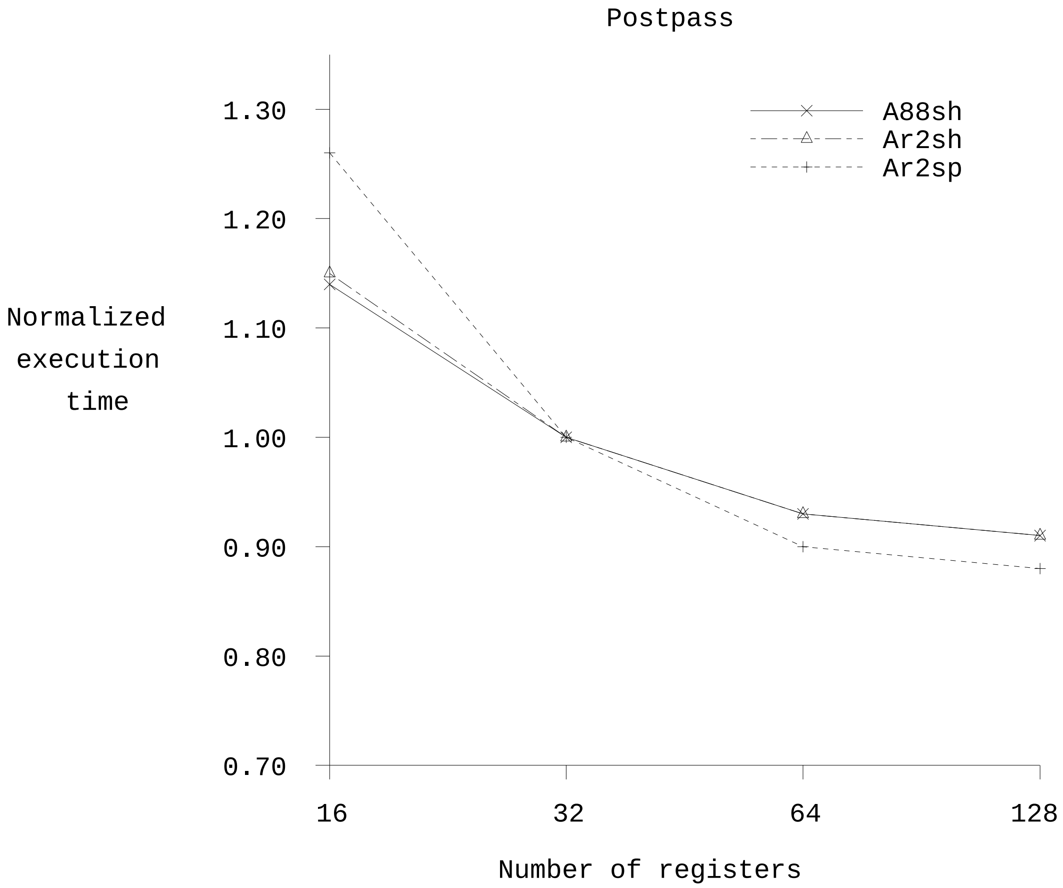



Figure 7.1: **Summary of the effect of register set size (Postpass)**. This graph summarizes the results from Section 7.2.1. It shows the harmonic mean of the normalized execution times of all four floating point intensive program groups for three architectures for Postpass. Times are normalized to 16 registers with the same architecture.

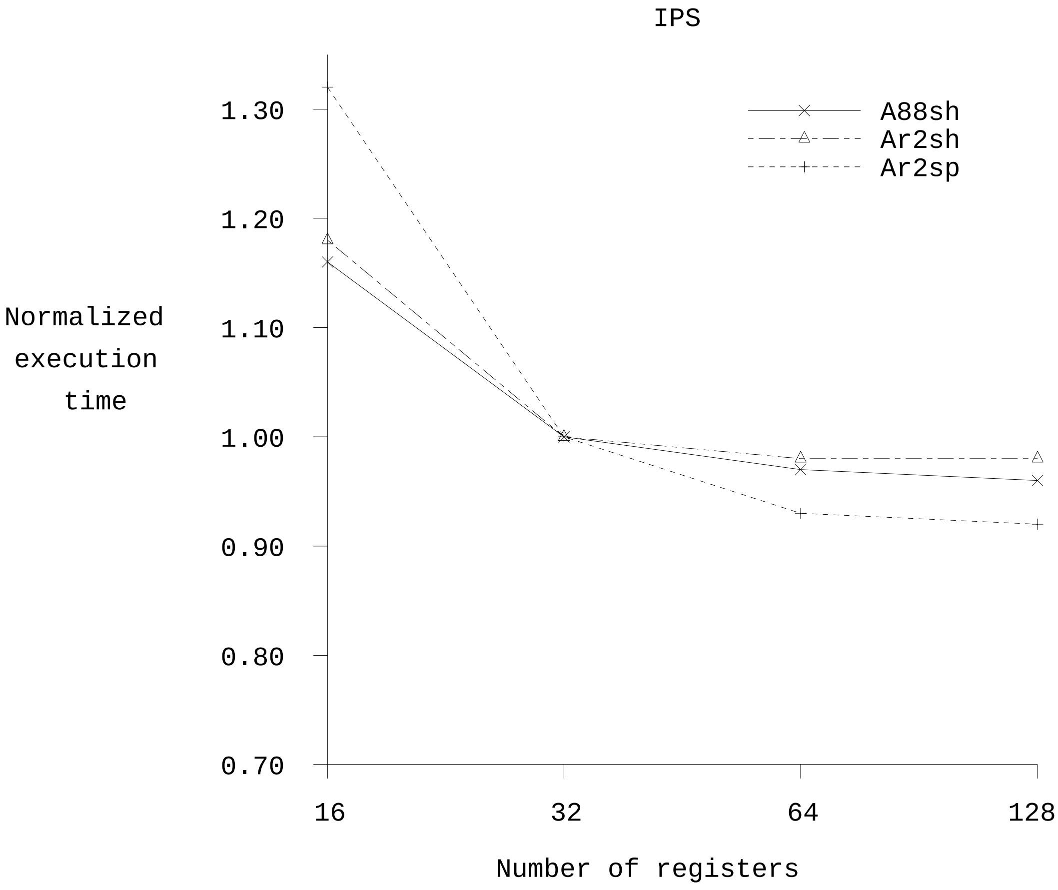



Figure 7.2: **Summary of the effect of register set size (IPS)**. This graph summarizes the results from Section 7.2.1. It shows the harmonic mean of the normalized execution times of all four floating point intensive program groups for three architectures for IPS. Times are normalized to 16 registers with the same architecture.

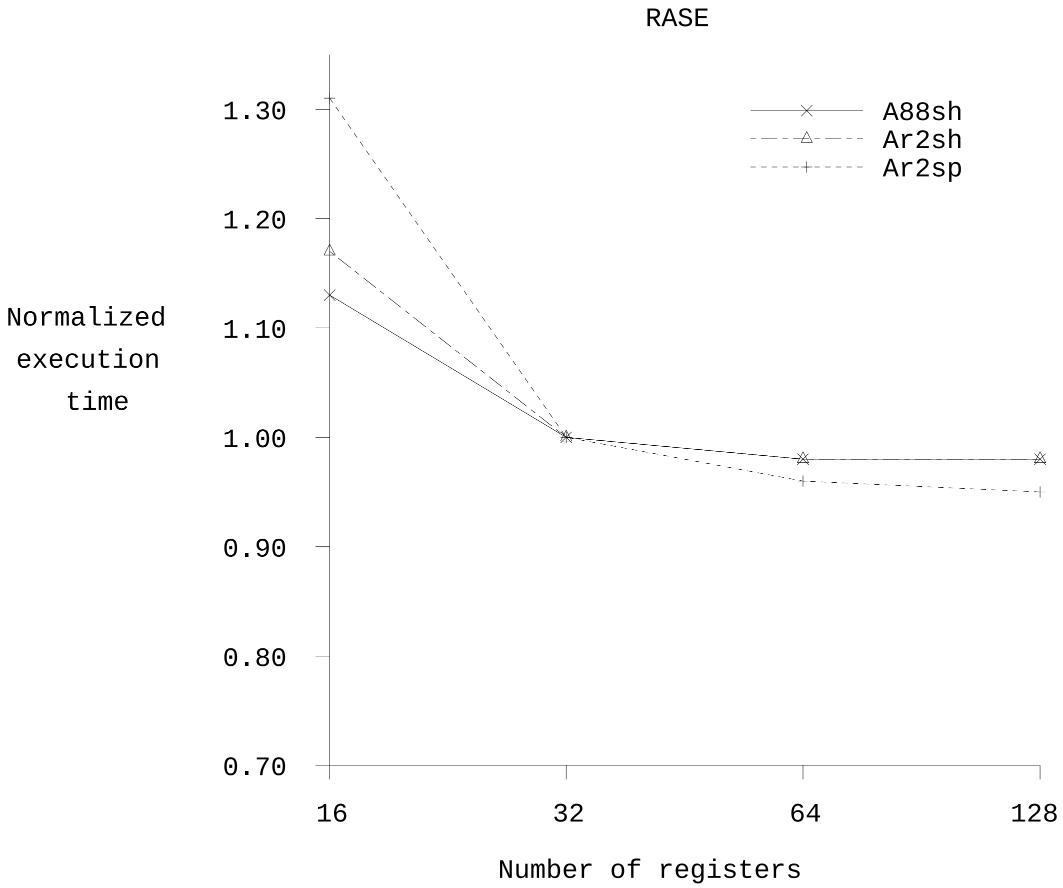



Figure 7.3: **Summary of the effect of register set size (RASE).** This graph summarizes the results from Section 7.2.1. It shows the harmonic mean of the normalized execution times of all four floating point intensive program groups for three architectures for RASE. Times are normalized to 16 registers with the same architecture.

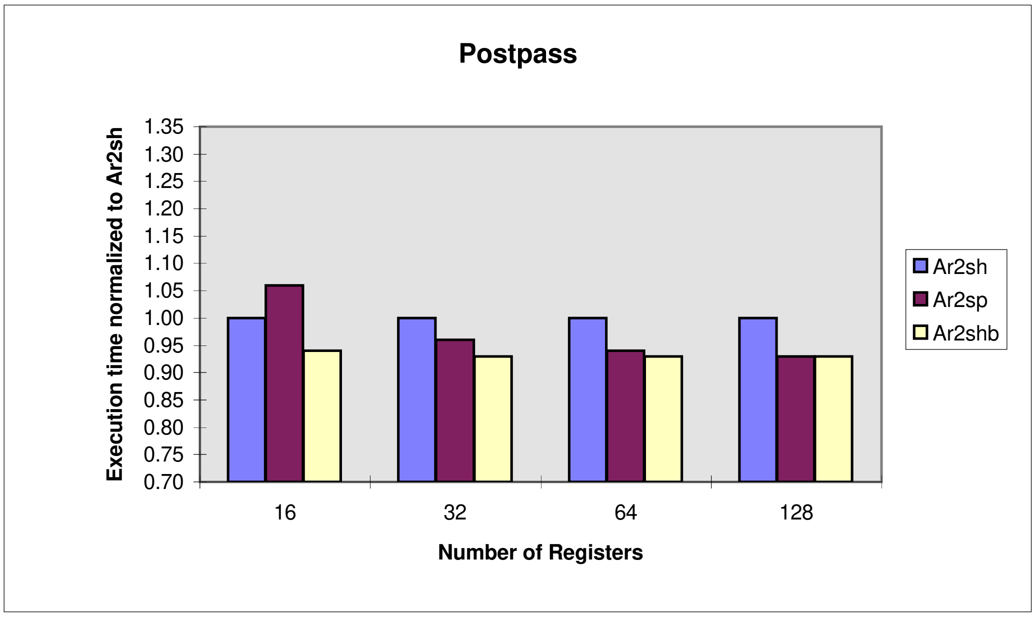



Figure 7.4: **Summary of the effect of register set organization (Postpass).** This graph summarizes the results from Section 7.2.2. It shows the harmonic mean of the normalized execution times of all four floating point intensive program groups for three architectures for Postpass. Times are normalized to Ar2sh with the same number of registers.

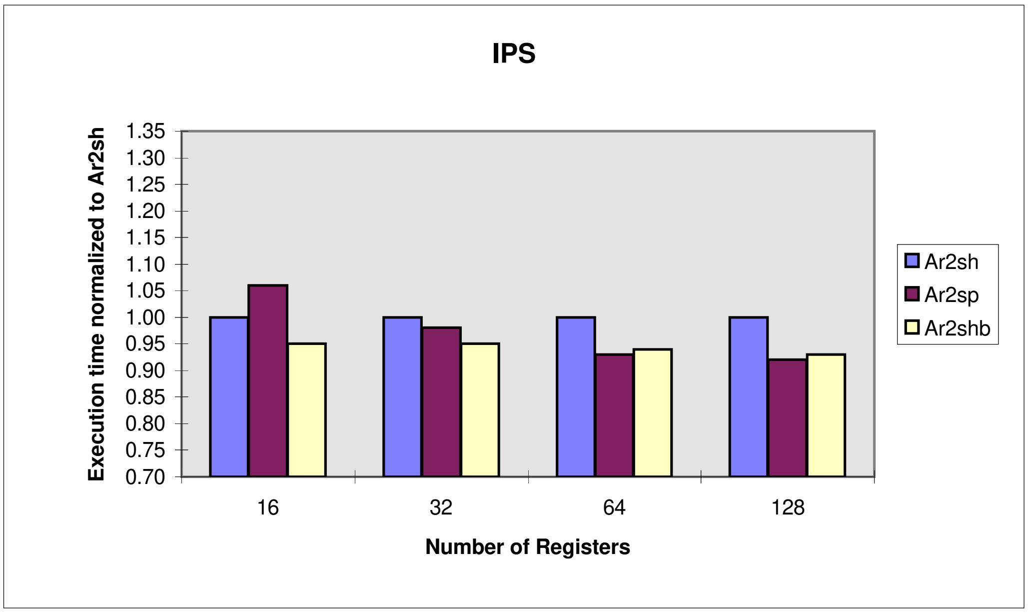



Figure 7.5: **Summary of the effect of register set organization (IPS)**. This graph summarizes the results from Section 7.2.2. It shows the harmonic mean of the normalized execution times of all four floating point intensive program groups for three architectures for IPS. Times are normalized to Ar2sh with the same number of registers.

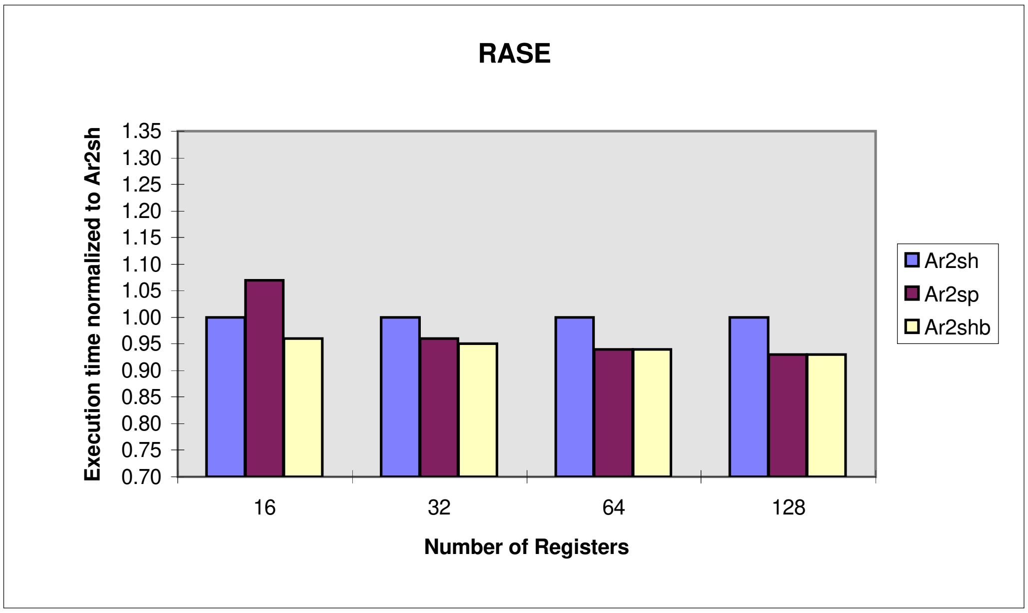



Figure 7.6: **Summary of the effect of register set organization (RASE)**. This graph summarizes the results from Section 7.2.2. It shows the harmonic mean of the normalized execution times of all four floating point intensive program groups for three architectures for RASE. Times are normalized to Ar2sh with the same number of registers.

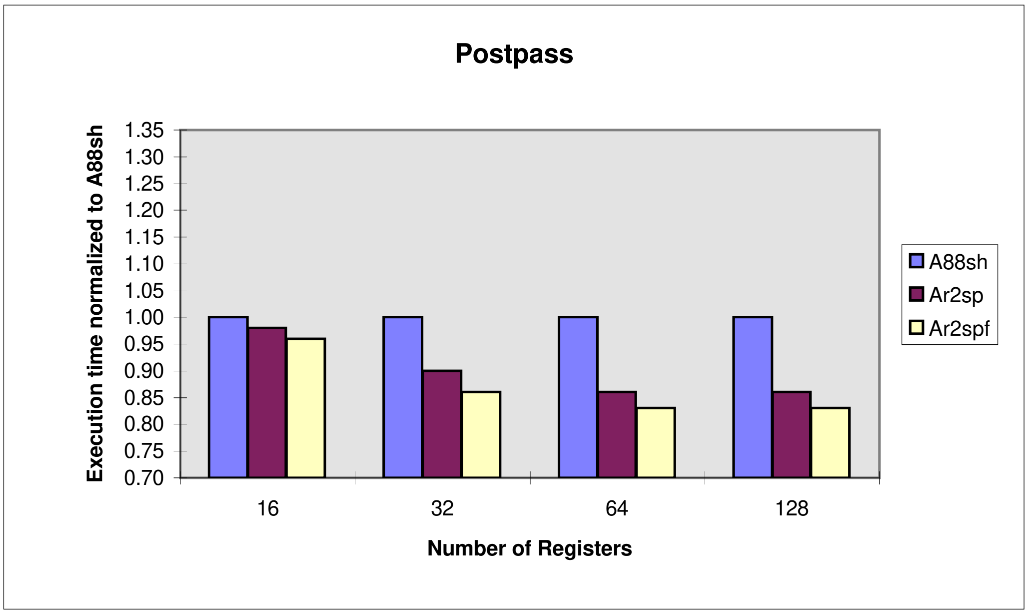



Figure 7.7: **Summary of the effect of operation latency (Postpass)**. This graph summarizes the results from Section 7.2.3. It shows the harmonic mean of the normalized execution times of all four floating point intensive program groups for three architectures for Postpass. Times are normalized to A88sh with the same number of registers.

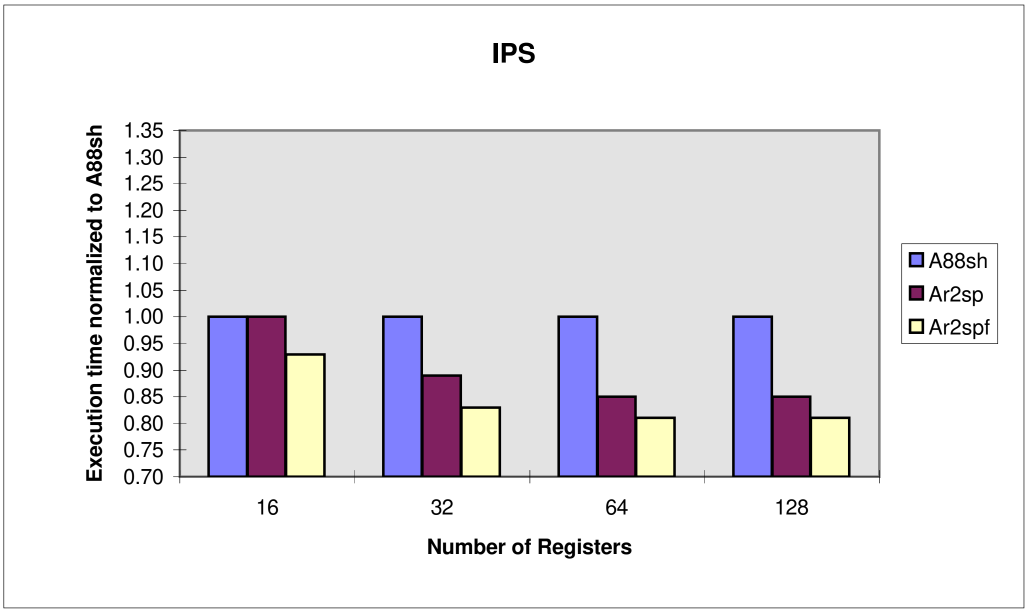



Figure 7.8: **Summary of the effect of operation latency (IPS).** This graph summarizes the results from Section 7.2.3. It shows the harmonic mean of the normalized execution times of all four floating point intensive program groups for three architectures for IPS. Times are normalized to A88sh with the same number of registers.

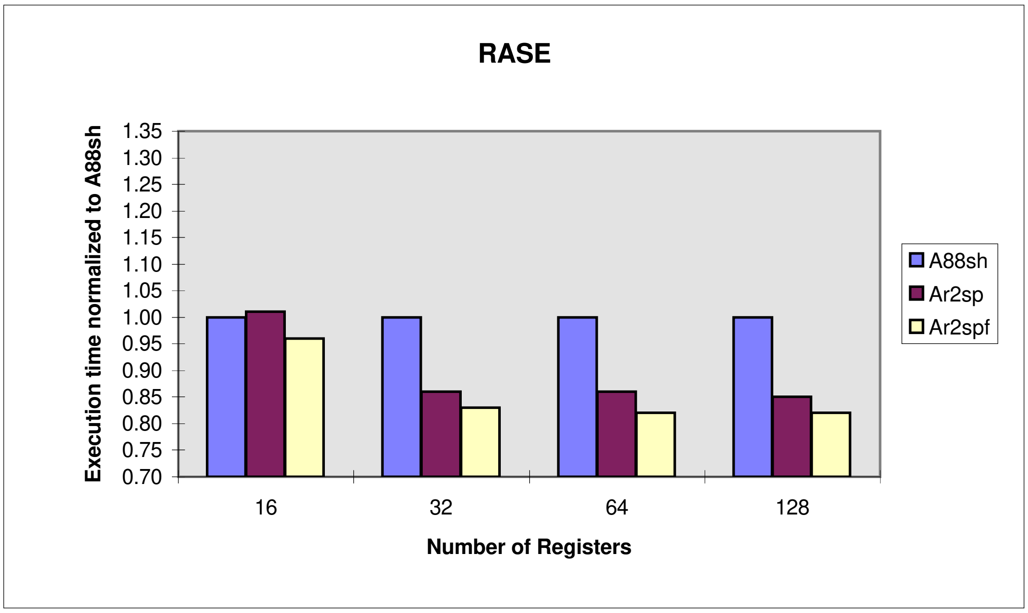



Figure 7.9: **Summary of the effect of operation latency (RASE).** This graph summarizes the results from Section 7.2.3. It shows the harmonic mean of the normalized execution times of all four floating point intensive program groups for three architectures for RASE. Times are normalized to A88sh with the same number of registers.

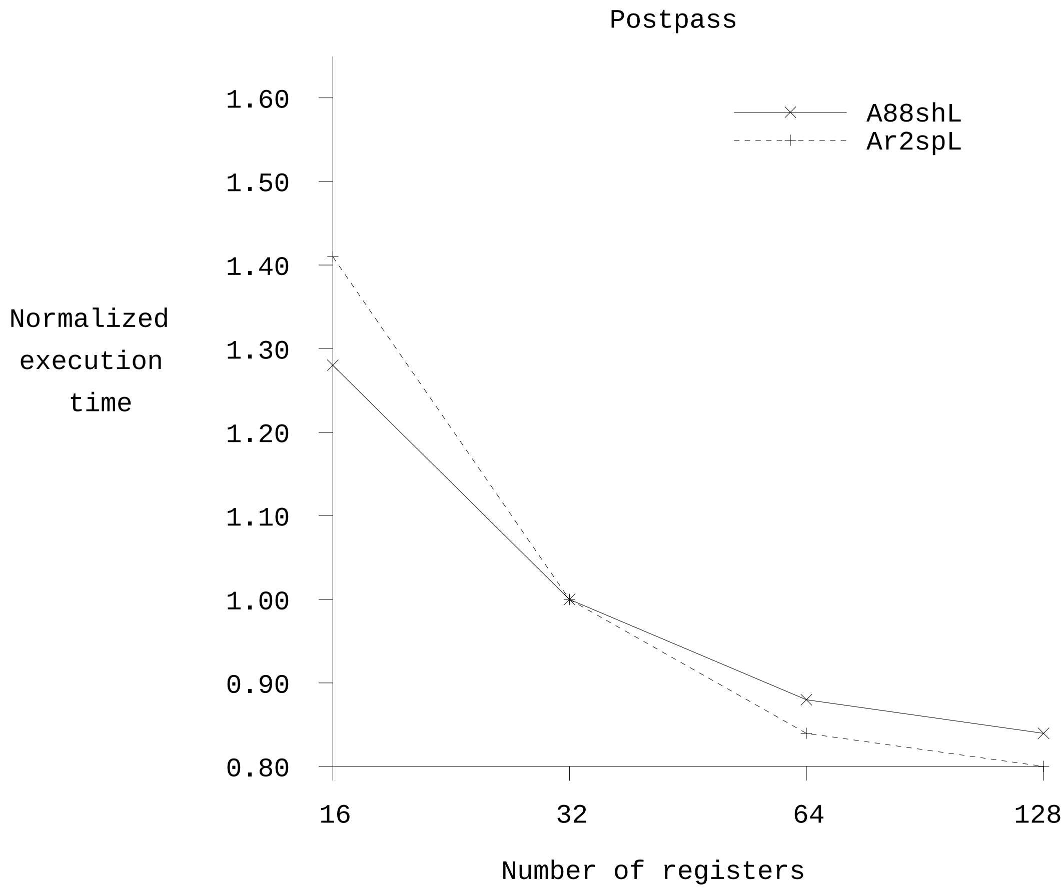



Figure 7.10: **Summary of the effect of register set size with long load latency (Postpass).** This graph summarizes the results from Section 7.2.4. It shows the harmonic mean of the normalized execution times of all four floating point intensive program groups for the two long load latency architectures for Postpass. Times are normalized to 32 registers with the same architecture. The y-axis range for this graph and the following two graphs differs from the previous graphs.

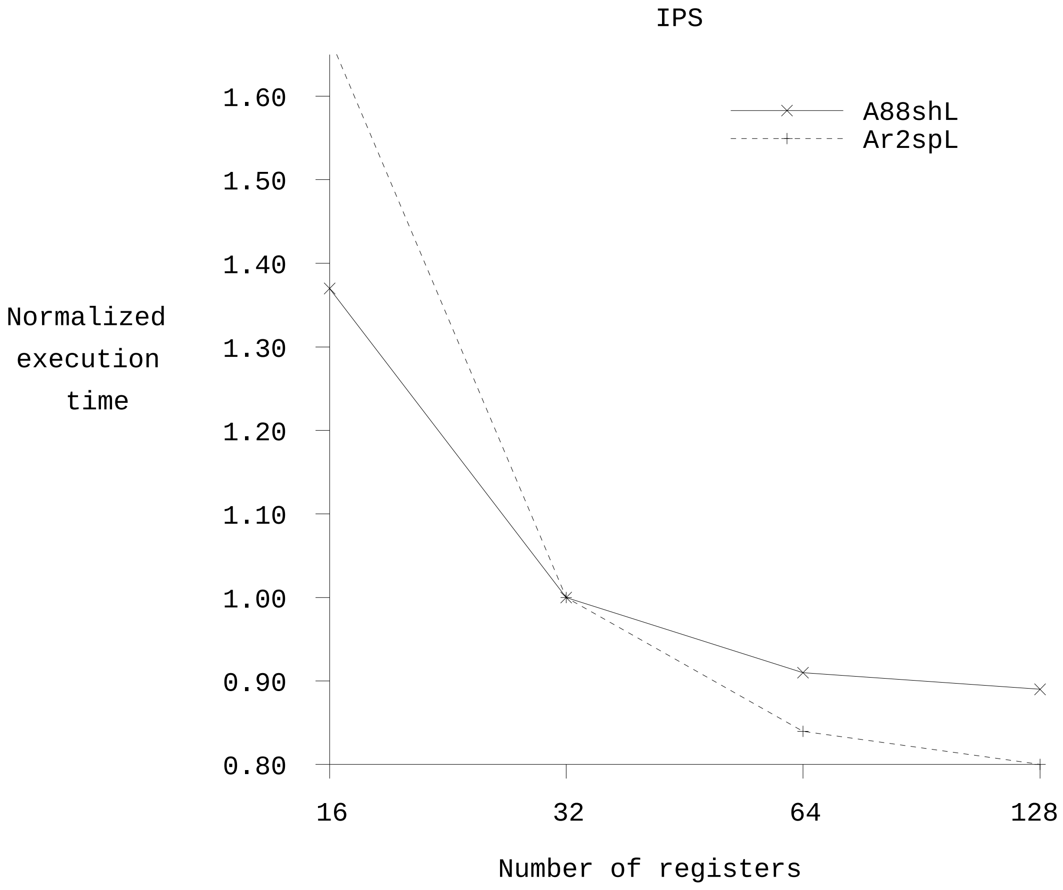



Figure 7.11: **Summary of the effect of register set size with long load latency (IPS).** This graph summarizes the results from Section 7.2.4. It shows the harmonic mean of the normalized execution times of all four floating point intensive program groups for the two long load latency architectures for IPS. Times are normalized to 32 registers with the same architecture.

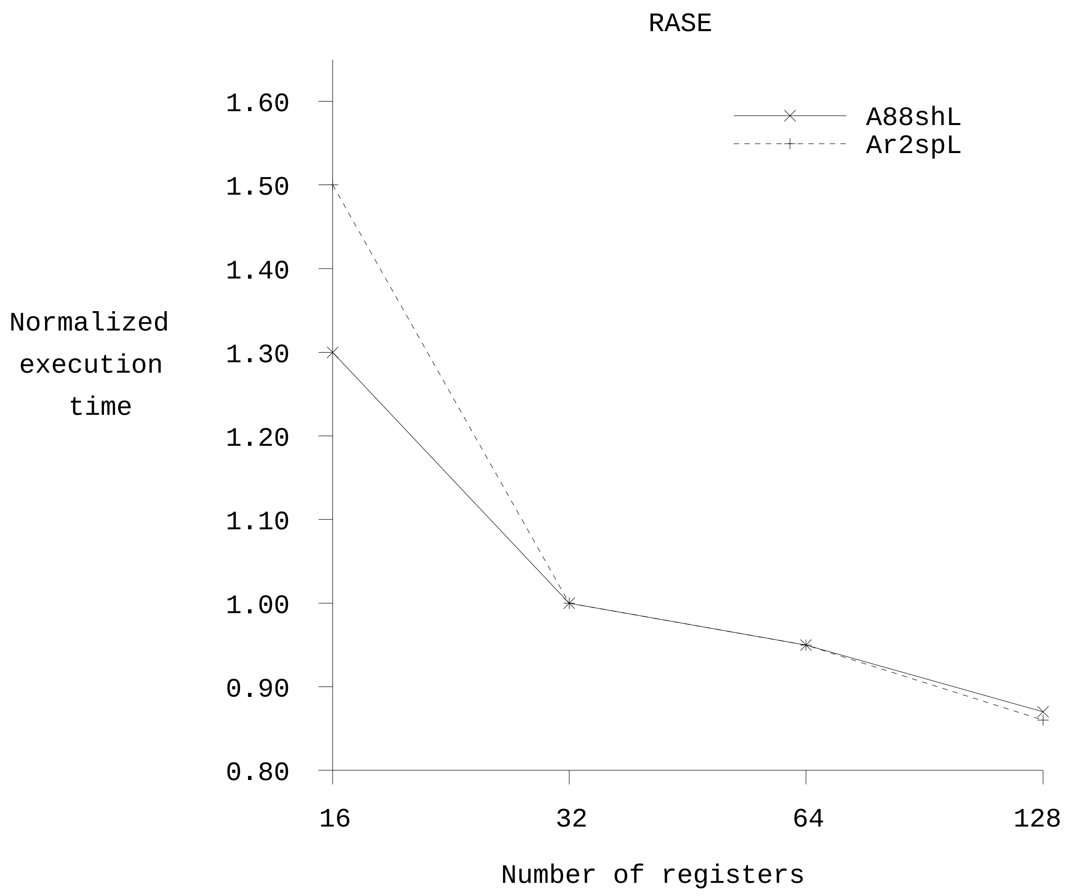



Figure 7.12: **Summary of the effect of register set size with long load latency (RASE).** This graph summarizes the results from Section 7.2.4. It shows the harmonic mean of the normalized execution times of all four floating point intensive program groups for the two long load latency architectures for RASE. Times are normalized to 32 registers with the same architecture.

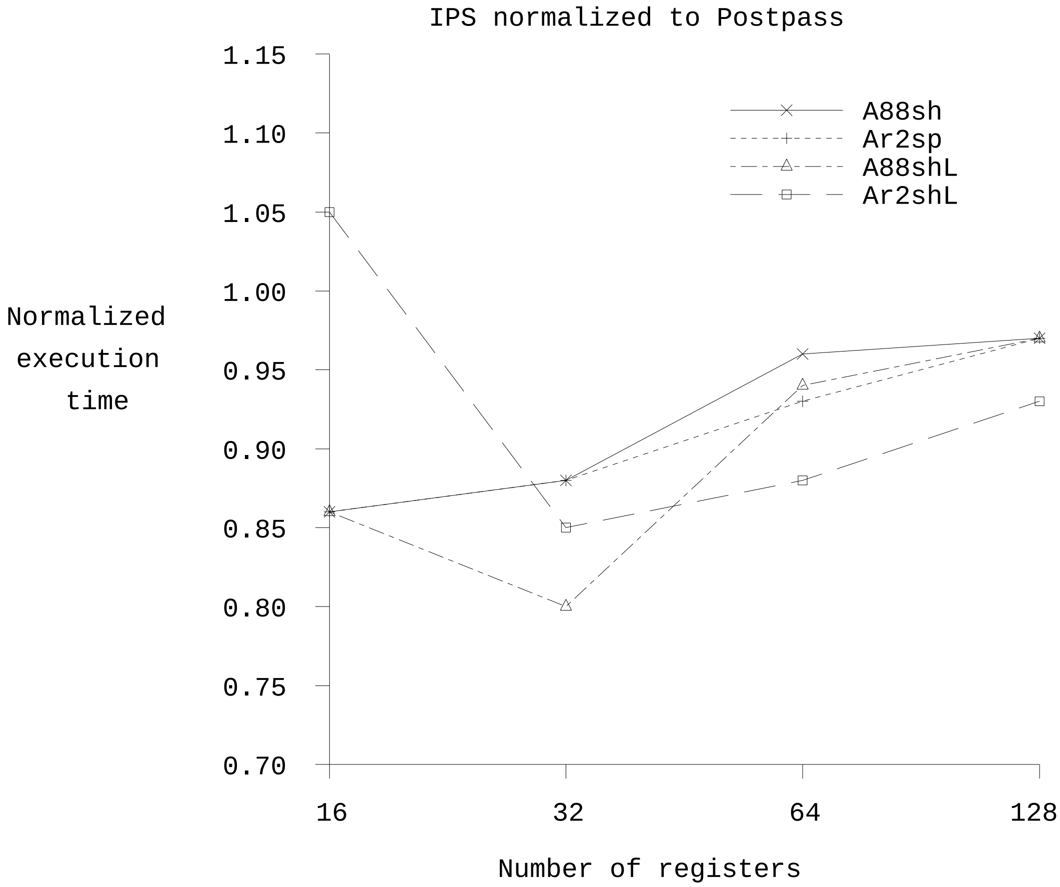



Figure 7.13: **Summary of the performance increase of IPS over Postpass**. This graph summarizes the results from Section 7.2.5. It shows the harmonic mean of the normalized execution times of all four floating point intensive program groups for four architectures for IPS normalized to Postpass. One would expect the performance improvement of IPS over Postpass to increase as register set size decreases; however, for the load load latency architectures with 16 registers IPS performs poorly, because it too heavily favors global pseudo-registers over locals. The y-axis range for this graph and the following graph differs from the previous graphs.

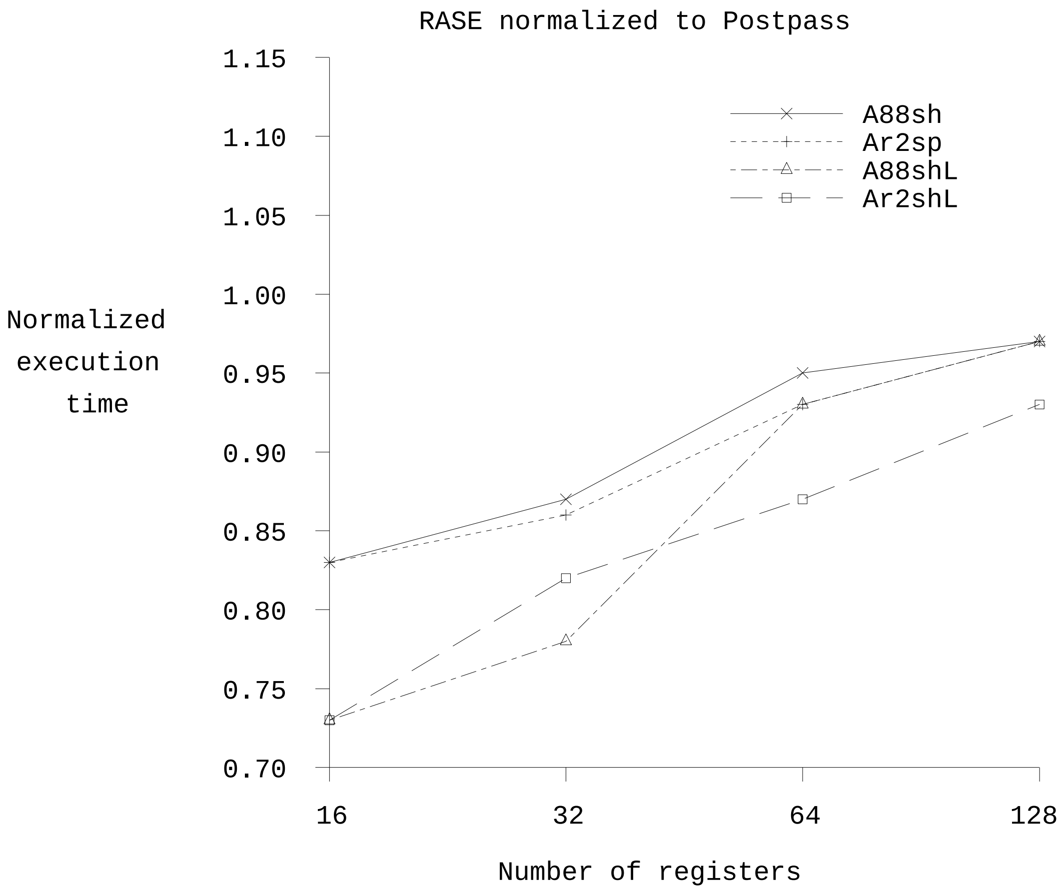



Figure 7.14: **Summary of the performance increase of RASE over Postpass**. This graph summarizes the results from Section 7.2.5. It shows the harmonic mean of the normalized execution times of all four floating point intensive program groups for four architectures for RASE normalized to Postpass. The performance improvement of RASE over Postpass continues to increase as register set size decreases, even on the long load latency architectures with 16 registers. This is because RASE is able to balance global and local register needs.

# Chapter 8

# Evaluation

This chapter evaluates the Marion system. The first section examines the machine descriptions for three targets and discusses the process of retargeting. The second section discusses system size and performance. The third section discusses Marion's coverage of common RISC features.

## 8.1 Machine Descriptions

Marion evolved in the same way as most other retargetable code generators. Although I initially attempted to design a broad description language, I had to extend and modify Maril as each significant target was implemented. The extensions for the R2000 and 88000 were minor, but, because I wanted to exploit most of the i860's unique features, including explicitly advanced pipelines and chaining, that architecture forced me to rethink the model and led me to add classes and temporal scheduling to Marion, as described in Chapter 4.

It may seem that building a retargetable code generator for RISCs would be easier than one for CISCs. RISCs have simplified some aspects of code generation, particularly code selection; however, instruction scheduling and its interaction with register allocation have added new complications. Support for register pairs and the pipeline timing and interconnect anomalies found on some RISCs increase the complexity. The challenge was to design a practical machine description language that could encapsulate as many of an architecture's scheduling and register details as possible, and build a system that could maintain and use that information to make judicious scheduling and register allocation choices.

The Maril description language is rich enough to encapsulate most scheduling information for RISCs. I have used it to build code generators for three commercially-available RISCs and for numerous architectural variations. Although Maril is not as succinct as possible, the descriptions for the R2000 and 88000 were easy to write. Modeling the i860 was more difficult, due to the complexity of its architecture. Table 8.1 shows the size and composition of the machine descriptions for these architectures. Instruction directives constitute the bulk of the descriptions and much of the instruction information is replicated, since most instructions' scheduling properties fall into one of a few categories. The use of a macro preprocessor could significantly reduce description size.

Using Maril to build a new target involves three steps. First, the compiler writer identifies the features of the target machine, including registers, pipelines and latencies. This typically involves scrutinizing the architecture manual or the programmer's reference manual and conceptualizing the machine model as the compiler will see it. If the target has multiple instruction issue or explicitly advanced pipelines, then the compiler writer should determine, in this step, how to model those features. For example, for explicitly advanced pipelines, the compiler writer must determine which

Table 8.1: **Maril machine description statistics**. The first part shows the section size (in number of non-blank non-comment lines). The second part gives the number of items of a each kind.

| | Section size (lines) | | |
|---|---|---|---|
| | 88000 | R2000 | i860 |
| Declare | 16 | 17 | 251 |
| Cwvm | 14 | 16 | 21 |
| Instr | 582 | 536 | 750 |
| Total | 612 | 569 | 1022 |

| | Number of items | | |
|---|---|---|---|
| | 88000 | R2000 | i860 |
| Instructions | 125 | 120 | 167 |
| Clocks | 0 | 0 | 4 |
| Elements | 0 | 0 | 140 |
| Classes | 0 | 0 | 67 |
| Auxiliary latencies | 6 | 0 | 12 |
| Glue transformations | 29 | 18 | 27 |
| Escape functions | 1 | 2 | 7 |
| Escape function C lines | 17 | 30 | 399 |

instructions control which pipelines.

The second step is to express the machine model in Maril. This is primarily a matter of declaring registers and resources, determining calling conventions and listing each instruction, along with its latency and resource requirements. It is important to make sure that registers and instructions are typed correctly. All datatypes supported in the IL must be represented by the register set. In addition, if an instruction performs an operation on values of some datatype, but the instruction's register operands hold other datatypes, type constraints should be used to restrict the permissible datatypes. The compiler writer should also check that resources used to control multiple instruction issue adhere to the machine model.

The third step is to complete the mapping between the IL and the target machine. This involves determining which IL constructs are not directly covered by the target, either by inspection or by running a set of test programs. Conversions and conditional branches usually require *glue* transformations and/or dummy instructions. If there is some "normal" language operation, such as divide or remainder, that is not directly supported by the target, then either a *glue* transformation or escape function is necessary. Other things to look for are operations that require multiple instructions, such as loads, stores or register moves. In addition, 32-bit constants and addresses usually require *glue* transformations. See Chapter 3 for details on these Maril constructs.

## 8.2 System Size and Performance

Marion was designed to support different scheduling algorithms and code generation strategies. The scheduling infrastructure can support most list scheduling algorithms. Other algorithm types

Table 8.2: **Marion system source code size** (in lines of C code). The figures for target- and strategy-independent do not include the front end, Each target-dependent portion is automatically produced by the code generator generator.

| Phase | Lines |
|---|---|
| Code Generator Generator | 4991 |
| Target- and strategy-independent | 10877 |
| Target-dependent, 88000 | 6864 |
| Target-dependent, R2000 | 5512 |
| Target-dependent, i860 | 8492 |
| Strategy-dependent, Postpass | 151 |
| Strategy-dependent, IPS | 1269 |
| Strategy-dependent, RASE | 3750 |

could be added with minor additions. I implemented three significantly different code generation strategies, Postpass, IPS and RASE. IPS was the last to be implemented, and took only one (expert) person-week.

Table 8.2 shows the size of the Marion system source code. The target-dependent portion that is output by the code generator is large, but roughly 75% is initialization code to construct pattern trees. The target- and strategy-independent portion includes the *glue* transformer, code selector, global register allocator, code DAG builder and scheduling support.

Marion compilers are not fast. Marion is a prototype system designed to show that retargetable instruction scheduling is feasible. It was not built for compile-time efficiency; instead it employs the "best simple" implementation whenever possible. For example, most algorithms use lists and naive search techniques. The register allocator's interference graph is implemented entirely with lists, instead of partially with a bit matrix (as Chaitin suggests [Cha82]). Also, resource vectors accommodate the longest operation, typically divide, but few instructions use more than a fraction of the vector; no attempt is made to limit resource vector comparisons to the relevant fraction. Improvements in these areas, along with more efficient data structures in other areas, would improve compile-time substantially.

Table 8.3 contains the execution times of the front end and Marion-built R2000 and i860 code generators when compiling a program suite on a DECstation 5000. The program suite is a subset of the workload described in Chapter 5. It includes the Nasker group, ARC2D from the Perfect group, SPHOT from the Misc group and Lcc from the Int group. "Dilation" is the ratio of instructions executed to instructions generated. The compile-time of all code generation strategies could be improved; the significance of the data is the relative speed of the strategies. IPS takes longer than Postpass, because it schedules each block twice and its scheduler is more complicated than Postpass's. RASE takes even longer; in effect it schedules four times (three times during pre-scheduling) and its scheduler is more complicated than IPS's. Because the i860 requires extensive use of temporal registers and classes, and because floating point operations are split into sub-operations, compiling for the i860 takes roughly twice as long as for the R2000.

Table 8.4 shows the percentage of total compilation time spent in each of the three portions of the Marion R2000 code generators (each with a different code generation strategy). Postpass

Table 8.3: **Marion compile-time performance.** Time spent in the front end, the Marion code generators and the MIPS host C compiler, when compiling the program suite for the R2000 and the i860. MIPS C is a production compiler; with the -O1 option, it performs local optimizations, but no global register allocation. All compilers were run on a 25Mhz DECstation 5000.

| R2000 target | | |
|---|---|---|
| Module | User time (secs) | Dilation ($\times 10^4$) |
| Lcc front end | 31 | 0.95 |
| Marion, Postpass | 989 | 28.49 |
| Marion, IPS | 1846 | 58.16 |
| Marion, RASE | 5969 | 192.60 |
| MIPS C -O1 | 54 | 1.66 |
| MIPS assembler | 32 | 0.99 |

| i860 target | | |
|---|---|---|
| Module | User time (secs) | Dilation ($\times 10^4$) |
| Lcc front end | 33 | 0.85 |
| Marion, Postpass | 1778 | 43.52 |
| Marion, IPS | 3128 | 83.80 |
| Marion, RASE | 10746 | 285.60 |

Table 8.4: **Marion code generator profile.** Percentage of compilation time spent in code generator portions for three Marion R2000 code generators.

| | Time fraction, % | | |
|---|---|---|---|
| Phase | Postpass | IPS | RASE |
| Target- & strategy-independent | 91.9 | 87.1 | 65.2 |
| Target-dependent | 4.3 | 3.3 | 2.6 |
| Strategy-dependent | 0.6 | 7.8 | 31.4 |

spends almost all of its time in target- and strategy-independent code, because the strategy itself is simple. In contrast, strategy-dependent code accounts for nearly 8% of IPS's time and over 31% of RASE's time.

For all code generators, roughly 13% of total time is spent manipulating resource vectors. Each resource vector is an array of 32-bit integers. The array size is the length of the target's longest latency. Each integer represents a cycle; each bit represents a resource. When comparing vectors, Marion makes no attempt to limit the the number of elements it examines.

Register allocation takes between 25% and 32% of total time, and code DAG building takes over 20% of total time for Postpass and IPS, but only 6% for RASE. RASE spends almost a third of its time in strategy-dependent code, most of it in the scheduler; for the R2000 RASE schedules three times during pre-scheduling plus once during final scheduling. Therefore, improving the compile-time performance of register allocation and resource vector manipulation (both target- and strategy-independent) would have a greater impact on Postpass and IPS than RASE. However, an improved scheduling implementation, for example using an ordered ready list and doing less recomputation,

Table 8.5: **Actual execution time and ratio of actual to estimated execution time of Marion-generated R2000 code** for the Livermore Loops. The execution time is in user time seconds. The mean is arithmetic for execution times and harmonic for ratios.

| Kernel | Exec time (secs) | | | Actual/Estimated | | |
|---|---|---|---|---|---|---|
| | Postp | IPS | RASE | Postp | IPS | RASE |
| 1 | 2.22 | 2.22 | 2.22 | 1.13 | 1.13 | 1.13 |
| 2 | 2.40 | 1.98 | 1.98 | 1.11 | 1.02 | 1.02 |
| 3 | 1.82 | 1.82 | 1.82 | 1.11 | 1.10 | 1.10 |
| 4 | 1.32 | 1.32 | 1.32 | 0.99 | 0.99 | 0.99 |
| 5 | 2.09 | 2.08 | 2.09 | 1.05 | 1.05 | 1.05 |
| 6 | 1.31 | 1.32 | 1.31 | 1.04 | 1.05 | 1.05 |
| 7 | 2.71 | 2.73 | 2.68 | 1.06 | 1.13 | 1.12 |
| 8 | 4.24 | 4.21 | 4.25 | 1.09 | 1.13 | 1.13 |
| 9 | 3.19 | 3.12 | 3.19 | 1.13 | 1.13 | 1.15 |
| 10 | 4.73 | 4.76 | 4.76 | 1.09 | 1.10 | 1.10 |
| 11 | 1.72 | 1.72 | 1.73 | 1.01 | 1.01 | 1.01 |
| 12 | 1.83 | 1.83 | 1.83 | 1.01 | 1.01 | 1.01 |
| 13 | 2.97 | 2.93 | 2.93 | 1.02 | 1.02 | 1.02 |
| 14 | 3.20 | 3.14 | 3.13 | 1.04 | 1.03 | 1.03 |
| Mean | 2.38 | 2.35 | 2.35 | 1.06 | 1.06 | 1.06 |

would have a greater impact on RASE's compile-time efficiency.

The performance data for the code generation strategies comparison is derived by combining basic block execution costs computed by each scheduler with execution frequencies computed by a separate profiling tool. Therefore, activity outside of the CPU, such as cache misses, is not considered; Marion assumes all memory references are cache hits. Table 8.5 compares the estimated execution cycles, computed by Marion schedulers, to the actual execution cycles for the first fourteen Livermore Loop kernels. Marion-produced code is assembled by the MIPS assembler and linked with the MIPS libraries. The actual cycles are computed by timing the execution on a DECstation 5000 and multiplying by the clock rate, which is 25MHz. The machine specification used to produce executable code differs in three ways from the one used in the code generation strategies comparison, because of the need to interface with the assembler:

(1) The MIPS C calling conventions are followed [Kan87]. None of the registers that MIPS reserves for the assembler and operating system are reserved in the strategies comparison. (See Chapter 6.) Therefore, the register allocators in the strategies comparison have more allocable registers.

(2) The *la* pseudo-instruction is used to load a 32-bit relocatable address into a register. In addition, other load and store instructions have 32-bit relocatable address operands that the assembler maps into 2-instruction sequences. In the strategies comparison 32-bit relocatables are broken into high and low halves, so that the two instructions can be individually scheduled.

(3) The global data pointer is not used. The code generators in the strategies comparison use a global data pointer, which decreases the cost (from two instructions to one) of accessing static data.

An R2000 assembly language formatter takes Marion's representation of target machine instructions and produces assembly code. This conversion routine was written by hand.

The ratio of actual time to estimated time varies between loops, because each loop has a different memory usage pattern. However, for a given loop, the ratio is consistent across strategies, with a few exceptions. Postpass's ratio is lower than IPS's or RASE's for Loops 7 and 8; in both cases Postpass's load instruction order is different from the load instruction order produced by either of the other strategies, which could cause different cache miss ratios. For Loop 2, Postpass's ratio is higher, but the load instruction order is the same. The higher ratio could be caused because no floating operation precedes any load, so that all such operations are delayed on a cache miss. A floating point operation that is initiated before a load does not stall when the CPU stalls.

For the Livermore Loops RASE-generated code was 26% faster than code produced by MIPS C -O1, which performs only local optimizations; code produced by MIPS C -O2, which performs global register allocation and global optimizations, was 42% faster than RASE code. This makes sense, because Marion compilers perform global register allocation, but no global optimizations. Also, the Marion R2000 compiler that produces executable code suffers because it must interface with the assembler.

## 8.3 Support for RISC Architectures

*Completeness* is a desirable quality of a retargetable system's machine description language. Completeness means that the language can model all machines in a particular class. Maril is not yet complete over the RISC class, but can model most RISC features. To get a perspective, the next few paragraphs list common architectural features and some of the commercially available RISC machines that contain those features. Each paragraph discusses how and whether Marion can support the feature. The feature is in boldface and the machines are in parentheses.

**Split register file** (R2000, i860, SPARC, IBM RS/6000 [OG90]): In the machine description the compiler-writer can declare any number of register sets. Special purpose registers can also be declared.

**Register pairs** (88000, i860, SPARC): The compiler writer can declare a register set to represent the pairs and indicate that the set overlays another register set with the **%equiv** directive. Escape functions allow halves of registers to be accessed.

**Register windows** (SPARC): A unique feature of the SPARC is its register windows. A *register window* is the subset of the machine's total register set that is visible by a particular procedure. Typically, when another procedure is called, the new procedure shifts the window, in effect, saving the previous procedure's registers. The SPARC uses register windows only for integer registers. Each window contains 24 registers, 8 *ins*, which overlap with the previous procedure's window, 8 *scratch* registers, which overlap with no other window, and 8 *outs*, which overlap with the window of a procedure that may be called by the current procedure. A procedure passes parameters in registers by placing them in *outs*. The called procedure sees them in its *ins*.

Marion does not currently support register windows, because changing windows in effect saves and renames the registers: the old *out*s become the new *in*s. Maril's **CWVM** could be extended to specify parameters and arguments separately, which models the register renaming, and to specify calleesave and callersave registers separately, which supports hidden register saving.

**Hard-wired registers** (R2000, 88000, i860, SPARC): Hard-wired registers, *i.e.*, those that always contain a constant, are supported by the **%hard** directive.

**Structural hazards** (R2000, 88000, i860, SPARC, RS/6000): Resource vectors declared with each instruction are checked during scheduling. This avoids structural hazards.

**Hardware priority scheme for resource contention** (88000): Marion does not support the 88000's write-back bus priority scheme. (See Section 4.3.) Adding Maril language features to accommodate this scheme would not be difficult; a priority range could be attached to pertinent resources and an instruction could indicate its priority for using that resource. However, it would be expensive to check the priorities during scheduling. Instead, Marion gives priority to the instruction that is scheduled first.

**Two-instruction double float load** (R2000): The escape mechanism allows the 2-instruction sequence to be generated, with each instruction accessing half of the double precision register. The two loads are then scheduled as separate instructions.

**Branch delay slots** (R2000, 88000, i860, SPARC): Each instruction directive indicates how many delay slots follow the instruction. Currently these slots are filled with no-ops, but Gross and Hennessy's method for filling them [GH82] could be implemented with the information available from the machine description.

**Branch delay slots conditionally executed** (i860, SPARC): A negative delay slot value in an instruction directive indicates the instruction in the slot is executed only if the branch is taken.

**Condition code register** (i860, SPARC, RS/6000): The condition code register can be declared as a separate temporal register (see Section 4.5); instruction directives can indicate that the result of a comparison is stored in this register. However, separate directives must be listed in the machine description for instructions that set the condition code as a side effect *e.g.*, subtract. Therefore, two instructions can sometimes be generated where one would suffice.

**Set-on-condition instruction** (R2000): The set-on-condition instruction represents a default if-then-else construct. An if-then-else construct encompasses multiple basic blocks. Because the IL presented to the Marion code selector is low-level, and, therefore, does not contain the procedure's control structure, the code selector cannot generate this instruction. A higher-level IL would help.

**Conditional call instruction** (R2000): Like the set-on-condition, the conditional call represents an IL construct that spans basic blocks. Therefore the code selector cannot generate it.

**Latency dependent on consumer instruction** (88000, i860): The auxiliary latency directive (**%aux**) specifies a pair of instructions with constraints and a latency. If the constraints are satisfied, a code DAG edge between two such instructions is labeled with the auxiliary latency. (See Section 3.3.3.)

**Multiple instruction issue** (i860, RS/6000): Marion supports multiple instruction issue in which the instruction set is partitioned such that an instruction from each partition can be issued per cycle. This can model the i860's multiple instruction issue and should be able to model the IBM RS/6000's as well. Marion does not support multiple identical functional units; introducing arrays of resources would be a natural extension, but would require resource scheduling. This extension alone would not support identical functional units, which each take inputs from a different register set.

**Loop decrement and branch instruction** (i860): Marion cannot support loop decrement and branch for two reasons. First, the simple pattern matcher cannot match a single instruction to a pair of IL constructs (the decrement and the branch). Second, Marion currently does not recognize loops in the IL.

**Multiple addressing modes** (88000, i860, RS/6000): The Marion code selector searches an ordered list of patterns that corresponding to instructions. Ordering the instructions that contain addressing modes, (*e.g.*, putting *register + displacement* before *register + register*), is sufficient to generate instructions with the desired addressing modes.

**Auto-increment addressing mode** (i860, RS/6000): Again, the auto-increment would require a single instruction to match a pair of IL constructs. In addition, selecting this address mode affects scheduling and register allocation opportunities [HO91].

**Bypass cache load** (i860): Here the problem is choosing between the normal load, which accesses the cache, and the load that bypasses the cache. Marion has no information indicating when it is profitable to bypass the cache. Marion always uses the normal load.

**Explicitly advanced pipelines** (i860): As discussed in Chapter 4, Marion supports explicitly advanced pipelines, including chaining between them.

Although Marion can represent most RISC features, there are a number of notable exceptions. These include the 88000's resource contention priority scheme, the SPARC's register windows, instruction side effects, such as setting the condition code, and general multiple instruction issue. Natural extensions could handle register windows and more general multiple instruction issue. Side effects could be handled by peephole optimization or more sophisticated code selection, but the interaction of these effects with instruction scheduling is yet unexplored. Efficiently modeling a resource contention priority scheme is difficult. Also, instructions representing IL constructs that span basic block boundaries cannot be generated without a higher-level IL or special case pattern matching.

From the compiler's point of view, by far the most difficult RISC feature to handle is explicitly advanced pipelines. Considerable effort was required to develop and implement temporal scheduling. In addition, temporal scheduling slows compilation speed by nearly a factor of 2. A second difficult feature is the priority scheme for resource contention. In effect, this feature means that the cycle on which an instruction uses a resource can vary, which complicates the scheduler's structural hazard avoidance.

In general, the R2000 is an easy target. There are few irregularities and no register pairs. A pipelined FPU and allowing an integer and floating point instruction to be issued on the same cycle would increase performance, without significant additional burden on the compiler. The R2000's set-on-condition instruction requires a higher-level transformation than Marion supports, but is advantageous because it avoids branching.

# Chapter 9

# Conclusion

The Marion system is a retargetable code generation system designed specifically for RISCs. The machine description language has primitives to describe most instruction scheduling requirements and can model a broad range of RISCs, including some superscalars. Marion cannot handle all details perfectly, but can model most of the important features required to produce a good instruction scheduler as part of a complete code generator. Marion can also model complicated features found in some RISCs, such as explicitly advanced pipelines and irregular packing restrictions.

Marion has been used to construct compilers for the MIPS R2000, the Motorola 88000 and the Intel i860. The machine descriptions for the R2000 and the 88000 use many of Marion's features, including resource vectors, auxiliary latencies, glue transformations and escape functions. The i860 target also uses the above features plus temporal scheduling with chaining, instruction classes and resource vectors to control multiple instruction issue.

This dissertation has investigated the interaction of instruction scheduling and register allocation by comparing three code generation strategies, each of which represents a different degree of communication between the two phases. The results indicate that at least an intermediate degree of integration is needed to produce efficient schedules. In particular, the code produced by the intermediate strategy IPS for floating point programs on the 88000, is, on the average, 13% faster than code produced by the Postpass strategy, which has no communication between the two phases. The RASE strategy, which closely couples the two phases, also exhibits a significant performance improvement over Postpass (averaging 15% on the 88000), but little over IPS on most programs and most architectures. However, on programs with large basic blocks and on architectures with small register sets or long load latencies, RASE is able to find a better balance between using registers for instruction scheduling and using them to reduce memory references. This results in better code than IPS. For example, on the 88000 for four of the Perfect group programs, RASE-produced code is 7% to 13% faster than IPS-produced code.

Additional experiments have examined the effect on performance of architectural features, including register set size and structure and load and operation latencies, versus code generation strategies. These experiments reinforce the results of the code generation strategies comparison. They also show that sophisticated code generation strategies can give computer architects more flexibility in making trade-offs between register set size and other components, because these strategies use registers more effectively.

Marion supports a broad range of RISCs, including the complicated i860. In addition, Marion was the enabling factor for the code generation strategies comparison and the experiments with architectural variations. Taken together, this demonstrates that retargetable instruction scheduling for RISCs is feasible and profitable.

## 9.1 A Production System

Marion is a prototype retargetable code generation system, but with additional work a production system could be derived. The additional work is in six areas, most of which are well-understood.

The first is global optimizations. To become a production system, the Marion front end needs standard global optimizations, particularly loop induction variable analysis, strength reduction and invariant removal.

The second area is improving compile-time performance. Marion takes a substantial amount of time to compile large programs. Much of this is due to the use of simple data structures and algorithms in the register allocator and scheduling support routines. A reimplementation using more efficient techniques should speed up the compiler, but compile-time will still differ between the code generation strategies: Postpass will remain the fastest and RASE will remain the slowest.

Third, Marion needs to fill branch delay slots with useful instructions. An implementation of Gross and Hennessy's algorithm [GH82] would meet this requirement.

Fourth, unsigned datatypes need to be added to Maril. This should be straight-forward.

Fifth, a method for exploiting side effects, such as condition codes, would be useful. This is not as important as the others, but a number of RISCs have condition codes. More sophisticated code selection or simple peephole optimization would be sufficient.

The sixth area is probably the most difficult, but also one of the most important. Maril should be improved to handle multiple identical functional units and register windows. In addition, more architectures should be examined for other important features. Adding new constructs to Maril is not that difficult, but it is important to keep the language succinct, so that eliciting the desired code from a machine description is not a nightmare.

## 9.2 Future Directions

A number of techniques that increase instruction-level parallelism have been developed. These include loop unrolling, trace scheduling and various forms of software pipelining. Bernstein and Rodeh [BR91] have recently developed another method, which schedules across basic block boundaries using the program dependence graph [FOW87] as the primary data structure. Enhancing Marion with one of these methods would enable further experiments to gauge the effect on performance of architectural variations and code generation strategies with respect to increased instruction-level parallelism. In addition, the interaction between these techniques and the other phases of code generation needs to be investigated.

Another area to explore is the interaction between code selection and instruction scheduling. Although code selection is simpler on RISCs than CISCs, there are opportunities to make code sequence choices. First, the evaluation order used by the code selector affects the register lifetime information that is presented to the register allocator. In addition, balanced expressions trees allow more scheduling opportunities, but at the expense of greater register pressure. A second opportunity exists on architectures where an operation can be performed by two different functional units. For example, on the R2000, if two integer multiplies need to be performed, it is faster to do one in the FPU and one in the integer multiply/divide unit, which is non-pipelined. However, if other floating point operations need to be performed, using the FPU for integer multiply may not be faster. The choice may also depend of where operands reside or how the result is used. A third opportunity is with instruction side effects, such as the condition code and auto-increment addressing mode. Code that exploits side effects can be more efficient, but can also limit scheduling choices. Finally, on architectures that support chaining between pipelines, such the i860, the code selector can decide to use the chaining. This is the case in Marion. Chaining reduces register pressure, but can limit scheduling opportunities, because the chained operations cannot be separated without involving the register allocator.

Marion provides a basis for exploring techniques to increase instruction-level parallelism and for investigating the interaction between various code generation phases.

# Bibliography


[ABB64] G. M. Amdahl, G. A. Blaauw, and F. P. Brooks, Jr. Architecture of the IBM System/360. *IBM Journal of Research and Development*, 8(2), April 1964.

[AGH+84] P. Aigrain, S. L. Graham, R. R. Henry, M. K. McKusick, and E. Pelegri-Llopart. Experience with a Graham-Glanville style code generator. *ACM SIGPLAN Symposium on Compiler Construction, SIGPLAN Notices*, 19(6), June 1984.

[AGT89] Alfred V. Aho, Mahadevan Ganapathi, and Steven W. K. Tjiang. Code generation using tree matching and dynamic programming. *ACM Transactions on Programming Languages and Systems*, 11(4), October 1989.

[AH82] Marc Auslander and Martin Hopkins. An overview of the PL.8 compiler. *ACM SIGPLAN Symposium on Compiler Construction, SIGPLAN Notices*, 17(6), June 1982.

[Ary85] Siamak Arya. An optimal instruction-scheduling model for a class of vector processors. *IEEE Transactions on Computers*, c-34, November 1985.

[BB84] D. Bailey and J. Barton. The NAS kernel benchmark program. NASA Technical Memorandum 87611, Ames Research Center, National Aeronautics and Space Administration, Moffet Field, CA, 1984.

[BCK+89] M. Berry, D. Chen, P. Koss, D. Kuck, S. Lo, Y. Pang, L. Pointer, R. Roloff, A. Sameh, E. Clementi, S.Chin, D. Schneider, G. Fox, P. Messina, D. Walker, C. Hsiung, J. Schwarzmeier, K. Lue, S. Orszag, F. Seidl, O. Johnson, R. Goodrum, and J. Martin. The PERFECT Club Benchmarks: Effective performance evaluation of supercomputers. *The International Journal of Supercomputer Applications*, 3(3), Fall 1989.

[BCKT89] Preston Briggs, Keith D. Cooper, Ken Kennedy, and Linda Torczon. Coloring heuristics for register allocation. *ACM SIGPLAN Conference on Programming Language Design and Implementation, SIGPLAN Notices*, 10(7), July 1989.

[BEH91a] David G. Bradlee, Susan J. Eggers, and Robert R. Henry. The effect on RISC performance of register set size and structure versus code generation strategy. *International Symposium on Computer Architecture*, May 1991.

[BEH91b] David G. Bradlee, Susan J. Eggers, and Robert R. Henry. Integrated register allocation and instruction scheduling for RISCs. *International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1991.

[BG89] David Bernstein and Izidor Gertner. Scheduling expressions on a pipelined processor with a maximal delay of one cycle. *ACM Transactions on Programming Languages and Systems*, 11(1), January 1989.

[BGM+89] D. Bernstein, M. C. Golumbic, Y. Mansour, R. Y. Pinter, D. Q. Goldin, H. Krawczyk, and I. Nahshon. Spill code minimization techniques for optimizing compilers. *ACM SIGPLAN Conference on Programming Language Design and Implementation, SIGPLAN Notices*, 24(7), July 1989.

[BHE91] David G. Bradlee, Robert R. Henry, and Susan J. Eggers. The Marion system for retargetable instruction scheduling. *ACM SIGPLAN Conference on Programming Language Design and Implementation, SIGPLAN Notices*, 26(7), July 1991.

[Bir87] Peter H. L. Bird. *Code Generation and Instruction Scheduling for Pipelined SISD Machines.* PhD thesis, University of Michigan, 1987.

[BN71] C. G. Bell and A. Newell. *Computer Structures: Readings and Examples.* McGraw Hill, New York, N. Y., 1971.

[BR91] David Bernstein and Michael Rodeh. Global instruction scheduling for superscalar machines. *ACM SIGPLAN Conference on Programming Language Design and Implementation, SIGPLAN Notices*, 26(7), July 1991.

[CAC+81] Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. Register allocation via coloring. *Computer Languages*, 6, 1981.

[Cat80] R. G. G. Cattell. Automatic derivation of code generators from machine descriptions. *ACM Transactions on Programming Languages and Systems*, 2(2), April 1980.

[CGLT89] Robert Cohn, Thomas Gross, Monica Lam, and P. S. Tseng. Architecture and compiler tradeoffs for a long instruction word microprocessor. *International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1989.

[CH84] Frederick C. Chow and John L. Hennessy. Register allocation by priority based coloring. *ACM SIGPLAN Symposium on Compiler Construction, SIGPLAN Notices*, 19(6), June 1984.

[CH90] Frederick C. Chow and John L. Hennessy. The priority-based coloring register allocation approach. *ACM Transactions on Programming Languages and Systems*, 12(4), October 1990.

[Cha81] A. E. Charlesworth. An approach to scientific array processing: The architectural design of the AP-120B/FPS-164 family. *IEEE Computer*, 14(9), September 1981.

[Cha82] Gregory J. Chaitin. Register allocation and spilling via graph coloring. *ACM SIGPLAN Symposium on Compiler Construction, SIGPLAN Notices*, 17(6), June 1982.

[Cho88] Frederick C. Chow. Minimizing register usage penalty at procedure calls. *ACM SIGPLAN Conference on Programming Language Design and Implementation, SIGPLAN Notices*, 23(7), July 1988.

[CNL79] R. G. G. Cattell, J. M. Newcomer, and B.W. Leverett. Code generation in a machine-independent compiler. *ACM SIGPLAN Symposium on Compiler Construction, SIGPLAN Notices*, 14(8), August 1979.

[Dav86] Jack W. Davidson. A retargetable instruction reorganizer. *ACM SIGPLAN Symposium on Compiler Construction, SIGPLAN Notices*, 21(7), July 1986.

[DF80] Jack W. Davidson and Christopher W. Fraser. The design and application of a retargetable peephole optimizer. *ACM Transactions on Programming Languages and Systems*, 2(2), April 1980.

[DF84a] Jack W. Davidson and Christopher W. Fraser. Automatic generation of peephole optimizations. *ACM SIGPLAN Symposium on Compiler Construction, SIGPLAN Notices*, 19(6), June 1984.

[DF84b] Jack W. Davidson and Christopher W. Fraser. Code selection through object code optimization. *ACM Transactions on Programming Languages and Systems*, 6(4), October 1984.

[DHB89] James C. Dehnert, Peter Y.-T. Hsu, and Joseph P. Bratt. Overlapped loop support in the Cydra 5. *International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1989.

[Dig81] Digital Equipment Corp., Maynard, Massachusetts. *DEC VAX Architecture Handbook*, 1981.

[DJ79] J. J. Dongarra and A. R. Jinds. Unrolling loops in Fortran. *Software Practice and Experience*, 9(3), March 1979.

[DLSM81] S. Davidson, D. Landskov, B. D. Shriver, and P. W. Mallett. Some experiments in local microcode compaction for horizontal machines. *IEEE Transactions on Computers*, c-30(7), July 1981.

[Ell85] John R. Ellis. *Bulldog: A Compiler for VLIW Architectures.* PhD thesis, Yale University, February 1985. Tech Report DCS/RR-364.

[FERN84] Joseph A. Fisher, John R. Ellis, John C. Ruttenberg, and Alexandru Nicolau. Parallel processing: A smart compiler and a dumb machine. *ACM SIGPLAN Symposium on Compiler Construction, SIGPLAN Notices*, 19(6), June 1984.

[FH90] Christopher W. Fraser and David R. Hanson. A code generation interface for ANSI C. Technical Report CS-TR-270-90, Dept. of Computer Science, Princeton University, July 1990.

[FH91] Christopher W. Fraser and Robert R. Henry. Hard-coding bottom-up code generation tables to save time and space. *Software Practice and Experience*, 21(1), January 1991. (also University of Washington Department of Computer Science and Engineering Technical Report 90-01-03, Seattle, Washington).

[Fis81] Joseph A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, c-30(7), July 1981.

[FOW87] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3), July 1987.

[FW86] Christopher W. Fraser and Alan L. Wendt. Integrating code generation and optimization. *ACM SIGPLAN Symposium on Compiler Construction, SIGPLAN Notices*, 21(7), July 1986.

[GF82] Mahadevan Ganapathi and Charles N. Fischer. Description-driven code generation using attribute grammars. In *ACM Symposium on Principles of Programming Languages*, January 1982.

[GF85] Mahadevan Ganapathi and Charles N. Fischer. Affix grammar driven code generation. *ACM Transactions on Programming Languages and Systems*, 7(4), October 1985.

[GG78] R. S. Glanville and S. L. Graham. A new method for compiler code generation. In *ACM Symposium on Principles of Programming Languages*, January 1978.

[GH82] Thomas R. Gross and John L. Hennessy. Optimizing delayed branches. In *Proceedings of IEEE MICRO-15*, October 1982.

[GH88] James R. Goodman and Wei-Chung Hsu. Code scheduling and register allocation in large basic blocks. In *International Conference on Supercomputing*, July 1988.

[GHS82] S. L. Graham, R. R. Henry, and R. S. Schulman. An experiment in table driven code generation. *ACM SIGPLAN Symposium on Compiler Construction, SIGPLAN Notices*, 17(6), June 1982.

[Gla77] Robert S. Glanville. *A Machine Independent Algorithm for Code Generation and Its Use In Retargetable Compilers.* PhD thesis, University of California, Berkeley, December 1977. Computer Science Division EECS Technical Report UCB/CSD 78-01.

[GM86] Phillip B. Gibbons and Steven S. Muchnick. Efficient instruction scheduling for a pipelined architecture. *ACM SIGPLAN Symposium on Compiler Construction, SIGPLAN Notices*, 21(7), July 1986.

[Gro83] Thomas R. Gross. *Code Optimization of Pipeline Constraints.* PhD thesis, Stanford University, December 1983. Computer Systems Laboratory Technical Report 83-255.

[HC88] Wen-mei W. Hwu and Pohua P. Chang. Exploiting parallel microprocessor microarchitectures with a compiler code generator. *International Symposium on Computer Architecture*, June 1988.

[HD89a] Robert R. Henry and Peter C. Damron. Algorithms for table-driven code generators using tree-pattern matching. Computer Science Department Technical Report 89-02-03, University of Washington, February 1989.

[HD89b] Robert R. Henry and Peter C. Damron. Performance of table-driven code generators using tree-pattern matching. Computer Science Department Technical Report 89-02-02, University of Washington, February 1989.

[Hen84] Robert R. Henry. *Graham-Glanville Code Generators.* PhD thesis, University of California, Berkeley, May 1984. Computer Science Division EECS Technical Report UCB/CSD 84/184.

[Hen87] Robert R. Henry. The CODEGEN user's manual. Computer Science Department Technical Report 87-08-04, University of Washington, August 1987.

[HG83] John L. Hennessy and Thomas R. Gross. Postpass code optimization of pipeline constraints. *ACM Transactions on Programming Languages and Systems*, 5(3), July 1983.

[HO91] C. Brian Hall and Kevin O'Brien. Performance characteristics of architectural features of the IBM RISC System/6000. *International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1991.

[HP90] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, Inc., San Mateo, California, 1990.

[Hsu87] Wei-Chung Hsu. *Register Allocation and Code Scheduling for Load/Store Architectures.* PhD thesis, University of Wisconsin-Madison, October 1987. Computer Sciences Technical Report 722.

[Int83] Intel Corp., Santa Clara, California. *Intel iAPX 86/88, 186/188 User's Manual: Programmer's Reference*, 1983.

[Int89] Intel Corp., Santa Clara, California. *i860 64-bit Microprocessor Programmer's Reference Manual*, 1989.

[Kan87] Gerry Kane. *MIPS R2000 RISC Architecture.* Prentice Hall, Englewood Cliffs, NJ, 1987.

[Kes84] Robert R. Kessler. An architectural description driven peephole optimizer. *ACM SIGPLAN Symposium on Compiler Construction, SIGPLAN Notices*, 19(6), June 1984.

[Kes86] Peter B. Kessler. Discovering machine-specific code improvements. *ACM SIGPLAN Symposium on Compiler Construction, SIGPLAN Notices*, 21(7), July 1986.

[Lam88] Monica Lam. Software pipelining: An effective technique for VLIW machines. *ACM SIGPLAN Conference on Programming Language Design and Implementation, SIGPLAN Notices*, 23(7), July 1988.

[LCH+80] B. W. Leverett, R. G. G. Cattell, S. O. Hobbs, J. M. Newcomer, A. H. Reiner, B. R. Schatz, and W. A. Wulf. An overview of the Production-Quality Compiler-Compiler project. *Computer*, 13(8), August 1980.

[LDSM80] D. Landskov, S. Davidson, B. Shriver, and P. W. Mallett. Local microcode compaction techniques. *ACM Computing Surveys*, 12(3), September 1980.

[Lev81] B. W. Leverett. *Machine Independent Register Allocation in Optimizing Compilers.* PhD thesis, Carnegie-Mellon University, 1981.

[LH86] James R. Larus and Paul N. Hilfinger. Register allocation in the SPUR Lisp compiler. *ACM SIGPLAN Symposium on Compiler Construction, SIGPLAN Notices*, 21(7), July 1986.

[LLM+87] E. Lawler, J. K. Lenstra, C. Martel, B. Simons, and L. Stockmeyer. Pipeline scheduling: A survey. Research Report RJ-5738, IBM, July 1987.

[McM72] F. H. McMahon. Fortran CPU performance analysis. Technical report, Lawrence Livermore Laboratories, 1972.

[Mot88] Motorola, Inc. *MC88100 RISC Microprocessor User's Manual*, 1988.

[OG90] R. R. Oehler and R. D. Groves. IBM RISC System/6000 processor architecture. *IBM Journal of Research and Development*, 34(1), January 1990.

[PF91] Todd A. Proebsting and Charles N. Fischer. Linear-time, optimal code scheduling for delayed-lead architectures. *ACM SIGPLAN Conference on Programming Language Design and Implementation, SIGPLAN Notices*, 26(7), July 1991.

[PS82] David A. Patterson and C. H. Sequin. A VLSI RISC. *IEEE Computer*, 15(9), September 1982.

[PS88] A. R. Pleszkun and G. S. Sohi. The performance potential of multiple functional unit processors. *International Symposium on Computer Architecture*, June 1988.

[PS90] Krishna V. Palem and Barbara B. Simons. Scheduling time-critical instructions on RISC machines. In *ACM Symposium on Principles of Programming Languages*, January 1990.

[Sta89] Richard M. Stallman. *Using and Porting GNU CC*. Free Software Foundation, Inc., Cambridge, Massachusetts, 1989.

[Sun87] Sun Microsystems, Inc., Mountain View, California. *The SPARC Architecture Manual*, 1987.

[SV89] Gurindar S. Sohi and Sriram Vajapeyam. Tradeoffs in instruction format design for horizontal architectures. *International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1989.

[Tho67] J. F. Thorlin. Code generation for PIE (parallel instruction execution) computers. In *AFIPS Conference Proceedings*, volume 30, April 1967.

[Tho70] J. E. Thorton. *Design of a Computer – The Control Data 6600*. Scott, Foresman and Co., Glenview, IL., 1970.

[Tie89] Michael D. Tiemann. The GNU instruction scheduler. Stanford University CS 343 Class Report, June 1989.

[Tom67] R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of Research and Development*, 11, January 1967.

[Tou84] Roy F. Touzeau. A Fortran compiler for the FPS-164 scientific computer. *ACM SIGPLAN Symposium on Compiler Construction, SIGPLAN Notices*, 19(6), June 1984.

[Wal86] David W. Wall. Global register allocation at link time. *ACM SIGPLAN Symposium on Compiler Construction, SIGPLAN Notices*, 21(7), July 1986. (also DEC WRL Research Report 86/3).

[War90] H. S. Warren, Jr. Instruction scheduling for the IBM RISC System/6000 processor. *IBM Journal of Research and Development*, 34(1), January 1990.

[Wei84] R. P. Weicker. Dhrystone: A synthetic systems programming benchmark. *Communications of the ACM*, 27(10), October 1984.

[WJW+75] W. Wulf, R. K. Johnsson, C. B. Weinstock, S. O. Hobbs, and C. M. Geschke. *The Design of an Optimizing Compiler*. Elsevier North-Holland, Inc., New York, 1975.

[WP87] David W. Wall and Michael L. Powell. The Mahler experience: Using an intermediate language as the machine description. *International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1987.

[WS87] Shlomo Weiss and James E. Smith. A study of scalar compilation techniques for pipelined supercomputers. *International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1987.

# Appendix A

# MIPS R2000 Machine Description

```
/****************
; Machine specification for MIPS R2000
;
****************/

#include "sinstrs.h"

declare
{
%reg r[0:31] (char|short|int|pointer);          /* general purpose registers */
%reg hi (char|short|int); lo (char|short|int);  /* int mul/div registers */
%reg f[0:15] (int|float|double);                /* float registers */
%reg bf (int);                                  /* bool signal from floating compare */

/* instruction pipeline stages
    IF: instruction fetch
    RD: decode/register fetch
    ALU: execution
    MEM: access memory
    WB: register writeback
*/
%resource IF; RD; ALU; MEM; WB;

/* floating point coprocessor pipeline
    FRD: float register decode
    FALU: float ALU
    FMEM: access memory
    FWB: check exceptions
    FFWB: float writeback
*/
%resource FRD; FALU; FMEM; FWB; FFWB;

%resource MD;                                   /* int mul/div unit */
%resource FMUL;                                 /* float mul unit */
%resource FDIV;                                 /* float div unit */

%memory m[0:2147483647];
%def const16 [−32768:32767];
%def uconst16 [0:65535] +halfreloc;
%def spconst16 [0:32767] +halfreloc;            /* special for loading relocatable */
```

```
%def const32 [−2147483648:−32769, 65536:2147483647]; /* only for glue */
%def uconst32 [65536:2147483647] +relocatable;
%label rlab [−32768:32767] +relative;
%label alab [0:100000000] +relocatable;
%label tmplab +local;
}

cwvm
{
%general (char|short|int|pointer) r;
%general (float|double) f;
%sp r[29] +down;                          /* stack pointer */
%fp r[30] +down;
%gp r[28] 32767;                          /* global data area pointer */
%allocable r[1:27,31];
%allocable f[0:15];
%hard r[0] 0;

%calleesave r[3:30]; f[2:15];
%arg (int) r[1] 1;                        /* 1st int arg in r[4] */
%arg (int) r[2] 2;
%arg (float|double) f[0] 1;
%arg (float|double) f[1] 2;

%retaddr r[31];
%result r[1] (int);
%result f[0] (float|double);
}

instr
{
%instr addi r, r, #const16
        {$1 = $2 + $3;}
        [IF; RD; ALU; MEM; WB]            (1,1,0)

%instr addi r, r, #const16
        {$1 = $3 + $2;}
        [IF; RD; ALU; MEM; WB]            (1,1,0)

/*** load signed constant ***/
%instr addi r, r[0], #const16
        {$1 = $3;}
        [IF; RD; ALU; MEM; WB]            (1,1,0)

%instr addiu r, r, #uconst16
        {$1 = $2 + $3;}
        [IF; RD; ALU; MEM; WB]            (1,1,0)

%instr addiu r, r, #uconst16
        {$1 = $3 + $2;}
        [IF; RD; ALU; MEM; WB]            (1,1,0)
```

```
%instr addu r, r, r
        {$1 = $2 + $3;}
        [IF; RD; ALU; MEM; WB]          (1,1,0)

%instr sub r, r, r
        {$1 = $2 - $3;}
        [IF; RD; ALU; MEM; WB]          (1,1,0)

/*** To cover unary minus ***/
%instr sub r, r[0], r
        {$1 = -$3;}
        [IF; RD; ALU; MEM; WB]          (1,1,0)

%instr andi r, r, #uconst16
        {$1 = $2 & $3;}
        [IF; RD; ALU; MEM; WB]          (1,1,0)

%instr andi r, r, #uconst16
        {$1 = $3 & $2;}
        [IF; RD; ALU; MEM; WB]          (1,1,0)

%instr ori r, r, #uconst16
        {$1 = $2 | $3;}
        [IF; RD; ALU; MEM; WB]          (1,1,0)

%instr ori r, r, #uconst16
        {$1 = $3 | $2;}
        [IF; RD; ALU; MEM; WB]          (1,1,0)

%instr xori r, r, #uconst16
        {$1 = $2 ^ $3;}
        [IF; RD; ALU; MEM; WB]          (1,1,0)

%instr xori r, r, #uconst16
        {$1 = $3 ^ $2;}
        [IF; RD; ALU; MEM; WB]          (1,1,0)

%instr and r, r, r
        {$1 = $2 & $3;}
        [IF; RD; ALU; MEM; WB]          (1,1,0)

%instr or r, r, r
        {$1 = $2 | $3;}
        [IF; RD; ALU; MEM; WB]          (1,1,0)

%instr xor r, r, r
        {$1 = $2 ^ $3;}
        [IF; RD; ALU; MEM; WB]          (1,1,0)
```

```
%instr nor r, r, r
        {$1 = ~($2 | $3);}
        [IF; RD; ALU; MEM; WB]          (1,1,0)

%instr lui r, #uconst16
        {$1 = $2 << 16;}
        [IF; RD; ALU; MEM; WB]          (1,1,0)

/*** load unsigned constant ***/
%instr ori r, r[0], #uconst16
        {$1 = $3;}
        [IF; RD; ALU; MEM; WB]          (1,1,0)

/*** complement ***/
%instr nor r, r, r[0]
        {$1 = ~$2;}
        [IF; RD; ALU; MEM; WB]          (1,1,0)

/*** these 2 set-reg-relational instrs won't really be used ***/
%instr slt r, r, r
        {$1 = $2 < $3;}                 /* signed compare */
        [IF; RD; ALU; MEM; WB]          (1,1,0)

%instr slti r, r, #const16
        {$1 = $2 < $3;}                 /* signed compare */
        [IF; RD; ALU; MEM; WB]          (1,1,0)

%instr sra r, r, #const16
        {$1 = $2 >>> $3;}               /* arithmetic shift */
        [IF; RD; ALU; MEM; WB]          (1,1,0)

%instr srav r, r, r
        {$1 = $2 >>> $3;}               /* arithmetic shift */
        [IF; RD; ALU; MEM; WB]          (1,1,0)

%instr sll r, r, #const16
        {$1 = $2 << $3;}
        [IF; RD; ALU; MEM; WB]          (1,1,0)

%instr sllv r, r, r                     /* logical shift left */
        {$1 = $2 << $3;}
        [IF; RD; ALU; MEM; WB]          (1,1,0)

%instr srl r, r, #const16
        {$1 = $2 >> $3;}                /* logical shift */
        [IF; RD; ALU; MEM; WB]          (1,1,0)

%instr srlv r, r, r
        {$1 = $2 >> $3;}                /* logical shift right */
        [IF; RD; ALU; MEM; WB]          (1,1,0)
```

```
/*** dummy converts between char, short, int.          ***/
%instr  dummy r, r                       ($1==char & $2==int)
         {$1 = char($2);}
         []                              (0,0,0)

%instr  dummy r, r                       ($1==char & $2==short)
         {$1 = char($2);}
         []                              (0,0,0)

%instr  dummy r, r                       ($1==short & $2==int)
         {$1 = short($2);}
         []                              (0,0,0)

%instr  dummy r, r                       ($1==int & $2==short)
         {$1 = int($2);}
         []                              (0,0,0)

%instr  dummy r, r                       ($1==int & $2==char)
         {$1 = int($2);}
         []                              (0,0,0)

%instr  dummy r, r                       ($1==short & $2==char)
         {$1 = short($2);}
         []                              (0,0,0)

%instr  mult r, r
         {lo = $1 * $2;}
         [IF; RD; 11*MD]                 (1,11,0)

%instr  div r, r
         {lo = $1 / $2;}
         [IF; RD; 34*MD]                 (1,34,0)

%instr  div r, r                         /* remainder */
         {hi = $1 % $2;}
         [IF; RD; 34*MD]                 (1,34,0)

%instr  mfhi r
         {$1 = hi;}
         [IF; RD; ALU; MEM, MD; WB, MD] (1,1,0)

%instr  mflo r
         {$1 = lo;}
         [IF; RD; ALU; MEM, MD; WB, MD] (1,1,0)


/***** int loads/stores *****/
%instr  lw r, #const16, r                ($1==int)
         {$1 = m[$3+$2];}
         [IF; RD; ALU; MEM; WB]          (1,2,1)
```

```
%instr lw r, #const16, r              ($1==int )
        {$1 = m[$2+$3];}
        [IF; RD; ALU; MEM; WB]        (1,2,1)

%instr lw r, #spconst16, r            ($1==int )
        {$1 = m[$3+$2];}
        [IF; RD; ALU; MEM; WB]        (1,2,1)

%instr sw r, #const16, r              ($1==int )
        {m[$3+$2] = $1;}
        [IF; RD; ALU; MEM; WB]        (1,1,0)

%instr sw r, #const16, r              ($1==int )
        {m[$2+$3] = $1;}
        [IF; RD; ALU; MEM; WB]        (1,1,0)

%instr sw r, #spconst16, r            ($1==int )
        {m[$3+$2] = $1;}
        [IF; RD; ALU; MEM; WB]        (1,1,0)

/***** These two patterns prevent blocks from (extr reg), etc *****/
%instr lw r, 0, r                     ($1==int )
        {$1 = m[$3];}
        [IF; RD; ALU; MEM; WB]        (1,2,1)

%instr sw r, 0, r                     ($1==int )
        {m[$3] = $1;}
        [IF; RD; ALU; MEM; WB]        (1,1,0)

/***** short loads/stores *****/
%instr lh r, #const16, r              ($1==short )
        {$1 = m[$3+$2];}
        [IF; RD; ALU; MEM; WB]        (1,2,1)

%instr lh r, #const16, r              ($1==short )
        {$1 = m[$2+$3];}
        [IF; RD; ALU; MEM; WB]        (1,2,1)

%instr lh r, #spconst16, r            ($1==short )
        {$1 = m[$3+$2];}
        [IF; RD; ALU; MEM; WB]        (1,2,1)

%instr sh r, #const16, r              ($1==short )
        {m[$3+$2] = $1;}
        [IF; RD; ALU; MEM; WB]        (1,1,0)

%instr sh r, #const16, r              ($1==short )
        {m[$2+$3] = $1;}
        [IF; RD; ALU; MEM; WB]        (1,1,0)
```

```
%instr sh r, #spconst16, r             ($1==short)
         {m[$3+$2] = $1;}
         [IF; RD; ALU; MEM; WB]          (1,1,0)

%instr lh r, 0, r                      ($1==short)
         {$1 = m[$3];}
         [IF; RD; ALU; MEM; WB]          (1,2,1)

%instr sh r, 0, r                      ($1==short)
         {m[$3] = $1;}
         [IF; RD; ALU; MEM; WB]          (1,1,0)

/***** char loads/stores *****/
%instr lb r, #const16, r               ($1==char)
         {$1 = m[$3+$2];}
         [IF; RD; ALU; MEM; WB]          (1,2,1)

%instr lb r, #const16, r               ($1==char)
         {$1 = m[$2+$3];}
         [IF; RD; ALU; MEM; WB]          (1,2,1)

%instr lb r, #spconst16, r             ($1==char)
         {$1 = m[$3+$2];}
         [IF; RD; ALU; MEM; WB]          (1,2,1)

%instr sb r, #const16, r               ($1==char)
         {m[$3+$2] = $1;}
         [IF; RD; ALU; MEM; WB]          (1,1,0)

%instr sb r, #const16, r               ($1==char)
         {m[$2+$3] = $1;}
         [IF; RD; ALU; MEM; WB]          (1,1,0)

%instr sb r, #spconst16, r             ($1==char)
         {m[$3+$2] = $1;}
         [IF; RD; ALU; MEM; WB]          (1,1,0)

%instr lb r, 0, r                      ($1==char)
         {$1 = m[$3];}
         [IF; RD; ALU; MEM; WB]          (1,2,1)
%instr sb r, 0, r                      ($1==char)
         {m[$3] = $1;}
         [IF; RD; ALU; MEM; WB]          (1,1,0)

%instr b #rlab
         {goto $1;}
         [IF; RD; ALU; MEM; WB]          (1,2,1)
         /* since 1 delayslot, latency==2 (but not really used) */
```

```
%instr bal #rlab
        {call $1;}
        [IF; RD; ALU; MEM; WB]          (1,2,1)

%instr jr r
        {goto $1;}
        [IF; RD; ALU; MEM; WB]          (1,2,1)

%instr jalr r
        {call $1;}
        [IF; RD; ALU; MEM; WB]          (1,2,1)

%instr beq r, r, #rlab
        {if $1 == $2
              goto $3;}
        [IF; RD; ALU; MEM; WB]          (1,2,1)

%instr bne r, r, #rlab
        {if $1 != $2
              goto $3;}
        [IF; RD; ALU; MEM; WB]          (1,2,1)

%instr blez r, #rlab
        {if $1 <= 0
              goto $2;}
        [IF; RD; ALU; MEM; WB]          (1,2,1)

%instr bgtz r, #rlab
        {if $1 > 0
              goto $2;}
        [IF; RD; ALU; MEM; WB]          (1,2,1)

%instr bltz r, #rlab
        {if $1 < 0
              goto $2;}
        [IF; RD; ALU; MEM; WB]          (1,2,1)

%instr bgez r, #rlab
        {if $1 >= 0
              goto $2;}
        [IF; RD; ALU; MEM; WB]          (1,2,1)

/***********************************************
 Floating point operations (single precision)
***********************************************/

%instr [s_lds] lwc1 f, #const16, r      ($1==float |int )
        {$1 = m[$3+$2];}
        [IF; RD; ALU; FMEM; FWB; FFWB] (1,2,1)
```

```
%instr lwc1 f, #const16, r               ($1==float |int )
        {$1 = m[$2+$3];}
        [IF; RD; ALU; FMEM; FWB; FFWB] (1,2,1)

%instr [s_sts] swc1 f, #const16, r       ($1==float |int )
        {m[$3+$2] = $1;}
        [IF; RD; ALU; FMEM; FWB; FFWB] (1,1,0)

%instr swc1 f, #const16, r               ($1==float |int )
        {m[$2+$3] = $1;}
        [IF; RD; ALU; FMEM; FWB; FFWB] (1,1,0)

/*****  These two patterns prevent blocks from (extr reg), etc *****/
%instr lwc1 f, 0, r                      ($1==float |int )
        {$1 = m[$3];}
        [IF; RD; ALU; FMEM; FWB; FFWB] (1,2,1)

%instr swc1 f, 0, r                      ($1==float |int )
        {m[$3] = $1;}
        [IF; RD; ALU; FMEM; FWB; FFWB] (1,1,0)

%instr mtc1 r, f                         (int )
        {$2 = move ($1);}
        [IF; RD; ALU; FMEM; FWB; FFWB] (1,2,1)

%instr mfc1 r, f                         (int )
        {$1 = move ($2);}
        [IF; RD; ALU; MEM; WB] (1,2,0)

%instr add.s f, f, f                     (float )
        {$1 = $2 + $3;}
        [IF; FRD; 2*FALU; FMEM; FWB; FFWB;] (1,2,0)

%instr sub.s f, f, f                     (float )
        {$1 = $2 − $3;}
        [IF; FRD; 2*FALU; FMEM; FWB; FFWB;] (1,2,0)

%instr mul.s f, f, f                     (float )
        {$1 = $2 * $3;}
        [IF; FRD; FALU; 2*FMUL; FALU; FMEM; FWB; FFWB;] (1,4,0)

%instr div.s f, f, f                     (float )
        {$1 = $2 / $3;}
        [IF; FRD; FALU; 8*FDIV; 2*FALU; FMEM; FWB; FFWB;] (1,11,0)

%instr neg.s f, f                        (float )
        {$1 = −$2;}
        [IF; FRD; FALU; FMEM; FWB; FFWB;] (1,1,0)
```

```
/* convert to int */
%instr  cvt.w.s f, f                    ($1==int  & $2==float)
        {$1 = int($2);}
        [IF; FRD; 2*FALU; FMEM; FWB; FFWB;] (1,2,0)

/* convert from int */
%instr  cvt.s.w f, f                    ($1==float  & $2==int)
        {$1 = float($2);}
        [IF; FRD; 3*FALU; FMEM; FWB; FFWB;] (1,3,0)

/*convert from double*/
%instr  cvt.s.d f, f                    ($1==float  & $2==double)
        {$1 = float($2);}
        [IF; FRD; 2*FALU; FMEM; FWB; FFWB;] (1,2,0)

%instr  c.eq.s bf, f, f                 ($2==float  & $3==float)
        {$1 = ($2 == $3);}
        [IF; FRD; FALU; FMEM; FWB; FFWB;] (1,2,1)

%instr  c.lt.s bf, f, f                 ($2==float  & $3==float)
        {$1 = ($2 < $3);}
        [IF; FRD; FALU; FMEM; FWB; FFWB;] (1,2,1)

%instr  c.le.s bf, f, f                 ($2==float  & $3==float)
        {$1 = ($2 <= $3);}
        [IF; FRD; FALU; FMEM; FWB; FFWB;] (1,2,1)


/************************************************
        double precision
************************************************/
/*** these load/stores are actually 2 instructions each ***/
%instr  *ldd f, #const16, r             ($1==double)
        {$1 = m[$3+$2];}
        []                              (0,0,0)

%instr  *ldd f, #const16, r             ($1==double)
        {$1 = m[$2+$3];}
        []                              (0,0,0)

%instr  *std f, #const16, r             ($1==double)
        {m[$3+$2] = $1;}
        []                              (0,0,0)

%instr  *std f, #const16, r             ($1==double)
        {m[$2+$3] = $1;}
        []                              (0,0,0)

%instr  *ldd f, 0, r                    ($1==double)
        {$1 = m[$3];}
        []                              (0,0,0)
```

```
%instr  *std f, 0, r                          ($1==double)
        {m[$3] = $1;}
        []                                    (0,0,0)

%instr  add.d f, f, f                         (double)
        {$1 = $2 + $3;}
        [IF; FRD; 2*FALU; FMEM; FWB; FFWB;]   (1,2,0)

%instr  sub.d f, f, f                         (double)
        {$1 = $2 − $3;}
        [IF; FRD; 2*FALU; FMEM; FWB; FFWB;]   (1,2,0)

%instr  mul.d f, f, f                         (double)
        {$1 = $2 * $3;}
        [IF; FRD; FALU; 3*FMUL; FALU; FMEM; FWB; FFWB;] (1,5,0)

%instr  div.d f, f, f                         (double)
        {$1 = $2 / $3;}
        [IF; FRD; FALU; 15*FDIV; 2*FALU; FMEM; FWB; FFWB;] (1,18,0)

%instr  neg.d f, f                            (double)
        {$1 = −$2;}
        [IF; FRD; FALU; FMEM; FWB; FFWB;]     (1,1,0)

/* convert to int */
%instr  cvt.w.d f, f                          ($1==int & $2==double)
        {$1 = int($2);}
        [IF; FRD; 2*FALU; FMEM; FWB; FFWB;]   (1,2,0)

/* convert from int */
%instr  cvt.d.w f, f                          ($2==int & $1==double)
        {$1 = double($2);}
        [IF; FRD; 3*FALU; FMEM; FWB; FFWB]    (1,3,0)

/*convert from float*/
%instr  cvt.d.s f, f                          ($2==float & $1==double)
        {$1 = double($2);}
        [IF; FRD; FALU; FMEM; FWB; FFWB;]     (1,1,0)

%instr  c.eq.d bf, f, f                       ($2==double & $3==double)
        {$1 = ($2 == $3);}
        [IF; FRD; FALU; FMEM; FWB; FFWB;]     (1,2,1)

%instr  c.lt.d bf, f, f                       ($2==double & $3==double)
        {$1 = ($2 < $3);}
        [IF; FRD; FALU; FMEM; FWB; FFWB;]     (1,2,1)

%instr  c.le.d bf, f, f                       ($2==double & $3==double)
        {$1 = ($2 <= $3);}
        [IF; FRD; FALU; FMEM; FWB; FFWB;]     (1,2,1)
```

```
%instr  bc1t  bf, #rlab
          {if  $1
               goto  $2;
          }
          [IF;  RD,  FRD;  ALU,  FALU;  MEM;  WB;]        (1,2,1)

%instr  bc1f  bf, #rlab
          {if  ~$1
               goto  $2;
          }
          [IF;  RD,  FRD;  ALU,  FALU;  MEM;  WB;]        (1,2,1)

%move  or  r,  r,  r[0]
          {$1  =  $2;}
          [IF;  RD;  ALU;  MEM;  WB]                       (1,1,0)

%move  movf  f,  f                                         (float|double)
          {$1  =  $2;}
          [IF;  FRD;  FALU;  FMEM;  FWB;  FFWB;]           (1,1,0)


%nop  noop                   /* really or r[0], r[0], r[0] */
          {}
          [IF;  RD;  ALU;  MEM;  WB]                       (1,1,0)

/* for loading 32-bit constant */
%glue  #const32                              (int)
          {$1  ==>  ((high($1)  <<  16)  |  low($1));}

/* for loading 32-bit address */
%glue  #uconst32                             (int)
          {$1  ==>  ((high($1)  <<  16)  +  low($1));}

/*** There is no subtract immediate ***/
%glue  r,  #const16
          {($1  -  $2)  ==>  ($1  +  eval(-$2));}
%glue  #const16,  r
          {($1  -  $2)  ==>  ($2  +  eval(-$1));}

/*** Transformations for relationals ***/
%glue  r                                     (int)
          {(0  <  $1)  ==>  ($1  >  0);}
%glue  r                                     (int)
          {(0  <=  $1)  ==>  ($1  >=  0);}
%glue  r                                     (int)
          {(0  >  $1)  ==>  ($1  <  0);}
%glue  r                                     (int)
          {(0  >=  $1)  ==>  ($1  <=  0);}
%glue  r,  r                                 (int)
          {($1  <  $2)  ==>  (($1  -  $2)  <  0);}
```

```
%glue r, r                                  (int)
        {($1 <= $2) ==> (($1 - $2) <= 0);}
%glue r, r                                  (int)
        {($1 > $2) ==> (($1 - $2) > 0);}
%glue r, r                                  (int)
        {($1 >= $2) ==> (($1 - $2) >= 0);}

%glue f, f                                  (float|double)
        {($1 != $2) ==> (~($1 == $2));}
%glue f, f                                  (float|double)
        {($1 > $2) ==> (~($1 <= $2));}
%glue f, f                                  (float|double)
        {($1 >= $2) ==> (~($1 < $2));}


/*** For converts ***/
%glue f                                     (float)
        {(int($1)) ==> (move(int($1)));}
%glue f                                     (double)
        {(int($1)) ==> (move(int($1)));}
%glue r                                     (int)
        {(float($1)) ==> (float(move($1)));}
%glue r                                     (int)
        {(double($1)) ==> (double(move($1)));}

/*** char|short conversions; int|short in replacement ensures expr is typed
     correctly, dummy instr matches it
***/
%glue r                                     ($1==char|short)
        {(int($1)) ==> ((int($1) << 16) >>> 16);}    /* >>> is arith >> */
%glue r                                     ($1==char)
        {(short($1)) ==> ((short($1) << 8) >>> 8);} /* >>> is arith >> */
}
```

# Appendix B

# MIPS R2000 Machine Description to Interface with the MIPS Assembler

This is the machine description used to produce executable code for the R2000. The code produced is assembled by the MIPS assembler and linked with the MIPS libraries. The differences from the machine description in Appendix A are as follows:

(1) The MIPS C calling conventions are followed [Kan87].

(2) The *la* pseudo-instruction is used to load a 32-bit relocatable address. In addition, other load and store instructions have 32-bit relocatable address operands. All of these map into a 2-instruction sequences.

(3) The global data pointer is not used.

(4) Delay slots are used to prevent *div* or *mul* from being scheduled immediately after *mflo* or *mfhi*.

```
/****************
; Machine specification for MIPS R2000 executable code
;          for CGP
****************/

#include "sinstrs.h"

declare
{
%reg r[0:31] (char|short|int|pointer);        /* general purpose registers */
%reg hi (char|short|int); lo (char|short|int); /* int mul/div registers */
%reg f[0:15] (int|float|double);              /* float registers */
%reg bf (int);                                /* bool signal from floating compare */

/* instruction pipeline stages
    IF: instruction fetch
    RD: decode/register fetch
    ALU: execution
    MEM: access memory
    WB: register writeback
*/
%resource IF; RD; ALU; MEM; WB;
```

```
/* floating point coprocessor pipeline
   FRD: float register decode
   FALU: float ALU
   FMEM: access memory
   FWB: check exceptions
   FFWB: float writeback
*/
%resource FRD; FALU; FMEM; FWB; FFWB;

%resource MD;                      /* int mul/div unit */
%resource FMUL;                    /* float mul unit */
%resource FDIV;                    /* float div unit */

%memory m[0:2147483647];
%def const16 [-32768:32767];
%def uconst16 [0:65535] +halfreloc;
%def spconst16 [0:32767] +halfreloc;   /* special for loading relocatable */
%def const32 [-2147483648:-32769, 65536:2147483647]; /* only for glue */
%def uconst32 [65536:2147483647] +relocatable;
%label rlab [-32768:32767] +relative;
%label alab [0:100000000] +relocatable;
%label tmplab +local;
}

cwvm
{
%general (char|short|int|pointer) r;
%general (float|double) f;
%sp r[29] +down;                   /* stack pointer */
%fp r[30] +down;
%allocable r[2:25,31];
%allocable f[0:15];
%hard r[0] 0;

%calleesave r[16:23]; f[10:15];
%arg (int) r[4] 1;                 /* 1st int arg in r[4] */
%arg (int) r[5] 2;
%arg (int) r[6] 3;
%arg (int) r[7] 4;
%arg (float|double) f[6] 1;
%arg (float|double) f[7] 2;
%retaddr r[31];
%result r[2] (int);
%result f[0] (float|double);
}

instr
{
%instr addi r, r, #const16
        {$1 = $2 + $3;}
        [IF; RD; ALU; MEM; WB]       (1,1,0)
```

```
%instr addi r, r, #const16
        {$1 = $3 + $2;}
        [IF; RD; ALU; MEM; WB]          (1,1,0)

/*** load signed constant ***/
%instr addi r, r[0], #const16
        {$1 = $3;}
        [IF; RD; ALU; MEM; WB]          (1,1,0)

%instr addiu r, r, #uconst16
        {$1 = $2 + $3;}
        [IF; RD; ALU; MEM; WB]          (1,1,0)

%instr addiu r, r, #uconst16
        {$1 = $3 + $2;}
        [IF; RD; ALU; MEM; WB]          (1,1,0)

%instr addu r, r, r
        {$1 = $2 + $3;}
        [IF; RD; ALU; MEM; WB]          (1,1,0)

%instr sub r, r, r
        {$1 = $2 - $3;}
        [IF; RD; ALU; MEM; WB]          (1,1,0)

/*** To cover unary minus ***/
%instr sub r, r[0], r
        {$1 = -$3;}
        [IF; RD; ALU; MEM; WB]          (1,1,0)

%instr andi r, r, #uconst16
        {$1 = $2 & $3;}
        [IF; RD; ALU; MEM; WB]          (1,1,0)

%instr andi r, r, #uconst16
        {$1 = $3 & $2;}
        [IF; RD; ALU; MEM; WB]          (1,1,0)

%instr ori r, r, #uconst16
        {$1 = $2 | $3;}
        [IF; RD; ALU; MEM; WB]          (1,1,0)

%instr ori r, r, #uconst16
        {$1 = $3 | $2;}
        [IF; RD; ALU; MEM; WB]          (1,1,0)

%instr xori r, r, #uconst16
        {$1 = $2 ^ $3;}
        [IF; RD; ALU; MEM; WB]          (1,1,0)
```

```
%instr xori r, r, #uconst16
        {$1 = $3 ^ $2;}
        [IF; RD; ALU; MEM; WB]          (1,1,0)

%instr and r, r, r
        {$1 = $2 & $3;}
        [IF; RD; ALU; MEM; WB]          (1,1,0)

%instr or r, r, r
        {$1 = $2 | $3;}
        [IF; RD; ALU; MEM; WB]          (1,1,0)

%instr xor r, r, r
        {$1 = $2 ^ $3;}
        [IF; RD; ALU; MEM; WB]          (1,1,0)

%instr nor r, r, r
        {$1 = ~($2 | $3);}
        [IF; RD; ALU; MEM; WB]          (1,1,0)

%instr lui r, #uconst16
        {$1 = $2 << 16;}
        [IF; RD; ALU; MEM; WB]          (1,1,0)

/*** load unsigned constant ***/
%instr ori r, r[0], #uconst16
        {$1 = $3;}
        [IF; RD; ALU; MEM; WB]          (1,1,0)

/*** complement ***/
%instr nor r, r, r[0]
        {$1 = ~$2;}
        [IF; RD; ALU; MEM; WB]          (1,1,0)

/*** these 2 set-reg-relational instrs won't really be used ***/
%instr slt r, r, r
        {$1 = $2 < $3;} /* signed compare */
        [IF; RD; ALU; MEM; WB]          (1,1,0)

%instr slti r, r, #const16
        {$1 = $2 < $3;} /* signed compare */
        [IF; RD; ALU; MEM; WB]          (1,1,0)

%instr sra r, r, #const16
        {$1 = $2 >>> $3;}               /* arithmetic shift */
        [IF; RD; ALU; MEM; WB]          (1,1,0)

%instr srav r, r, r
        {$1 = $2 >>> $3;}               /* arithmetic shift */
        [IF; RD; ALU; MEM; WB]          (1,1,0)
```

```
%instr sll r, r, #const16
        {$1 = $2 << $3;}
        [IF; RD; ALU; MEM; WB]          (1,1,0)

%instr sllv r, r, r                     /* logical shift left */
        {$1 = $2 << $3;}
        [IF; RD; ALU; MEM; WB]          (1,1,0)

%instr srl r, r, #const16
        {$1 = $2 >> $3;}                /* logical shift */
        [IF; RD; ALU; MEM; WB]          (1,1,0)

%instr srlv r, r, r
        {$1 = $2 >> $3;}                /* logical shift right */
        [IF; RD; ALU; MEM; WB]          (1,1,0)

/*** dummy converts between char, short, int.       ***/
%instr dummy r, r                       ($1==char & $2==int)
        {$1 = char($2);}
        []                              (0,0,0)

%instr dummy r, r                       ($1==char & $2==short)
        {$1 = char($2);}
        []                              (0,0,0)

%instr dummy r, r                       ($1==short & $2==int)
        {$1 = short($2);}
        []                              (0,0,0)

%instr dummy r, r                       ($1==int & $2==short)
        {$1 = int($2);}
        []                              (0,0,0)

%instr dummy r, r                       ($1==int & $2==char)
        {$1 = int($2);}
        []                              (0,0,0)

%instr dummy r, r                       ($1==short & $2==char)
        {$1 = short($2);}
        []                              (0,0,0)

%instr mult r, r
        {lo = $1 * $2;}
        [IF; RD; 11*MD]                 (1,11,0)

%instr divu r, r
        {lo = $1 / $2;}
        [IF; RD; 34*MD]                 (1,34,0)
```

```
%instr  divu r, r                          /* remainder */
        {hi = $1 % $2;}
        [IF; RD; 34*MD]                    (1,34,0)

%instr  mfhi r
        {$1 = hi;}
        [IF; RD; ALU; MEM, MD; WB, MD] (1,1,2)
        /* delay slots prevent div/mult from following directly */

%instr  mflo r
        {$1 = lo;}
        [IF; RD; ALU; MEM, MD; WB, MD] (1,1,2)

/***** int loads/stores *****/
%instr  lw r, #const16, r                  ($1==int )
        {$1 = m[$3+$2];}
        [IF; RD; ALU; MEM; WB]             (1,2,1)

%instr  lw r, #const16, r                  ($1==int )
        {$1 = m[$2+$3];}
        [IF; RD; ALU; MEM; WB]             (1,2,1)

%instr  lw r, #spconst16, r                ($1==int )
        {$1 = m[$3+$2];}
        [IF; RD; ALU; MEM; WB]             (1,2,1)

%instr  lw r, #uconst32                    ($1==int )
        {$1 = m[$2];}
        [IF; RD,IF; ALU,RD; MEM,ALU; WB,MEM; WB] (2,3,1)

%instr  sw r, #const16, r                  ($1==int )
        {m[$3+$2] = $1;}
        [IF; RD; ALU; MEM; WB]             (1,1,0)

%instr  sw r, #const16, r                  ($1==int )
        {m[$2+$3] = $1;}
        [IF; RD; ALU; MEM; WB]             (1,1,0)

%instr  sw r, #spconst16, r                ($1==int )
        {m[$3+$2] = $1;}
        [IF; RD; ALU; MEM; WB]             (1,1,0)

%instr  sw r, #uconst32                    ($1==int )
        {m[$2] = $1;}
        [IF; RD,IF; ALU,RD; MEM,ALU; WB,MEM; WB] (2,3,0)

/*****  These two patterns prevent blocks from (extr reg), etc *****/
%instr  lw r, 0, r                         ($1==int )
        {$1 = m[$3];}
        [IF; RD; ALU; MEM; WB]             (1,2,1)
```

```
%instr sw r, 0, r                          ($1==int )
        {m[$3] = $1;}
        [IF; RD; ALU; MEM; WB]             (1,1,0)

/***** short loads/stores *****/
%instr lh r, #const16, r                   ($1==short )
        {$1 = m[$3+$2];}
        [IF; RD; ALU; MEM; WB]             (1,2,1)

%instr lh r, #const16, r                   ($1==short )
        {$1 = m[$2+$3];}
        [IF; RD; ALU; MEM; WB]             (1,2,1)

%instr lh r, #spconst16, r                 ($1==short )
        {$1 = m[$3+$2];}
        [IF; RD; ALU; MEM; WB]             (1,2,1)

%instr lh r, #uconst32                     ($1==short )
        {$1 = m[$2];}
        [IF; RD,IF; ALU,RD; MEM,ALU; WB,MEM; WB] (2,3,1)

%instr sh r, #const16, r                   ($1==short )
        {m[$3+$2] = $1;}
        [IF; RD; ALU; MEM; WB]             (1,1,0)

%instr sh r, #const16, r                   ($1==short )
        {m[$2+$3] = $1;}
        [IF; RD; ALU; MEM; WB]             (1,1,0)


%instr sh r, #spconst16, r                 ($1==short )
        {m[$3+$2] = $1;}
        [IF; RD; ALU; MEM; WB]             (1,1,0)

%instr sh r, #uconst32                     ($1==short )
        {m[$2] = $1;}
        [IF; RD,IF; ALU,RD; MEM,ALU; WB,MEM; WB] (2,3,0)

%instr lh r, 0, r                          ($1==short )
        {$1 = m[$3];}
        [IF; RD; ALU; MEM; WB]             (1,2,1)

%instr sh r, 0, r                          ($1==short )
        {m[$3] = $1;}
        [IF; RD; ALU; MEM; WB]             (1,1,0)

/***** char loads/stores *****/
%instr lb r, #const16, r                   ($1==char )
        {$1 = m[$3+$2];}
        [IF; RD; ALU; MEM; WB]             (1,2,1)
```

```
%instr lb r, #const16, r              ($1==char )
        {$1 = m[$2+$3];}
        [IF; RD; ALU; MEM; WB]        (1,2,1)

%instr lb r, #spconst16, r            ($1==char )
        {$1 = m[$3+$2];}
        [IF; RD; ALU; MEM; WB]        (1,2,1)

%instr lb r, #uconst32                ($1==char )
        {$1 = m[$2];}
        [IF; RD,IF; ALU,RD; MEM,ALU; WB,MEM; WB] (2,3,1)

%instr sb r, #const16, r              ($1==char )
        {m[$3+$2] = $1;}
        [IF; RD; ALU; MEM; WB]        (1,1,0)

%instr sb r, #const16, r              ($1==char )
        {m[$2+$3] = $1;}
        [IF; RD; ALU; MEM; WB]        (1,1,0)

%instr sb r, #spconst16, r            ($1==char )
        {m[$3+$2] = $1;}
        [IF; RD; ALU; MEM; WB]        (1,1,0)

%instr sb r, #uconst32                ($1==char )
        {m[$2] = $1;}
        [IF; RD,IF; ALU,RD; MEM,ALU; WB,MEM; WB] (2,3,0)

%instr lb r, 0, r                     ($1==char )
        {$1 = m[$3];}
        [IF; RD; ALU; MEM; WB]        (1,2,1)

%instr sb r, 0, r                     ($1==char )
        {m[$3] = $1;}
        [IF; RD; ALU; MEM; WB]        (1,1,0)

%instr b #rlab
        {goto $1;}
        [IF; RD; ALU; MEM; WB]        (1,2,1)    /* ready==2, since 1 delayslot */

%instr bal #rlab
        {call $1;}
        [IF; RD; ALU; MEM; WB]        (1,2,1)

%instr j r
        {goto $1;}
        [IF; RD; ALU; MEM; WB]        (1,2,1)

%instr jal r
        {call $1;}
        [IF; RD; ALU; MEM; WB]        (1,2,1)
```

```
%instr beq r, r, #rlab
        {if $1 == $2
              goto $3;}
        [IF; RD; ALU; MEM; WB]          (1,2,1)

%instr bne r, r, #rlab
        {if $1 != $2
              goto $3;}
        [IF; RD; ALU; MEM; WB]          (1,2,1)

%instr blez r, #rlab
        {if $1 <= 0
              goto $2;}
        [IF; RD; ALU; MEM; WB]          (1,2,1)

%instr bgtz r, #rlab
        {if $1 > 0
              goto $2;}
        [IF; RD; ALU; MEM; WB]          (1,2,1)

%instr bltz r, #rlab
        {if $1 < 0
              goto $2;}
        [IF; RD; ALU; MEM; WB]          (1,2,1)

%instr bgez r, #rlab
        {if $1 >= 0
              goto $2;}
        [IF; RD; ALU; MEM; WB]          (1,2,1)


/************************************************
 Floating point operations (single precision)
************************************************/

%instr [s_lds] lwc1 f, #const16, r              ($1==float |int )
        {$1 = m[$3+$2];}
        [IF; RD; ALU; FMEM; FWB; FFWB] (1,2,1)

%instr lwc1 f, #const16, r                      ($1==float |int )
        {$1 = m[$2+$3];}
        [IF; RD; ALU; FMEM; FWB; FFWB] (1,2,1)

%instr lwc1 f, #uconst32                        ($1==float |int )
        {$1 = m[$2];}
        [IF; RD,IF; ALU,RD; FMEM,ALU; FWB,FMEM; FFWB,FWB] (2,3,1)

%instr [s_sts] swc1 f, #const16, r              ($1==float |int )
        {m[$3+$2] = $1;}
        [IF; RD; ALU; FMEM; FWB; FFWB] (1,1,0)
```

```
%instr swc1 f, #const16, r              ($1==float |int )
        {m[$2+$3] = $1;}
        [IF; RD; ALU; FMEM; FWB; FFWB] (1,1,0)

%instr swc1 f, #uconst32                ($1==float |int )
        {m[$2] = $1;}
        [IF; RD,IF; ALU,RD; FMEM,ALU; FWB,FMEM; FFWB,FWB] (2,3,1)

/*****  These two patterns prevent blocks from (extr reg), etc *****/
%instr lwc1 f, 0, r                     ($1==float |int )
        {$1 = m[$3];}
        [IF; RD; ALU; FMEM; FWB; FFWB] (1,2,1)

%instr swc1 f, 0, r                     ($1==float |int )
        {m[$3] = $1;}
        [IF; RD; ALU; FMEM; FWB; FFWB] (1,1,0)

%instr mtc1 r, f                        (int )
        {$2 = move($1);}
        [IF; RD; ALU; FMEM; FWB; FFWB] (1,2,1)

%instr mfc1 r, f                        (int )
        {$1 = move($2);}
        [IF; RD; ALU; MEM; WB]          (1,2,0)

%instr add.s f, f, f                              (float )
        {$1 = $2 + $3;}
        [IF; FRD; 2*FALU; FMEM; FWB; FFWB;]     (1,2,0)

%instr sub.s f, f, f                              (float )
        {$1 = $2 − $3;}
        [IF; FRD; 2*FALU; FMEM; FWB; FFWB;]     (1,2,0)

%instr mul.s f, f, f                              (float )
        {$1 = $2 * $3;}
        [IF; FRD; FALU; 2*FMUL; FALU; FMEM; FWB; FFWB;] (1,4,0)

%instr div.s f, f, f                              (float )
        {$1 = $2 / $3;}
        [IF; FRD; FALU; 8*FDIV; 2*FALU; FMEM; FWB; FFWB;] (1,11,0)

%instr neg.s f, f                                 (float )
        {$1 = −$2;}
        [IF; FRD; FALU; FMEM; FWB; FFWB;]       (1,1,0)

/* convert to int */
%instr cvt.w.s f, f                               ($1==int & $2==float )
        {$1 = int($2);}
        [IF; FRD; 2*FALU; FMEM; FWB; FFWB;]     (1,2,0)
```

```
/* convert from int */
%instr cvt.s.w f, f                          ($1==float & $2==int)
        {$1 = float($2);}
        [IF; FRD; 3*FALU; FMEM; FWB; FFWB;]    (1,3,0)

/*convert from double*/
%instr cvt.s.d f, f                          ($1==float & $2==double)
        {$1 = float($2);}
        [IF; FRD; 2*FALU; FMEM; FWB; FFWB;]    (1,2,0)

%instr c.eq.s bf, f, f                       ($2==float & $3==float)
        {$1 = ($2 == $3);}
        [IF; FRD; FALU; FMEM; FWB; FFWB;]      (1,2,1)

%instr c.lt.s bf, f, f                       ($2==float & $3==float)
        {$1 = ($2 < $3);}
        [IF; FRD; FALU; FMEM; FWB; FFWB;]      (1,2,1)

%instr c.le.s bf, f, f                       ($2==float & $3==float)
        {$1 = ($2 <= $3);}
        [IF; FRD; FALU; FMEM; FWB; FFWB;]      (1,2,1)

/***********************************************
        double precision
***********************************************/
/*** these load/stores are actually 2 instructions each ***/
%instr *ldd f, #const16, r              ($1==double)
        {$1 = m[$3+$2];}
        []                              (0,0,0)

%instr *ldd f, #const16, r              ($1==double)
        {$1 = m[$2+$3];}
        []                              (0,0,0)

%instr *std f, #const16, r              ($1==double)
        {m[$3+$2] = $1;}
        []                              (0,0,0)

%instr *std f, #const16, r              ($1==double)
        {m[$2+$3] = $1;}
        []                              (0,0,0)

%instr *ldd f, 0, r                     ($1==double)
        {$1 = m[$3];}
        []                              (0,0,0)

%instr *std f, 0, r                     ($1==double)
        {m[$3] = $1;}
        []                              (0,0,0)
```

```
%instr  add.d f, f, f                              (double)
        {$1 = $2 + $3;}
        [IF; FRD; 2*FALU; FMEM; FWB; FFWB;]        (1,2,0)

%instr  sub.d f, f, f                              (double)
        {$1 = $2 - $3;}
        [IF; FRD; 2*FALU; FMEM; FWB; FFWB;]        (1,2,0)

%instr  mul.d f, f, f                              (double)
        {$1 = $2 * $3;}
        [IF; FRD; FALU; 3*FMUL; FALU; FMEM; FWB; FFWB;] (1,5,0)

%instr  div.d f, f, f                              (double)
        {$1 = $2 / $3;}
        [IF; FRD; FALU; 15*FDIV; 2*FALU; FMEM; FWB; FFWB;] (1,18,0)

%instr  neg.d f, f                                 (double)
        {$1 = -$2;}
        [IF; FRD; FALU; FMEM; FWB; FFWB;]          (1,1,0)

/* convert to int */
%instr  cvt.w.d f, f                               ($1==int & $2==double)
        {$1 = int($2);}
        [IF; FRD; 2*FALU; FMEM; FWB; FFWB;]        (1,2,0)

/* convert from int */
%instr  cvt.d.w f, f                               ($2==int & $1==double)
        {$1 = double($2);}
        [IF; FRD; 3*FALU; FMEM; FWB; FFWB]         (1,3,0)

/*convert from float*/
%instr  cvt.d.s f, f                               ($2==float & $1==double)
        {$1 = double($2);}
        [IF; FRD; FALU; FMEM; FWB; FFWB;]          (1,1,0)

%instr  c.eq.d bf, f, f                            ($2==double & $3==double)
        {$1 = ($2 == $3);}
        [IF; FRD; FALU; FMEM; FWB; FFWB;]          (1,2,1)

%instr  c.lt.d bf, f, f                            ($2==double & $3==double)
        {$1 = ($2 < $3);}
        [IF; FRD; FALU; FMEM; FWB; FFWB;]          (1,2,1)

%instr  c.le.d bf, f, f                            ($2==double & $3==double)
        {$1 = ($2 <= $3);}
        [IF; FRD; FALU; FMEM; FWB; FFWB;]          (1,2,1)
```

```
%instr bc1t bf, #rlab
        {if $1
            goto $2;
        }
        [IF; RD, FRD; ALU, FALU; MEM; WB;]      (1,2,1)

%instr bc1f bf, #rlab
        {if ~$1
            goto $2;
        }
        [IF; RD, FRD; ALU, FALU; MEM; WB;]      (1,2,1)

%instr la r, #uconst32
        {$1 = $2;}
        [IF; RD,IF; ALU,RD; MEM,ALU; WB,MEM; WB] (2,2,0)


%move or r, r, r[0]
        {$1 = $2;}
        [IF; RD; ALU; MEM; WB]                  (1,1,0)

%move movf f, f                                 float|double)
        {$1 = $2;}
        [IF; FRD; FALU; FMEM; FWB; FFWB;]       (1,1,0)

%nop noop               /* really or r[0], r[0], r[0] */
        {}
        [IF; RD; ALU; MEM; WB]                  (1,1,0)

/* for loading 32-bit constant */
%glue #const32                          (int)
        {$1 ==> ((high($1) << 16) | low($1));}

/* for loading 32-bit address */
/******* NOT POSSIBLE FOR ASSEMBLER ********
%glue #uconst32                         (int)
        {$1 ==> ((high($1) << 16) + low($1));}
***********************************/

/*** There is no subtract immediate ***/
%glue r, #const16
        {($1 - $2) ==> ($1 + eval(-$2));}
%glue #const16, r
        {($1 - $2) ==> ($2 + eval(-$1));}

/*** Transformations for relationals ***/
%glue r                                 (int)
        {(0 < $1) ==> ($1 > 0);}
%glue r                                 (int)
        {(0 <= $1) ==> ($1 >= 0);}
```

```
%glue r                                     (int)
        {(0 > $1) ==> ($1 < 0);}
%glue r                                     (int)
        {(0 >= $1) ==> ($1 <= 0);}
%glue r, r                                  (int)
        {($1 < $2) ==> (($1 - $2) < 0);}
%glue r, r                                  (int)
        {($1 <= $2) ==> (($1 - $2) <= 0);}
%glue r, r                                  (int)
        {($1 > $2) ==> (($1 - $2) > 0);}
%glue r, r                                  (int)
        {($1 >= $2) ==> (($1 - $2) >= 0);}
%glue f, f                                  (float|double)
        {($1 != $2) ==> (~($1 == $2));}
%glue f, f                                  (float|double)
        {($1 > $2) ==> (~($1 <= $2));}
%glue f, f                                  (float|double)
        {($1 >= $2) ==> (~($1 < $2));}

/*** For converts ***/
%glue f                                     (float)
        {(int($1)) ==> (move(int($1)));}
%glue f                                     (double)
        {(int($1)) ==> (move(int($1)));}
%glue r                                     (int)
        {(float($1)) ==> (float(move($1)));}
%glue r                                     (int)
        {(double($1)) ==> (double(move($1)));}

/*** char|short conversions; int|short in replacement ensures expr is typed
     correctly, dummy instr matches it
***/
%glue r                                     ($1==char|short)
        {(int($1)) ==> ((int($1) << 16) >>> 16);}      /* >>> is arith >> */
%glue r                                     ($1==char)
        {(short($1)) ==> ((short($1) << 8) >>> 8);} /* >>> is arith >> */
}
```

# Appendix C

# Motorola 88000 Machine Description

```
/****************
; Machine specification for Motorola 88000
;          for CGP
;
****************/

#include "sinstrs.h"

declare
{
%reg  r[0:31]  (char|short|int|pointer|float);      /* general purpose registers */
%reg  d[0:15]  (double);                             /* these overlap the gprs */
%equiv  r[0]  d[0];

/* integer unit stages
    IF: instruction fetch
    ID: decode/register fetch (reply)
    IE: execution
    IW: register writeback
*/
%resource  IF; ID; IE; IW;

/* data unit stages:
    DE: data execute (calculates effective address)
    DA: data address (drives address bus)
    DR: data reply (reads data off bus)
*/
%resource  DE; DA; DR;

/* floating point unit (FPU) stages:
    F1: inital FPU stage (shared by add and multiply pipelines)
    FA2: add stage 2
    FA3: add stage 3
    FA4: add stage 4
    FM2: multiply stage 2
    FM3: multiply stage 3
    FM4: multiply stage 4
    FM5: multiply stage 5
    FL: final FPU stage (shared)
*/
```

```
%resource F1;
%resource FA2; FA3; FA4;
%resource FM2; FM3; FM4; FM5;
%resource FL;

%memory m[0:2147483647];

/* All immediates are unsigned.      32-bit constants take 2 instrs to
   load.   Negative constants down to -65535 can be loaded in 1 instr
   using sub, hence the breakdown between nconst16 and const32
*/
%def nconst16 [-65535:-1];
%def const32 [-2147483648:-65536, 65536:2147483647];
%def uconst16 [0:65535] +halfreloc;
%def uconst32 [65536: 2147483647] +relocatable;
%label rlab [-32768:32767] +relative;
%label alab [0:100000000] +relocatable;
%label tmplab +local;
}

cwvm
{
%general (char|short|int|pointer|float) r;
%general (double) d;
%sp r[31] +down;                          /* stack pointer */
%fp r[30] +up;
%gp r[29] 65536;                          /* global data area pointer */
%allocable r[1:28];
%hard r[0] 0;

%calleesave r[4:31];
%arg (int|float) r[2] 1;                  /* 1st integer arg in r[2] */
%arg (int|float) r[3] 2;
%arg (double) d[1] 1;
%retaddr r[1];
%result r[2] (int|float);
%result d[1] (double);
%evalargs +left;
}


instr
{
%instr add r, r, #uconst16               (char|short|int)
       {$1 = $2 + $3;}
       [IF; ID; IE; IW;]                 (1,1,0)

%instr add r, r, #uconst16               (char|short|int)
       {$1 = $3 + $2;}
       [IF; ID; IE; IW;]                 (1,1,0)
```

```
%instr  add  r, r, r                        (char|short|int)
        {$1 = $2 + $3;}
        [IF; ID; IE; IW;]                   (1,1,0)

%instr  sub  r, r[0], #uconst16             (char|short|int)
        {$1 = -$3;}
        [IF; ID; IE; IW;]                   (1,1,0)

/*** To cover unary minus ***/
%instr  sub  r, r[0], r                     (char|short|int)
        {$1 = -$3;}
        [IF; ID; IE; IW;]                   (1,1,0)

%instr  sub  r, r, #uconst16                (char|short|int)
        {$1 = $2 - $3;}
        [IF; ID; IE; IW;]                   (1,1,0)

%instr  sub  r, r, r                        (char|short|int)
        {$1 = $2 - $3;}
        [IF; ID; IE; IW;]                   (1,1,0)

%instr  mul  r, r, #uconst16                (char|short|int)
        {$1 = $2 * $3;}
        [IF; ID; F1; FM2; FM3; FL; IW;]     (1,4,0)

%instr  mul  r, r, #uconst16                (char|short|int)
        {$1 = $3 * $2;}
        [IF; ID; F1; FM2; FM3; FL; IW;]     (1,4,0)

%instr  mul  r, r, r                        (char|short|int)
        {$1 = $2 * $3;}
        [IF; ID; F1; FM2; FM3; FL; IW;]     (1,4,0)

%instr  div  r, r, #uconst16                (char|short|int)
        {$1 = $2 / $3;}
        [IF; ID; F1; 34*FA2; FA3; FA4; FL; IW;] (1,38,0)

%instr  div  r, r, r                        (char|short|int)
        {$1 = $2 / $3;}
        [IF; ID; F1; 34*FA2; FA3; FA4; FL; IW;] (1,38,0)

%instr  masku  r, r, #uconst16              (char|short|int)
        {$1 = $2 & ($3 << 16);}
        [IF; ID; IE; IW;]                   (1,1,0)

%instr  masku  r, r, #uconst16              (char|short|int)
        {$1 = ($3 << 16) & $2;}
        [IF; ID; IE; IW;]                   (1,1,0)
```

```
%instr  mask r, r, #uconst16          (char|short|int)
          {$1 = $2 & $3;}
          [IF; ID; IE; IW;]           (1,1,0)

%instr  mask r, r, #uconst16          (char|short|int)
          {$1 = $3 & $2;}
          [IF; ID; IE; IW;]           (1,1,0)

%instr  andc r, r, r                  (char|short|int)
          {$1 = $2 & ~$3;}
          [IF; ID; IE; IW;]           (1,1,0)

%instr  andc r, r, r                  (char|short|int)
          {$1 = ~$3 & $2;}
          [IF; ID; IE; IW;]           (1,1,0)

%instr  and r, r, r                   (char|short|int)
          {$1 = $2 & $3;}
          [IF; ID; IE; IW;]           (1,1,0)

%instr  or r, r, #uconst16            (char|short|int)
          {$1 = $2 | $3;}
          [IF; ID; IE; IW;]           (1,1,0)

%instr  or r, r, #uconst16            (char|short|int)
          {$1 = $3 | $2;}
          [IF; ID; IE; IW;]           (1,1,0)

%instr  orc r, r, r                   (char|short|int)
          {$1 = $2 | ~$3;}
          [IF; ID; IE; IW;]           (1,1,0)

%instr  orc r, r, r                   (char|short|int)
          {$1 = ~$3 | $2;}
          [IF; ID; IE; IW;]           (1,1,0)

%instr  oru r, r, #uconst16           (char|short|int)
          {$1 = $2 | ($3 << 16);}
          [IF; ID; IE; IW;]           (1,1,0)

%instr  oru r, r, #uconst16           (char|short|int)
          {$1 = ($3 << 16) | $2;}
          [IF; ID; IE; IW;]           (1,1,0)

%instr  or r, r, r                    (char|short|int)
          {$1 = $2 | $3;}
          [IF; ID; IE; IW;]           (1,1,0)

%instr  oru r, r[0], #uconst16        (char|short|int)
          {$1 = ($3 << 16);}          /* load constant upper */
          [IF; ID; IE; IW;]           (1,1,0)
```

```
%instr or r, r[0], #uconst16           (char|short|int)
       {$1 = $3;}                      /* load constant */
       [IF; ID; IE; IW;]               (1,1,0)

%instr orc r, r[0], r                  (char|short|int)
       {$1 = ~$3;}                     /* unary complement */
       [IF; ID; IE; IW;]               (1,1,0)

%instr xoru r, r, #uconst16            (char|short|int)
       {$1 = $2 ^ ($3 << 16);}
       [IF; ID; IE; IW;]               (1,1,0)

%instr xoru r, r, #uconst16            (char|short|int)
       {$1 = ($3 << 16) ^ $2;}
       [IF; ID; IE; IW;]               (1,1,0)

%instr xor r, r, #uconst16             (char|short|int)
       {$1 = $2 ^ $3;}
       [IF; ID; IE; IW;]               (1,1,0)

%instr xor r, r, #uconst16             (char|short|int)
       {$1 = $3 ^ $2;}
       [IF; ID; IE; IW;]               (1,1,0)

%instr xorc r, r, r                    (char|short|int)
       {$1 = $2 ^ ~$3;}
       [IF; ID; IE; IW;]               (1,1,0)

%instr xorc r, r, r                    (char|short|int)
       {$1 = ~$3 ^ $2;}
       [IF; ID; IE; IW;]               (1,1,0)

%instr xor r, r, r                     (char|short|int)
       {$1 = $2 ^ $3;}
       [IF; ID; IE; IW;]               (1,1,0)

%instr cmp r, r, #uconst16             (char|short|int)
       {$1 = $2 :: $3;}                /* compare */
       [IF; ID; IE; IW;]               (1,1,0)

%instr cmp r, r, r                     (char|short|int)
       {$1 = $2 :: $3;}                /* compare */
       [IF; ID; IE; IW;]               (1,1,0)

%instr ext r, r, 0, #uconst16          (char|short|int)
       {$1 = $2 >>> $4;}               /* arithmetic right shift */
       [IF; ID; IE; IW;]               (1,1,0)
```

```
%instr  ext r, r, r                    (char|short|int)
        {$1 = $2 >>> $3;}              /* arithmetic right shift */
                                       /* assumes $3 < 32 */
        [IF; ID; IE; IW;]              (1,1,0)

%instr  extu r, r, 0, #uconst16        (char|short|int)
        {$1 = $2 >> $4;}               /* logical right shift */
        [IF; ID; IE; IW;]              (1,1,0)

%instr  extu r, r, r                   (char|short|int)
        {$1 = $2 >> $3;}               /* logical right shift */
                                       /* assumes $3 < 32 */
        [IF; ID; IE; IW;]              (1,1,0)

%instr  mak r, r, 0, #uconst16         (char|short|int)
        {$1 = $2 << $4;}               /* logical left shift */
        [IF; ID; IE; IW;]              (1,1,0)

%instr  mak r, r, r                    (char|short|int)
        {$1 = $2 << $3;}               /* logical left shift */
                                       /* assumes $3 < 32 */
        [IF; ID; IE; IW;]              (1,1,0)

/*** converts from char|short to int ***/
%instr  ext r, r, 8, 0                 ($1==int & $2==char)
        {$1 = int($2);}
        [IF; ID; IE; IW;]              (1,1,0)

%instr  ext r, r, 16, 0                ($1==int & $2==short)
        {$1 = int($2);}
        [IF; ID; IE; IW;]              (1,1,0)

%instr  ext r, r, 8, 0                 ($1==short & $2==char)
        {$1 = short($2);}
        [IF; ID; IE; IW;]              (1,1,0)

/*** dummy converts from int to char|short ***/
%instr  dummy r, r                     ($1==char & $2==int)
        {$1 = char($2);}
        []                             (0,0,0)

%instr  dummy r, r                     ($1==char & $2==short)
        {$1 = char($2);}
        []                             (0,0,0)

%instr  dummy r, r                     ($1==short & $2==int)
        {$1 = short($2);}
        []                             (0,0,0)
```

```
/*** branches ***/
%instr  brn  #rlab
          {goto  $1;}
          [IF;  ID;  IE;]                  (1,2,1)

%instr  bsrn  #rlab
          {call  $1;}
          [IF;  ID;  IE;]                  (1,2,1)

%instr  jmpn  r                            (int )
          {goto  $1;}
          [IF;  ID;  IE;]                  (1,2,1)

%instr  jsrn  r                            (int )
          {call  $1;}
          [IF;  ID;  IE;]                  (1,2,1)

%instr  beq0n  r,  #rlab                   ($1==int )
                                           /* requires transform to use cmp instr */
          {if  $1  ==  0
                goto  $2;}
          [IF;  ID;  IE;]                  (1,2,1)

%instr  bne0n  r,  #rlab                   ($1==int )
                                           /* bcnd.n ne0, r, #rlab */
          {if  $1  !=  0
                goto  $2;}
          [IF;  ID;  IE;]                  (1,2,1)

%instr  ble0n  r,  #rlab                   ($1==int )
          {if  $1  <=  0
                goto  $2;}
          [IF;  ID;  IE;]                  (1,2,1)

%instr  bgt0n  r,  #rlab                   ($1==int )
          {if  $1  >  0
                goto  $2;}
          [IF;  ID;  IE;]                  (1,2,1)

%instr  blt0n  r,  #rlab                   ($1==int )
          {if  $1  <  0
                goto  $2;}
          [IF;  ID;  IE;]                  (1,2,1)

%instr  bge0n  r,  #rlab                   ($1==int )
          {if  $1  >=  0
                goto  $2;}
          [IF;  ID;  IE;]                  (1,2,1)
```

```
/*** loads/stores ***/
%instr ld r, r, #uconst16            ($1==int|float)
       {$1 = m[$2+$3];}
       [IF; ID; DE; DA,IW; DR;]      (1,3,0)

%instr ld r, r, #uconst16            ($1==int|float)
       {$1 = m[$3+$2];}
       [IF; ID; DE; DA,IW; DR;]      (1,3,0)

%instr ldsc r, r, r                  ($1==int|float)
       {$1 = m[$2+($3<<2)];}         /* scaled load */
       [IF; ID; DE; DA,IW; DR;]      (1,3,0)

%instr ldsc r, r, r                  ($1==int|float)
       {$1 = m[($3<<2)+$2];}         /* scaled load */
       [IF; ID; DE; DA,IW; DR;]      (1,3,0)

%instr ld r, r, r                    ($1==int|float)
       {$1 = m[$2+$3];}
       [IF; ID; DE; DA,IW; DR;]      (1,3,0)

%instr ld r, r, 0                    ($1==int|float)
       {$1 = m[$2];}
       [IF; ID; DE; DA,IW; DR;]      (1,3,0)

%instr st r, r, #uconst16            ($1==int|float)
       {m[$2+$3] = $1;}
       [IF; ID; DE; DA,IW; DR;]      (1,1,0)

%instr st r, r, #uconst16            ($1==int|float)
       {m[$3+$2] = $1;}
       [IF; ID; DE; DA,IW; DR;]      (1,1,0)

%instr stsc r, r, r                  ($1==int|float)
       {m[$2+($3<<2)] = $1;}         /* scaled store */
       [IF; ID; DE; DA,IW; DR;]      (1,1,0)

%instr stsc r, r, r                  ($1==int|float)
       {m[($3<<2)+$2] = $1;}         /* scaled store */
       [IF; ID; DE; DA,IW; DR;]      (1,1,0)

%instr st r, r, r                    ($1==int|float)
       {m[$2+$3] = $1;}
       [IF; ID; DE; DA,IW; DR;]      (1,1,0)

%instr st r, r, 0                    ($1==int|float)
       {m[$2] = $1;}
       [IF; ID; DE; DA,IW; DR;]      (1,1,0)
```

```
%instr  ld.b r, r, #uconst16              ($1==char & $2==int & $3==int)
          {$1 = m[$2+$3];}
          [IF; ID; DE; DA,IW; DR;]         (1,3,0)

%instr  ld.b r, r, #uconst16              ($1==char & $2==int & $3==int)
          {$1 = m[$3+$2];}
          [IF; ID; DE; DA,IW; DR;]         (1,3,0)

%instr  ld.b r, r, r                      ($1==char & $2==int & $3==int)
          {$1 = m[$2+$3];}
          [IF; ID; DE; DA,IW; DR;]         (1,3,0)

%instr  ld.b r, r, 0                      ($1==char & $2==int & $3==int)
          {$1 = m[$2];}
          [IF; ID; DE; DA,IW; DR;]         (1,3,0)

%instr  st.b r, r, #uconst16              ($1==char & $2==int & $3==int)
          {m[$2+$3] = $1;}
          [IF; ID; DE; DA,IW; DR;]         (1,1,0)

%instr  st.b r, r, #uconst16              ($1==char & $2==int & $3==int)
          {m[$3+$2] = $1;}
          [IF; ID; DE; DA,IW; DR;]         (1,1,0)

%instr  st.b r, r, r                      ($1==char & $2==int & $3==int)
          {m[$2+$3] = $1;}
          [IF; ID; DE; DA,IW; DR;]         (1,1,0)

%instr  st.b r, r, 0                      ($1==char & $2==int & $3==int)
          {m[$2] = $1;}
          [IF; ID; DE; DA,IW; DR;]         (1,1,0)

%instr  ld.h r, r, #uconst16              ($1==short & $2==int & $3==int)
          {$1 = m[$2+$3];}
          [IF; ID; DE; DA,IW; DR;]         (1,3,0)

%instr  ld.h r, r, #uconst16              ($1==short & $2==int & $3==int)
          {$1 = m[$3+$2];}
          [IF; ID; DE; DA,IW; DR;]         (1,3,0)

%instr  ldsc.h r, r, r                    ($1==short & $2==int & $3==int)
          {$1 = m[$2+($3<<1)];}
          [IF; ID; DE; DA,IW; DR;]         (1,3,0)

%instr  ldsc.h r, r, r                    ($1==short & $2==int & $3==int)
          {$1 = m[($3<<1)+$2];}
          [IF; ID; DE; DA,IW; DR;]         (1,3,0)

%instr  ld.h r, r, r                      ($1==short & $2==int & $3==int)
          {$1 = m[$2+$3];}
          [IF; ID; DE; DA,IW; DR;]         (1,3,0)
```

```
%instr ld.h r, r, 0                    ($1==short & $2==int & $3==int)
        {$1 = m[$2];}
        [IF; ID; DE; DA,IW; DR;]        (1,3,0)

%instr st.h r, r, #uconst16            ($1==short & $2==int & $3==int)
        {m[$2+$3] = $1;}
        [IF; ID; DE; DA,IW; DR;]        (1,1,0)

%instr st.h r, r, #uconst16            ($1==short & $2==int & $3==int)
        {m[$3+$2] = $1;}
        [IF; ID; DE; DA,IW; DR;]        (1,1,0)

%instr stsc.h r, r, r                  ($1==short & $2==int & $3==int)
        {m[$2+($3<<1)] = $1;}
        [IF; ID; DE; DA,IW; DR;]        (1,1,0)

%instr stsc.h r, r, r                  ($1==short & $2==int & $3==int)
        {m[($3<<1)+$2] = $1;}
        [IF; ID; DE; DA,IW; DR;]        (1,1,0)

%instr st.h r, r, r                    ($1==short & $2==int & $3==int)
        {m[$2+$3] = $1;}
        [IF; ID; DE; DA,IW; DR;]        (1,1,0)

%instr st.h r, r, 0                    ($1==short & $2==int & $3==int)
        {m[$2] = $1;}
        [IF; ID; DE; DA,IW; DR;]        (1,1,0)

%instr ld.d d, r, #uconst16
        {$1 = m[$2+$3];}
        [IF; ID; DE; DA,DE,IW; DR,DA,IW; DR;]        (1,4,0)

%instr ld.d d, r, #uconst16
        {$1 = m[$3+$2];}
        [IF; ID; DE; DA,DE,IW; DR,DA,IW; DR;]        (1,4,0)

%instr ldsc.d d, r, r
        {$1 = m[$2+($3<<3)];}                        /* scaled load */
        [IF; ID; DE; DA,DE,IW; DR,DA,IW; DR;]        (1,4,0)

%instr ldsc.d d, r, r
        {$1 = m[($3<<3)+$2];}                        /* scaled load */
        [IF; ID; DE; DA,DE,IW; DR,DA,IW; DR;]        (1,4,0)

%instr ld.d d, r, r
        {$1 = m[$2+$3];}
        [IF; ID; DE; DA,DE,IW; DR,DA,IW; DR;]        (1,4,0)

%instr ld.d d, r, 0
        {$1 = m[$2];}
        [IF; ID; DE; DA,DE,IW; DR,DA,IW; DR;]        (1,4,0)
```

```
%instr st.d d, r, #uconst16
        {m[$2+$3] = $1;}
        [IF; ID; DE,ID; DA,IW,DE; DR,DA,IW; DR;]          (1,1,0)

%instr st.d d, r, #uconst16
        {m[$3+$2] = $1;}
        [IF; ID; DE,ID; DA,IW,DE; DR,DA,IW; DR;]          (1,1,0)

%instr stsc.d d, r, r
        {m[$2+($3<<3)] = $1;}                              /* scaled store */
        [IF; ID; DE,ID; DA,IW,DE; DR,DA,IW; DR;]          (1,1,0)

%instr stsc.d d, r, r
        {m[($3<<3)+$2] = $1;}                              /* scaled store */
        [IF; ID; DE,ID; DA,IW,DE; DR,DA,IW; DR;]          (1,1,0)

%instr st.d d, r, r
        {m[$2+$3] = $1;}
        [IF; ID; DE,ID; DA,IW,DE; DR,DA,IW; DR;]          (1,1,0)

%instr st.d d, r, 0
        {m[$2] = $1;}
        [IF; ID; DE,ID; DA,IW,DE; DR,DA,IW; DR;]          (1,1,0)


/* Floating point operations (single precision) */
%instr fadd.s r, r, r                                (float)
        {$1 = $2 + $3;}
        [IF; ID; F1; FA2; FA3; FA4; FL; IW;]         (1,5,0)

%instr fsub.s r, r, r                                (float)
        {$1 = $2 - $3;}
        [IF; ID; F1; FA2; FA3; FA4; FL; IW;]         (1,5,0)

%instr fsub.s r, r[0], r                             (float)
        {$1 = -$3;}
        [IF; ID; F1; FA2; FA3; FA4; FL; IW;]         (1,5,0)

%instr fmul.s r, r, r                                (float)
        {$1 = $2 * $3;}
        [IF; ID; F1; FM2; FM3; FM4; FM5; FL; IW;] (1,6,0)

%instr fdiv.s r, r, r                                (float)
        {$1 = $2 / $3;}
        [IF; ID; F1; 26*FA2; FA3; FA4; FL; IW;]      (1,30,0)

%instr flt.s r, r                                    ($1==float & $2==int)
        {$1 = float($2);}
        [IF; ID; F1; FA2; FA3; FA4; FL; IW;]         (1,5,0)
```

```
%instr  fadd.sds r, d, r[0]                         ($1==float)
        {$1 = float($2);}
        [IF; ID; F1; FA2; FA3; FA4; FL; IW;]        (1,5,0)

%instr  trnc.s r, r                                 ($1==int & $2==float)
        {$1 = int($2);}
        [IF; ID; F1; FA2; FA3; FA4; FL; IW;]        (1,5,0)

%instr  fcmp.s r, r, r                      ($1==int & $2==float & $3==float)
        {$1 = $2 :: $3;}
        [IF; ID; F1; FA2; FA3; FA4; FL; IW;]        (1,5,0)


/* Floating point operations (double precision) */
%instr  fadd.d d, d, d
        {$1 = $2 + $3;}
        [IF; ID; F1,ID; F1; FA2; FA3; FA4; FL; IW,FL; IW;]          (1,6,0)

%instr  fsub.d d, d, d
        {$1 = $2 - $3;}
        [IF; ID; F1,ID; F1; FA2; FA3; FA4; FL; IW,FL; IW;]          (1,6,0)

%instr  fsub.dsd d, r[0], d                                         ($2==float)
        {$1 = -$3;}
        [IF; ID; F1,ID; F1; FA2; FA3; FA4; FL; IW,FL; IW;]          (1,6,0)

%instr  fmul.d d, d, d
        {$1 = $2 * $3;}
        [IF; ID; F1,ID; 3*F1; FM2; FM3; FM4; FM5; FL; IW,FL; IW;] (1,9,0)

%instr  fdiv.d d, d, d
        {$1 = $2 / $3;}
        [IF; ID; F1,ID; F1; 55*FA2; FA3; FA4; FL; IW,FL; IW;]       (1,60,0)

%instr  flt.d d, r                                      ($2==int)
        {$1 = double($2);}
        [IF; ID; F1; FA2; FA3; FA4; FL; IW,FL; IW;]     (1,5,0)

%instr  fadd.dss d, r, r[0]                             ($2==float)
        {$1 = double($2);}
        [IF; ID; F1; FA2; FA3; FA4; FL; IW,FL; IW;]     (1,5,0)

%instr  trnc.sd r, d                                    ($1==int)
        {$1 = int($2);}
        [IF; ID; F1,ID; F1; FA2; FA3; FA4; FL; IW;]     (1,6,0)

%instr  fcmp.d r, d, d                                  ($1==int)
        {$1 = $2 :: $3;}
        [IF; ID; F1,ID; F1; FA2; FA3; FA4; FL; IW;]     (1,6,0)
```

```
%move [s_movs] or r, r, r[0]
        {$1 = $2;}
        [IF; ID; IE; IW;]                  (1,1,0)

%move *movd d, d                           (double)
        {$1 = $2;}                 /* 2-instr sequence */
        []                                 (0,0,0)

%nop noop                          /* really or r[0], r[0], r[0] */
        {}
        [IF; ID; IE; IW;]                  (1,1,0)

/*** Auxiliary latencies; takes 1 additional cycle to get the result
     to the data bus
***/
%aux fadd.d : st.d        (1.$1==2.$1) (7)
%aux fsub.d : st.d        (1.$1==2.$1) (7)
%aux fmul.d : st.d        (1.$1==2.$1) (10)
%aux fdiv.d : st.d        (1.$1==2.$1) (61)
%aux flt.d : st.d         (1.$1==2.$1) (6)
%aux fadd.dss : st.d      (1.$1==2.$1) (6)

/* for loading 32-bit constant */
%glue #const32                                    (int)
        {$1 ==> (low($1) | (high($1) << 16));}

/* for adding negative constant */
%glue r, #nconst16                                (char|short|int|pointer)
        {($1 + $2) ==> ($1 - eval(-$2));}

/* for loading negative constant */
%glue #nconst16                                   (char|short|int|pointer)
        {$1 ==> (- eval(-$1));}

/* for loading 32-bit address */
%glue #uconst32                                   (int)
        {$1 ==> ((high($1) << 16) + low($1));}

/*** Remainder ***/
%glue r, r                                        (char|short|int)
        {($1 % $2) ==> ($1 - (($1 / $2) * $2));}

/*** Transformations for relationals ***/
%glue r, #uconst16                                (int)
        {($1 == $2) ==> (($1 :: $2) == 0);}
%glue r, #uconst16                                (int)
        {($1 != $2) ==> (($1 :: $2) != 0);}
%glue r, #uconst16                                (int)
        {($1 < $2) ==> (($1 :: $2) < 0);}
%glue r, #uconst16                                (int)
        {($1 <= $2) ==> (($1 :: $2) <= 0);}
```

```
%glue r, #uconst16                          (int)
        {($1 > $2) ==> (($1 :: $2) > 0);}
%glue r, #uconst16                          (int)
        {($1 >= $2) ==> (($1 :: $2) >= 0);}

%glue #uconst16, r                          (int)
        {($1 == $2) ==> (($2 :: $1) == 0);}
%glue #uconst16, r                          (int)
        {($1 != $2) ==> (($2 :: $1) != 0);}
%glue #uconst16, r                          (int)
        {($1 < $2) ==> (($2 :: $1) > 0);}
%glue #uconst16, r                          (int)
        {($1 <= $2) ==> (($2 :: $1) >= 0);}
%glue #uconst16, r                          (int)
        {($1 > $2) ==> (($2 :: $1) < 0);}
%glue #uconst16, r                          (int)
        {($1 >= $2) ==> (($2 :: $1) <= 0);}

%glue r, r                                  (int|float)
        {($1 == $2) ==> (($1 :: $2) == 0);}
%glue r, r                                  (int|float)
        {($1 != $2) ==> (($1 :: $2) != 0);}
%glue r, r                                  (int|float)
        {($1 < $2) ==> (($1 :: $2) < 0);}
%glue r, r                                  (int|float)
        {($1 <= $2) ==> (($1 :: $2) <= 0);}
%glue r, r                                  (int|float)
        {($1 > $2) ==> (($1 :: $2) > 0);}
%glue r, r                                  (int|float)
        {($1 >= $2) ==> (($1 :: $2) >= 0);}

%glue d, d
        {($1 == $2) ==> (($1 :: $2) == 0);}
%glue d, d
        {($1 != $2) ==> (($1 :: $2) != 0);}
%glue d, d
        {($1 < $2) ==> (($1 :: $2) < 0);}
%glue d, d
        {($1 <= $2) ==> (($1 :: $2) <= 0);}
%glue d, d
        {($1 > $2) ==> (($1 :: $2) > 0);}
%glue d, d
        {($1 >= $2) ==> (($1 :: $2) >= 0);}

}
```

# Appendix D

# Intel i860 Machine Description

| X mnemonic key | | | |
|---|---|---|---|
| Position | Meaning | Character | Meaning |
| 1 | Mul opnd 1 | r | KR (placeholder only) |
| 2 | Mul opnd 2 | t | T |
| 3 | Add opnd 1 | 1 | src1 |
| 4 | Add opnd 2 | 2 | src2 |
| 5 | Adder operation | p | add |
| 6 | rdest from | s | subtract |
| 7-8 | Special reg loads | a | from Adder |
| | | m | from Multiplier |
| | | i | KI |

```
/****************
; Machine specification for Intel i860
;           for CGP
;
*/

#include "sinstrs.h"

declare
{
%reg [r_r] r[0:31] (char |short |int |pointer);          /* general purpose registers */
%reg f[0:31] (float |int );
%reg d[0:15] (double );
%equiv f[0] d[0];

%reg fl_1 (float |double );                              /* hack: floating 1.0 */

/* instruction pipeline stages */
%resource ALU;          /* core execution */
%resource WB;           /* core register write-back */

/* float pipeline stages */
%resource A1;           /* Adder stage 1 */
%resource A2;           /* Adder stage 2 */
%resource A3;           /* Adder stage 3 */
%resource M1;           /* Multiplier stage 1 */
%resource M2;           /* Multiplier stage 2 */
```

```
%resource M3;          /* Multiplier stage 3 */
%resource FWB;         /* Float register write-back */
%resource T;           /* T latch setting hardware */
%resource FQ;          /* mystery piece of hardware shared by fst and
                               scalar float instrs (except frcp) */

%resource DBUS;        /* data bus */
%resource DCH;         /* data cache */

%memory m[0:2147000000];
%def const16 [-32768:32767];
%def uconst16 [0:65535] +halfreloc;
%def spconst16 [0:32767] +halfreloc;        /* special for loading relocatable */
%def const32 [-2147483648:-32769, 65536:2147483647]; /* only for glue */
%def uconst32 [65536:2147483647] +relocatable;
%label rlab [-32768:32767] +relative;
%label alab [-100000000:100000000] +relocatable;
%label tmplab +local;

/******************************************************
Latches, elements and classes to handle floating point pipelines
*****************************************************/

/*** Clocks separate from primary clock ***/
%clock clk_a;          /* adder clock */
%clock clk_m;          /* multiplier clock */
%clock clk_t;          /* t latch clock */
%clock clk_cc;         /* condition code changer */

/*** latches representing partial results ***/
%reg[r_a1] a1 (int|float|double; clk_a) +temporal;
%reg[r_a2] a2 (int|float|double; clk_a) +temporal;
%reg[r_a3] a3 (int|float|double; clk_a) +temporal;
%reg[r_m1] m1 (float|double; clk_m) +temporal;
%reg[r_m2] m2 (float|double; clk_m) +temporal;
%reg[r_m3] m3 (float; clk_m) +temporal;
%reg t (float|double; clk_t) +temporal;
%reg cc (int; clk_cc) +temporal;            /* codition code */

%element       pfadd.s (clk_a);             /* 1st stage */
%element       pfsub.s (clk_a);
%element       pfmul.s (clk_m);
%element       pfamov.sd (clk_a);
%element       pfgt.s (clk_a);
%element       pfle.s (clk_a);
%element       pfeq.s (clk_a);
%element       pftrunc.s (clk_a);
%element       pfadd.d (clk_a);
%element       pfsub.d (clk_a);
%element       pfmul.d (clk_m);
%element       pfamov.ds (clk_a);
```

```
%element        pfgt.d (clk_a);
%element        pfle.d (clk_a);
%element        pfeq.d (clk_a);
%element        pftrunc.d (clk_a);

/*** multiply and add/sub combinations ***/                /* Intel mnemonic */
%element        Xr21mpa.s (clk_a, clk_m);                  /* r2p1.s */
%element        Xr2tmpar.s (clk_a, clk_m);                 /* r2pt.s */
%element        Xr21apat.s (clk_a, clk_m, clk_t);          /* r2ap1.s */
%element        Xr2tapart.s (clk_a, clk_m, clk_t);         /* r2apt.s */
%element        Xi21mpa.s (clk_a, clk_m);                  /* i2p1.s */
%element        Xi2tmpai.s (clk_a, clk_m);                 /* i2pt.s */
%element        Xi21apat.s (clk_a, clk_m, clk_t);          /* i2ap1.s */
%element        Xi2tapait.s (clk_a, clk_m, clk_t);         /* 12apt.s */
%element        Xra12part.s (clk_a, clk_m, clk_t);         /* rat1p2.s */
%element        X12ampa.s (clk_a, clk_m);                  /* m12apm.s */
%element        Xra12par.s (clk_a, clk_m);                 /* ra1p2.s */
%element        X12tapat.s (clk_a, clk_m, clk_t);          /* m12ttpa.s */
%element        Xia12pat.s (clk_a, clk_m, clk_t);          /* iat1p2.s */
%element        X12tmpa.s (clk_a, clk_m);                  /* m12tpm.s */
%element        Xia12pa.s (clk_a, clk_m);                  /* ia1p2.s */
%element        X12tapa.s (clk_a, clk_m);                  /* m12tpa.s */
%element        Xr21mpm.s (clk_a, clk_m);                  /* mr2p1.s */
%element        Xr2tmpmr.s (clk_a, clk_m);                 /* mr2pt.s */
%element        Xr21mpmt.s (clk_a, clk_m, clk_t);          /* mr2mp1.s */
%element        Xr2tmpmrt.s (clk_a, clk_m, clk_t);         /* mr2mpt.s */
%element        Xi21mpm.s (clk_a, clk_m);                  /* mi2p1.s */
%element        Xi2tmpmi.s (clk_a, clk_m);                 /* mi2pt.s */
%element        Xi21mpmt.s (clk_a, clk_m, clk_t);          /* mi2mp1.s */
%element        Xi2tmpmit.s (clk_a, clk_m, clk_t);         /* mi2mpt.s */
%element        Xrm12pmt.s (clk_a, clk_m, clk_t);          /* mrmt1p2.s */
%element        X12mmpm.s (clk_a, clk_m);                  /* mm12mpm.s */
%element        Xrm12pm.s (clk_a, clk_m);                  /* mrm1p2.s */
%element        X12tmpmt.s (clk_a, clk_m, clk_t);          /* mm12ttpm.s */
%element        Xim12pmt.s (clk_a, clk_m, clk_t);          /* mimt1p2.s */
%element        X12tmpm.s (clk_a, clk_m);                  /* mm12tpm.s */
%element        Xim12pm.s (clk_a, clk_m);                  /* mim1p2.s */

%element        Xr21msa.s (clk_a, clk_m);                  /* r2s1.s */
%element        Xr2tmsar.s (clk_a, clk_m);                 /* r2st.s */
%element        Xr21asat.s (clk_a, clk_m, clk_t);          /* r2as1.s */
%element        Xr2tasart.s (clk_a, clk_m, clk_t);         /* r2ast.s */
%element        Xi21msa.s (clk_a, clk_m);                  /* i2s1.s */
%element        Xi2tmsai.s (clk_a, clk_m);                 /* i2st.s */
%element        Xi21asat.s (clk_a, clk_m, clk_t);          /* i2as1.s */
%element        Xi2tasait.s (clk_a, clk_m, clk_t);         /* 12ast.s */
%element        Xra12sart.s (clk_a, clk_m, clk_t);         /* rat1s2.s */
%element        X12amsa.s (clk_a, clk_m);                  /* m12asm.s */
%element        Xra12sar.s (clk_a, clk_m);                 /* ra1s2.s */
%element        X12tasat.s (clk_a, clk_m, clk_t);          /* m12ttsa.s */
%element        Xia12sat.s (clk_a, clk_m, clk_t);          /* iat1s2.s */
```

```
%element    X12tmsa.s (clk_a, clk_m);              /* m12tsm.s */
%element    Xia12sa.s (clk_a, clk_m);              /* ia1s2.s */
%element    X12tasa.s (clk_a, clk_m);              /* m12tsa.s */
%element    Xr21msm.s (clk_a, clk_m);              /* mr2s1.s */
%element    Xr2tmsmr.s (clk_a, clk_m);             /* mr2st.s */
%element    Xr21msmt.s (clk_a, clk_m, clk_t);      /* mr2ms1.s */
%element    Xr2tmsmrt.s (clk_a, clk_m, clk_t);     /* mr2mst.s */
%element    Xi21msm.s (clk_a, clk_m);              /* mi2s1.s */
%element    Xi2tmsmi.s (clk_a, clk_m);             /* mi2st.s */
%element    Xi21msmt.s (clk_a, clk_m, clk_t);      /* mi2ms1.s */
%element    Xi2tmsmit.s (clk_a, clk_m, clk_t);     /* mi2mst.s */
%element    Xrm12smt.s (clk_a, clk_m, clk_t);      /* mrmt1s2.s */
%element    X12mmsm.s (clk_a, clk_m);              /* mm12msm.s */
%element    Xrm12sm.s (clk_a, clk_m);              /* mrm1s2.s */
%element    X12tmsmt.s (clk_a, clk_m, clk_t);      /* mm12ttsm.s */
%element    Xim12smt.s (clk_a, clk_m, clk_t);      /* mimt1s2.s */
%element    X12tmsm.s (clk_a, clk_m);              /* mm12tsm.s */
%element    Xim12sm.s (clk_a, clk_m);              /* mim1s2.s */

%element    Xr21mpa.d (clk_a, clk_m);              /* r2p1.d */
%element    Xr2tmpar.d (clk_a, clk_m);             /* r2pt.d */
%element    Xr21apat.d (clk_a, clk_m, clk_t);      /* r2ap1.d */
%element    Xr2tapart.d (clk_a, clk_m, clk_t);     /* r2apt.d */
%element    Xi21mpa.d (clk_a, clk_m);              /* i2p1.d */
%element    Xi2tmpai.d (clk_a, clk_m);             /* i2pt.d */
%element    Xi21apat.d (clk_a, clk_m, clk_t);      /* i2ap1.d */
%element    Xi2tapait.d (clk_a, clk_m, clk_t);     /* 12apt.d */
%element    Xra12part.d (clk_a, clk_m, clk_t);     /* rat1p2.d */
%element    X12ampa.d (clk_a, clk_m);              /* m12apm.d */
%element    Xra12par.d (clk_a, clk_m);             /* ra1p2.d */
%element    X12tapat.d (clk_a, clk_m, clk_t);      /* m12ttpa.d */
%element    Xia12pat.d (clk_a, clk_m, clk_t);      /* iat1p2.d */
%element    X12tmpa.d (clk_a, clk_m);              /* m12tpm.d */
%element    Xia12pa.d (clk_a, clk_m);              /* ia1p2.d */
%element    X12tapa.d (clk_a, clk_m);              /* m12tpa.d */
%element    Xr21mpm.d (clk_a, clk_m);              /* mr2p1.d */
%element    Xr2tmpmr.d (clk_a, clk_m);             /* mr2pt.d */
%element    Xr21mpmt.d (clk_a, clk_m, clk_t);      /* mr2mp1.d */
%element    Xr2tmpmrt.d (clk_a, clk_m, clk_t);     /* mr2mpt.d */
%element    Xi21mpm.d (clk_a, clk_m);              /* mi2p1.d */
%element    Xi2tmpmi.d (clk_a, clk_m);             /* mi2pt.d */
%element    Xi21mpmt.d (clk_a, clk_m, clk_t);      /* mi2mp1.d */
%element    Xi2tmpmit.d (clk_a, clk_m, clk_t);     /* mi2mpt.d */
%element    Xrm12pmt.d (clk_a, clk_m, clk_t);      /* mrmt1p2.d */
%element    X12mmpm.d (clk_a, clk_m);              /* mm12mpm.d */
%element    Xrm12pm.d (clk_a, clk_m);              /* mrm1p2.d */
%element    X12tmpmt.d (clk_a, clk_m, clk_t);      /* mm12ttpm.d */
%element    Xim12pmt.d (clk_a, clk_m, clk_t);      /* mimt1p2.d */
%element    X12tmpm.d (clk_a, clk_m);              /* mm12tpm.d */
%element    Xim12pm.d (clk_a, clk_m);              /* mim1p2.d */
```

```
%element    Xr21msa.d (clk_a, clk_m);                 /* r2s1.d */
%element    Xr2tmsar.d (clk_a, clk_m);                /* r2st.d */
%element    Xr21asat.d (clk_a, clk_m, clk_t);         /* r2as1.d */
%element    Xr2tasart.d (clk_a, clk_m, clk_t);        /* r2ast.d */
%element    Xi21msa.d (clk_a, clk_m);                 /* i2s1.d */
%element    Xi2tmsai.d (clk_a, clk_m);                /* i2st.d */
%element    Xi21asat.d (clk_a, clk_m, clk_t);         /* i2as1.d */
%element    Xi2tasait.d (clk_a, clk_m, clk_t);        /* 12ast.d */
%element    Xra12sart.d (clk_a, clk_m, clk_t);        /* rat1s2.d */
%element    X12amsa.d (clk_a, clk_m);                 /* m12asm.d */
%element    Xra12sar.d (clk_a, clk_m);                /* ra1s2.d */
%element    X12tasat.d (clk_a, clk_m, clk_t);         /* m12ttsa.d */
%element    Xia12sat.d (clk_a, clk_m, clk_t);         /* iat1s2.d */
%element    X12tmsa.d (clk_a, clk_m);                 /* m12tsm.d */
%element    Xia12sa.d (clk_a, clk_m);                 /* ia1s2.d */
%element    X12tasa.d (clk_a, clk_m);                 /* m12tsa.d */
%element    Xr21msm.d (clk_a, clk_m);                 /* mr2s1.d */
%element    Xr2tmsmr.d (clk_a, clk_m);                /* mr2st.d */
%element    Xr21msmt.d (clk_a, clk_m, clk_t);         /* mr2ms1.d */
%element    Xr2tmsmrt.d (clk_a, clk_m, clk_t);        /* mr2mst.d */
%element    Xi21msm.d (clk_a, clk_m);                 /* mi2s1.d */
%element    Xi2tmsmi.d (clk_a, clk_m);                /* mi2st.d */
%element    Xi21msmt.d (clk_a, clk_m, clk_t);         /* mi2ms1.d */
%element    Xi2tmsmit.d (clk_a, clk_m, clk_t);        /* mi2mst.d */
%element    Xrm12smt.d (clk_a, clk_m, clk_t);         /* mrmt1s2.d */
%element    X12mmsm.d (clk_a, clk_m);                 /* mm12msm.d */
%element    Xrm12sm.d (clk_a, clk_m);                 /* mrm1s2.d */
%element    X12tmsmt.d (clk_a, clk_m, clk_t);         /* mm12ttsm.d */
%element    Xim12smt.d (clk_a, clk_m, clk_t);         /* mimt1s2.d */
%element    X12tmsm.d (clk_a, clk_m);                 /* mm12tsm.d */
%element    Xim12sm.d (clk_a, clk_m);                 /* mim1s2.d */


                                          /* Operation */
                                          /* ---------- */
%class Mr2.s {Xr2@@@@*.s};                /* m1 = kr * $2 */
%class Mi2.s {Xi2@@@@*.s};                /* m1 = ki * $2 */
%class Mra.s {Xra@@@@*.s};                /* m1 = kr * a3 */
%class Mia.s {Xia@@@@*.s};                /* m1 = ki * a3 */
%class M12.s {pfmul.s, X12@@@@*.s};       /* m1 = $1 * $2 */
%class Mrm.s {Xrm@@@@*.s};                /* m1 = kr * m3 */
%class Mim.s {Xim@@@@*.s};                /* m1 = ki * m3 */
%class A1m.s {X@@1mp@*.s};                /* a1 = $1 + m3 */
%class Atm.s {X@@tmp@*.s};                /* a1 = t + m3 */
%class A1a.s {X@@1ap@*.s};                /* a1 = $1 + a3 */
%class Ata.s {X@@tap@*.s};                /* a1 = t + a3 */
%class A12.s {pfadd.s, X@@12p@*.s};       /* a1 = $1 + $2 */
%class Aam.s {X@@amp@*.s};                /* a1 = a3 + m3 */
%class Amm.s {X@@mmp@*.s};                /* a1 = m3 + m3 */
%class S1m.s {X@@1ms@*.s};                /* a1 = $1 - m3 */
%class Stm.s {X@@tms@*.s};                /* a1 = t - m3 */
%class S1a.s {X@@1as@*.s};                /* a1 = $1 - a3 */
```

```
%class  Sta.s {X@@tas@*.s};                          /* a1 = t - a3 */
%class  S12.s {pfsub.s, X@@12s@*.s};                 /* a1 = $1 - $2 */
%class  Sam.s {X@@ams@*.s};                          /* a1 = a3 - m3 */
%class  Smm.s {X@@mms@*.s};                          /* a1 = m3 - m3 */
%class  GT12.s {pfgt.s};                             /* CC = $1 > $2 */
%class  LE12.s {pfle.s};                             /* CC = $1 <= $2 */
%class  EQ12.s {pfeq.s};                             /* CC = $1 == $2 */
%class  Krld.s {X@@@@@@r*.s};                        /* kr = $1 */
%class  Kild.s {X@@@@@@i*.s};                        /* ki = $1 */
%class  Tld.s {X@@@@@@*t.s};                         /* t = m3 */
%class  Ra.s {pfadd.@, pfsub.@, pftrunc.@, pfamov.@@,
          pfgt.@, pfle.@, pfeq.@, X@@@@@a*.@};       /* $3 = a3 */
%class  Rm.s {pfmul.@, X@@@@@m*.@};                  /* $3 = m3 */
%class  M2.s {pfmul.s, X*.s};                        /* m2 = m1 */
%class  M3.s {pfmul.s, X*.s};                        /* m3 = m2 */

                                                     /* Operation */
                                                     /* ---------- */
%class  Mr2.d {Xr2@@@@*.d};                          /* m1 = kr * $2 */
%class  Mi2.d {Xi2@@@@*.d};                          /* m1 = ki * $2 */
%class  Mra.d {Xra@@@@*.d};                          /* m1 = kr * a3 */
%class  Mia.d {Xia@@@@*.d};                          /* m1 = ki * a3 */
%class  M12.d {pfmul.d, X12@@@@*.d};                 /* m1 = $1 * $2 */
%class  Mrm.d {Xrm@@@@*.d};                          /* m1 = kr * m3 */
%class  Mim.d {Xim@@@@*.d};                          /* m1 = ki * m3 */
%class  A1m.d {X@@1mp@*.d};                          /* a1 = $1 + m3 */
%class  Atm.d {X@@tmp@*.d};                          /* a1 = t + m3 */
%class  A1a.d {X@@1ap@*.d};                          /* a1 = $1 + a3 */
%class  Ata.d {X@@tap@*.d};                          /* a1 = t + a3 */
%class  A12.d {pfadd.d, X@@12p@*.d};                 /* a1 = $1 + $2 */
%class  Aam.d {X@@amp@*.d};                          /* a1 = a3 + m3 */
%class  Amm.d {X@@mmp@*.d};                          /* a1 = m3 + m3 */
%class  S1m.d {X@@1ms@*.d};                          /* a1 = $1 - m3 */
%class  Stm.d {X@@tms@*.d};                          /* a1 = t - m3 */
%class  S1a.d {X@@1as@*.d};                          /* a1 = $1 - a3 */
%class  Sta.d {X@@tas@*.d};                          /* a1 = t - a3 */
%class  S12.d {pfsub.d, X@@12s@*.d};                 /* a1 = $1 - $2 */
%class  Sam.d {X@@ams@*.d};                          /* a1 = a3 - m3 */
%class  Smm.d {X@@mms@*.d};                          /* a1 = m3 - m3 */
%class  GT12.d {pfgt.d};                             /* CC = $1 > $2 */
%class  LE12.d {pfle.d};                             /* CC = $1 <= $2 */
%class  EQ12.d {pfeq.d};                             /* CC = $1 == $2 */
%class  Krld.d {X@@@@@@r*.d};                        /* kr = $1 */
%class  Kild.d {X@@@@@@i*.d};                        /* ki = $1 */
%class  Tld.d {X@@@@@@*t.d};                         /* t = m3 */
%class  Ra.d {pfadd.@, pfsub.@, pftrunc.@, pfamov.@@,
          pfgt.@, pfle.@, pfeq.@, X@@@@@a*.@};       /* $3 = a3 */
%class  Rm.d {pfmul.@, X@@@@@m*.@};                  /* $3 = m2 */
%class  M2.d {pfmul.d, X*.d};                        /* m2 = m1 */
%class  A2 {pfadd.@, pfsub.@, pfamov.@@, pfgt.@@, pfle.@@,
              pfeq.@, pftrunc.@, X*.@};              /* a2 = a1 */
```

```
%class  A3 {pfadd.@, pfsub.@, pfamov.@@, pfgt.@, pfle.@,
             pfeq.@, pftrunc.@, X*.@};               /* a3 = a2 */
%class  Acvtfi {pftrunc.s};                          /* a1 = int($1) */
%class  Acvtdi {pftrunc.d};                          /* a1 = int($1) */
%class  Acvtfd {pfamov.sd};                          /* a1 = double($1) */
%class  Acvtdf {pfamov.ds};                          /* a1 = float($1) */
}

cwvm
{
%general  (char|short|int|pointer) r;
%general  (float) f;
%general  (double) d;
%sp  r[2] +down;                   /* stack pointer */
%fp  r[3] +down;
%gp  r[4] 32768;                   /* global data area pointer */
%allocable  r[1,5:31]; f[2:31];
%hard  r[0] 0;
%hard  f[0] 0;
%hard  f[1] 0;
%hard  d[0] 0;

%calleesave  r[2:29]; f[2:29];
%arg  (int) r[30] 1;
%arg  (int) r[31] 2;
%arg  (float) f[30] 1;
%arg  (float) f[31] 2;
%arg  (double) d[15] 1;
%result  r[30] (int);
%result  f[30] (float);
%result  d[15] (double);

%retaddr  r[1];
%evalargs  +left;
}


instr
{
/*** integer arithmetic ***/
%instr  adds #const16, r, r            (; clk_cc)
        {$3 = $1 + $2;}
        [ALU; WB]                      (1,1,0)

%instr  adds #const16, r, r            (; clk_cc)
        {$3 = $2 + $1;}
        [ALU; WB]                      (1,1,0)

%instr  adds #spconst16, r, r          (; clk_cc)          /* array addr */
        {$3 = $2 + $1;}
        [ALU; WB]                      (1,1,0)
```

```
%instr  adds r, r, r                    (; clk_cc)
         {$3 = $1 + $2;}
         [ALU; WB]                      (1,1,0)

%instr  subs #const16, r, r             (; clk_cc)
         {$3 = $1 − $2;}
         [ALU; WB]                      (1,1,0)

%instr [s_subs] subs r, r, r            (; clk_cc)
         {$3 = $1 − $2;}
         [ALU; WB]                      (1,1,0)

%instr  *idiv r, r, r
         {$3 = $1 / $2;}
         []                             (0,0,0)

%instr  *irem r, r, r
         {$3 = $1 % $2;}
         []                             (0,0,0)

/*** To cover unary minus    and complement ***/
%instr  subs r, r[0], r                 (; clk_cc)
         {$3 = −$1;}
         [ALU; WB]                      (1,1,0)

%instr  com r, r                        (; clk_cc) /* really subs −1, r, r */
         {$2 = ~$1;}
         [ALU; WB]                      (1,1,0)

/*** logical ***/
%instr  andnoth #uconst16, r, r         (; clk_cc)
         {$3 = ~($1 << 16) & $2;}
         [ALU; WB]                      (1,1,0)

%instr  andnoth #uconst16, r, r         (; clk_cc)
         {$3 = $2 & ~($1 << 16);}
         [ALU; WB]                      (1,1,0)

%instr  andnot #uconst16, r, r          (; clk_cc)
         {$3 = ~$1 & $2;}
         [ALU; WB]                      (1,1,0)

%instr  andnot #uconst16, r, r          (; clk_cc)
         {$3 = $2 & ~$1;}
         [ALU; WB]                      (1,1,0)

%instr  andnot r, r, r                  (; clk_cc)
         {$3 = ~$1 & $2;}
         [ALU; WB]                      (1,1,0)
```

```
%instr and #uconst16, r, r              (; clk_cc)
        {$3 = $1 & $2;}
        [ALU; WB]                       (1,1,0)

%instr and #uconst16, r, r              (; clk_cc)
        {$3 = $2 & $1;}
        [ALU; WB]                       (1,1,0)

%instr andh #uconst16, r, r             (; clk_cc)
        {$3 = ($1 << 16) & $2;}
        [ALU; WB]                       (1,1,0)

%instr andh #uconst16, r, r             (; clk_cc)
        {$3 = $2 & ($1 << 16);}
        [ALU; WB]                       (1,1,0)

%instr and r, r, r                      (; clk_cc)
        {$3 = $1 & $2;}
        [ALU; WB]                       (1,1,0)

%instr or #uconst16, r, r               (; clk_cc)
        {$3 = $1 | $2;}
        [ALU; WB]                       (1,1,0)

%instr or #uconst16, r, r               (; clk_cc)
        {$3 = $2 | $1;}
        [ALU; WB]                       (1,1,0)

%instr orh #uconst16, r, r              (; clk_cc)
        {$3 = ($1 << 16) | $2;}
        [ALU; WB]                       (1,1,0)

%instr orh #uconst16, r, r              (; clk_cc)
        {$3 = $2 | ($1 << 16);}
        [ALU; WB]                       (1,1,0)

%instr or r, r, r                       (; clk_cc)
        {$3 = $1 | $2;}
        [ALU; WB]                       (1,1,0)

%instr xor #uconst16, r, r              (; clk_cc)
        {$3 = $1 ^ $2;}
        [ALU; WB]                       (1,1,0)

%instr xor #uconst16, r, r              (; clk_cc)
        {$3 = $2 ^ $1;}
        [ALU; WB]                       (1,1,0)

%instr [s_xorh] xorh #uconst16, r, r    (; clk_cc)
        {$3 = ($1 << 16) ^ $2;}
        [ALU; WB]                       (1,1,0)
```

```
%instr xorh #uconst16, r, r          (; clk_cc)
       {$3 = $2 ^ ($1 << 16);}
       [ALU; WB]                     (1,1,0)

%instr xor r, r, r                   (; clk_cc)
       {$3 = $1 ^ $2;}
       [ALU; WB]                     (1,1,0)

/*** instructions for loading constants ***/
%instr[s_orh] orh #uconst16, r[0], r (; clk_cc)
       {$3 = $1 << 16;}
       [ALU; WB]                     (1,1,0)

%instr or #uconst16, r[0], r         (; clk_cc)
       {$3 = $1;}
       [ALU; WB]                     (1,1,0)

%instr subs #const16, r[0], r        (; clk_cc)
       {$3 = $1;}
       [ALU; WB]                     (1,1,0)

/*** dummy converts between char, short, int.      ***/
%instr dummy r, r                    ($1==char & $2==int)
       {$1 = char($2);}
       []                            (0,0,0)

%instr dummy r, r                    ($1==char & $2==short)
       {$1 = char($2);}
       []                            (0,0,0)

%instr dummy r, r                    ($1==short & $2==int)
       {$1 = short($2);}
       []                            (0,0,0)

%instr dummy r, r                    ($1==int & $2==short)
       {$1 = int($2);}
       []                            (0,0,0)

%instr dummy r, r                    ($1==int & $2==char)
       {$1 = int($2);}
       []                            (0,0,0)

%instr dummy r, r                    ($1==short & $2==char)
       {$1 = short($2);}
       []                            (0,0,0)

/*** shifts ***/
%instr shl #uconst16, r, r
       {$3 = $2 << $1;}
       [ALU; WB]                     (1,1,0)
```

```
%instr shl r, r, r
        {$3 = $2 << $1;}
        [ALU; WB]                       (1,1,0)

%instr shr #uconst16, r, r
        {$3 = $2 >> $1;}                /* logical shift */
        [ALU; WB]                       (1,1,0)

%instr shr r, r, r
        {$3 = $2 >> $1;}                /* logical shift */
        [ALU; WB]                       (1,1,0)

%instr shra #uconst16, r, r
        {$3 = $2 >>> $1;}               /* arithmetic shift */
        [ALU; WB]                       (1,1,0)

%instr shra r, r, r
        {$3 = $2 >>> $1;}               /* arithmetic shift */
        [ALU; WB]                       (1,1,0)

/*** integer load/stores; loads use the WB, but it does not appear
     to conflict with arithmetic/logical uses of WB
***/
%instr ld.l #const16, r, r              ($3==int)
        {$3 = m[$2+$1];}
        [ALU; DBUS,DCH]                 (1,2,0)

%instr ld.l #spconst16, r, r            ($3==int)
        {$3 = m[$2+$1];}
        [ALU; DBUS,DCH]                 (1,2,0)

%instr ld.l #const16, r, r              ($3==int)
        {$3 = m[$1+$2];}
        [ALU; DBUS,DCH]                 (1,2,0)

%instr ld.l r, r, r                     ($3==int)
        {$3 = m[$1+$2];}
        [ALU; DBUS,DCH]                 (1,2,0)

%instr st.l r, #const16, r              ($1==int)
        {m[$3+$2] = $1;}
        [ALU; DBUS; DCH]                (1,1,0)

%instr st.l r, #spconst16, r            ($1==int)
        {m[$3+$2] = $1;}
        [ALU; DBUS; DCH]                (1,1,0)

%instr st.l r, #const16, r              ($1==int)
        {m[$2+$3] = $1;}
        [ALU; DBUS; DCH]                (1,1,0)
```

```
/*****   These two patterns prevent blocks from (extr reg), etc *****/
%instr ld.l 0, r, r                  ($3==int)
        {$3 = m[$2];}
        [ALU; DBUS,DCH]              (1,2,0)

%instr st.l r, 0, r                  ($1==int)
        {m[$3] = $1;}
        [ALU; DBUS; DCH]             (1,1,0)

/*** short load/stores ***/
%instr ld.s #const16, r, r           ($3==short)
        {$3 = m[$2+$1];}
        [ALU; DBUS,DCH]              (1,2,0)

%instr ld.s #spconst16, r, r         ($3==short)
        {$3 = m[$2+$1];}
        [ALU; DBUS,DCH]              (1,2,0)

%instr ld.s #const16, r, r           ($3==short)
        {$3 = m[$1+$2];}
        [ALU; DBUS,DCH]              (1,2,0)

%instr ld.s r, r, r                  ($3==short)
        {$3 = m[$1+$2];}
        [ALU; DBUS,DCH]              (1,2,0)

%instr st.s r, #const16, r           ($1==short)
        {m[$3+$2] = $1;}
        [ALU; DBUS; DCH]             (1,1,0)

%instr st.s r, #spconst16, r         ($1==short)
        {m[$3+$2] = $1;}
        [ALU; DBUS; DCH]             (1,1,0)

%instr st.s r, #const16, r           ($1==short)
        {m[$2+$3] = $1;}
        [ALU; DBUS; DCH]             (1,1,0)

%instr ld.s 0, r, r                  ($3==short)
        {$3 = m[$2];}
        [ALU; DBUS,DCH]              (1,2,0)

%instr st.s r, 0, r                  ($1==short)
        {m[$3] = $1;}
        [ALU; DBUS; DCH]             (1,1,0)

/*** char load/stores ***/
%instr ld.b #const16, r, r           ($3==char)
        {$3 = m[$2+$1];}
        [ALU; DBUS,DCH]              (1,2,0)
```

```
%instr ld.b #spconst16, r, r        ($3==char )
        {$3 = m[$2+$1];}
        [ALU; DBUS,DCH]             (1,2,0)

%instr ld.b #const16, r, r          ($3==char )
        {$3 = m[$1+$2];}
        [ALU; DBUS,DCH]             (1,2,0)

%instr ld.b r, r, r                 ($3==char )
        {$3 = m[$1+$2];}
        [ALU; DBUS,DCH]             (1,2,0)

%instr st.b r, #const16, r          ($1==char )
        {m[$3+$2] = $1;}
        [ALU; DBUS; DCH]            (1,1,0)

%instr st.b r, #spconst16, r        ($1==char )
        {m[$3+$2] = $1;}
        [ALU; DBUS; DCH]            (1,1,0)

%instr st.b r, #const16, r          ($1==char )
        {m[$2+$3] = $1;}
        [ALU; DBUS; DCH]            (1,1,0)

%instr ld.b 0, r, r                 ($3==char )
        {$3 = m[$2];}
        [ALU; DBUS,DCH]             (1,2,0)

%instr st.b r, 0, r                 ($1==char )
        {m[$3] = $1;}
        [ALU; DBUS; DCH]            (1,1,0)

/*****  Compares and branches (see transformations also) *****/
%instr subs #const16, r, r[0]       (; clk_cc)
        {cc = ($1 < $2);}
        [ALU; WB]                   (1,2,0)

%instr subs r, r, r[0]              (; clk_cc)
        {cc = ($1 < $2);}
        [ALU; WB]                   (1,2,0)

%instr subs #const16, r, r[0]       (; clk_cc)
        {cc = ($2 > $1);}
        [ALU; WB]                   (1,2,0)

%instr subs r, r, r[0]              (; clk_cc)
        {cc = ($2 > $1);}
        [ALU; WB]                   (1,2,0)
```

```
%instr xor #uconst16, r, r[0]          (; clk_cc)
        {cc = ($1 == $2);}
        [ALU; WB]                      (1,1,0)

%instr xor r, r, r[0]                  (; clk_cc)
        {cc = ($1 == $2);}
        [ALU; WB]                      (1,1,0)

%instr br #rlab
        {goto $1;}
        [ALU]                          (1,2,1)

%instr callr #rlab                     /* really `call` */
        {call $1;}
        [ALU; DBUS]                    (1,2,1)

%instr bri r
        {goto $1;}
        [ALU]                          (1,2,1)

%instr calli r
        {call $1;}
        [ALU; DBUS]                    (1,2,1)

%instr bc.t #rlab
        {if cc
            goto $1;}
        [ALU]                          (1,2,−1)

%instr bnc.t #rlab
        {if ~cc
            goto $1;}
        [ALU]                          (1,2,−1)

/* Floating point load/stores (single precision) */
%instr [s_fldl] fld.l #const16, r, f
        {$3 = m[$2+$1];}
        [ALU; DBUS,DCH]                (1,3,0)

%instr fld.l #spconst16, r, f
        {$3 = m[$2+$1];}
        [ALU; DBUS,DCH]                (1,3,0)

%instr fld.l #const16, r, f
        {$3 = m[$1+$2];}
        [ALU; DBUS,DCH]                (1,3,0)

%instr fld.l r, r, f
        {$3 = m[$2+$1];}
        [ALU; DBUS,DCH]                (1,3,0)
```

```
%instr fst.l f, #const16, r
         {m[$3+$2] = $1;}
         [ALU,FQ; DBUS; DCH]            (1,3,0)

%instr fst.l f, #spconst16, r
         {m[$3+$2] = $1;}
         [ALU,FQ; DBUS; DCH]            (1,1,0)

%instr fst.l f, #const16, r
         {m[$2+$3] = $1;}
         [ALU,FQ; DBUS; DCH]            (1,1,0)

%instr fst.l f, r, r
         {m[$3+$2] = $1;}
         [ALU,FQ; DBUS; DCH]            (1,1,0)

%instr fld.l 0, r, f
         {$3 = m[$2];}
         [ALU; DBUS,DCH]                (1,3,0)

%instr fst.l f, 0, r
         {m[$3] = $1;}
         [ALU,FQ; DBUS; DCH]            (1,1,0)

/* Floating point load/stores (double precision) */
%instr[s_fldd] fld.d #const16, r, d
         {$3 = m[$2+$1];}
         [ALU; DBUS,DCH]                (1,3,0)

%instr fld.d #spconst16, r, d
         {$3 = m[$2+$1];}
         [ALU; DBUS,DCH]                (1,3,0)

%instr fld.d #const16, r, d
         {$3 = m[$1+$2];}
         [ALU; DBUS,DCH]                (1,3,0)

%instr fld.d r, r, d
         {$3 = m[$2+$1];}
         [ALU; DBUS,DCH]                (1,3,0)

%instr fst.d d, #const16, r
         {m[$3+$2] = $1;}
         [ALU,FQ; DBUS; DCH]            (1,1,0)

%instr fst.d d, #spconst16, r
         {m[$3+$2] = $1;}
         [ALU,FQ; DBUS; DCH]            (1,1,0)
```

```
%instr  fst.d  d,  #const16,  r
          {m[$2+$3]  =  $1;}
          [ALU,FQ;  DBUS;  DCH]                   (1,1,0)

%instr  fst.d  d,  r,  r
          {m[$3+$2]  =  $1;}
          [ALU,FQ;  DBUS;  DCH]                   (1,1,0)

%instr  fld.d  0,  r,  d
          {$3  =  m[$2];}
          [ALU;  DBUS,DCH]                        (1,3,0)

%instr  fst.l  d,  0,  r
          {m[$3]  =  $1;}
          [ALU,FQ;  DBUS;  DCH]                   (1,1,0)

/***  transfers  between  core  and  float  units  ***/
%instr [s_ixfr]  ixfr  r,  f                      ($2==int )
          {$2  =  move($1);}
          [ALU;  DBUS]                            (1,3,0)

%instr [s_fxfr]  fxfr  f,  r                      ($1==int )
          {$2  =  move($1);}
          [A1,A2,A3,M1,M2,M3,FWB;  DBUS]          (1,2,0)

%instr  GT12.s  f,  f,  f[0]                      ($1==float  &  $2==float ;  clk_a,clk_cc)
          {cc  =  ($1  >  $2);}
          [A1]                                    (1,2,0)

%instr  LE12.s  f,  f,  f[0]                      ($1==float  &  $2==float ;  clk_a,clk_cc)
          {cc  =  ~($1  <=  $2);}
          [A1]                                    (1,2,0)

%instr  EQ12.s  f,  f,  f[0]                      ($1==float  &  $2==float ;  clk_a,clk_cc)
          {cc  =  ($1  ==  $2);}
          [A1]                                    (1,2,0)

/***  add  initiations  ***/
%instr  Atm.s                                     (float ;  clk_a,clk_m,clk_t)
          {a1  =  t  +  m3;}
          [A1]                                    (1,1,0)

%instr  Atm.s                                     (float ;  clk_a,clk_m,clk_t)
          {a1  =  m3  +  t;}
          [A1]                                    (1,1,0)

%instr  Aam.s                                     (float ;  clk_a,clk_m)
          {a1  =  a3  +  m3;}
          [A1]                                    (1,1,0)
```

```
%instr  Aam.s                              (float ; clk_a,clk_m)
        {a1 = m3 + a3;}
        [A1]                               (1,1,0)

%instr  A1m.s f                            (float ; clk_a,clk_m)
        {a1 = $1 + m3;}
        [A1]                               (1,1,0)

%instr  A1m.s f                            (float ; clk_a,clk_m)
        {a1 = m3 + $1;}
        [A1]                               (1,1,0)

%instr [s_A12s]  A12.s f, f                        (float ; clk_a)
        {a1 = $1 + $2;}
        [A1]                               (1,1,0)

/*** sub initiations ***/
%instr  Stm.s                              (float ; clk_a,clk_m,clk_t)
        {a1 = t - m3;}
        [A1]                               (1,1,0)

%instr  Sam.s                              (float ; clk_a,clk_m)
        {a1 = a3 - m3;}
        [A1]                               (1,1,0)

%instr [s_S1ms]  S1m.s f                   (float ; clk_a,clk_m)
        {a1 = $1 - m3;}
        [A1]                               (1,1,0)

%instr [s_S12s]  S12.s f, f                (float ; clk_a)
        {a1 = $1 - $2;}
        [A1]                               (1,1,0)

%instr  S1m.s f[0]                         (float ; clk_a,clk_m)
        {a1 = - m3;}
        [A1]                               (1,1,0)

%instr  S12.s f[0], f                      (float ; clk_a)
        {a1 = - $2;}
        [A1]                               (1,1,0)
/*** mul initiations ***/
%instr [s_M12s]  M12.s f, f                (float ; clk_m)
        {m1 = $1 * $2;}
        [M1]                               (1,1,0)

/*** pipeline pushers ***/
%instr [s_A2]  A2                          (; clk_a)
        {a2 = a1;}
        [A2]                               (1,1,0)
```

```
%instr [s_A3]  A3                      (; clk_a)
        {a3 = a2;}
        [A3]                           (1,1,0)

%instr [s_M2s]  M2.s                   (float ; clk_m)
        {m2 = m1;}
        [M2]                           (1,1,0)

%instr [s_M3s]  M3.s                   (float ; clk_m)
        {m3 = m2;}
        [M3]                           (1,1,0)

/*** result from pipeline ***/
%instr [s_Ras]  Ra.s f                 (; clk_a)
        {$1 = a3;}
        [FWB]                          (1,0,0)    /* most bypasses exist */

%instr [s_Rms]  Rm.s f                 (float ; clk_m)
        {$1 = m3;}
        [FWB]                          (1,0,0)    /* most bypasses exist */

%instr  Tld.s                          (float ; clk_m, clk_t)
        {t = m3;}
        [T]                            (1,1,0)

%instr  *fdivs f, f, f                 (float )
        {$3 = $1 / $2;}
        []                             (0,0,0)

%instr [s_frcps]  frcp.s f, f          (float ; clk_m)
        {$2 = fl_1 / $1;}
        [A1,A2,A3,M1,M2,M3,FWB; A1,A2,A3,M1,M2,M3,FWB;
                A1,A2,A3,M1,M2,M3,FWB; FWB]
        (1,3,0)

%instr  GT12.d d, d, d[0]              (; clk_a,clk_cc)
        {cc = ($1 > $2);}
        [A1]                           (1,2,0)

%instr  LE12.d d, d, d[0]              (; clk_a,clk_cc)
        {cc = ~($1 <= $2);}
        [A1]                           (1,2,0)

%instr  EQ12.d d, d, d[0]              (; clk_a,clk_cc)
        {cc = ($1 == $2);}
        [A1]                           (1,2,0)

/*** add initiations ***/
%instr  Atm.d                          (double ; clk_a,clk_m,clk_t)
        {a1 = t + m2;}
        [A1]                           (1,1,0)
```

```
%instr  Atm.d                       (double; clk_a,clk_m,clk_t)
        {a1 = m2 + t;}
        [A1]                        (1,1,0)

%instr  Aam.d                       (double; clk_a,clk_m)
        {a1 = a3 + m2;}
        [A1]                        (1,1,0)

%instr  Aam.d                       (double; clk_a,clk_m)
        {a1 = m2 + a3;}
        [A1]                        (1,1,0)

%instr  A1m.d d                     (double; clk_a,clk_m)
        {a1 = $1 + m2;}
        [A1]                        (1,1,0)

%instr  A1m.d d                     (double; clk_a,clk_m)
        {a1 = m2 + $1;}
        [A1]                        (1,1,0)

%instr [s_A12d]  A12.d d, d         (double; clk_a)
        {a1 = $1 + $2;}
        [A1]                        (1,1,0)

/*** sub initiations ***/
%instr  Stm.d                       (double; clk_a,clk_m,clk_t)
        {a1 = t - m2;}
        [A1]                        (1,1,0)

%instr  Sam.d                       (double; clk_a,clk_m)
        {a1 = a3 - m2;}
        [A1]                        (1,1,0)

%instr [s_S1md]  S1m.d d            (double; clk_a,clk_m)
        {a1 = $1 - m2;}
        [A1]                        (1,1,0)

%instr [s_S12d]  S12.d d, d         (double; clk_a)
        {a1 = $1 - $2;}
        [A1]                        (1,1,0)

%instr  S1m.d d[0]                  (double; clk_a,clk_m)
        {a1 = - m2;}
        [A1]                        (1,1,0)

%instr  S12.d d[0], d               (double; clk_a)
        {a1 = - $2;}
        [A1]                        (1,1,0)
```

```
/*** mul initiations ***/
%instr [s_M12d] M12.d d, d              (double ; clk_m)
        {m1 = $1 * $2;}
        [2*M1]                          (1,2,0)

/*** pipeline pushers ***/
%instr [s_M2d] M2.d                     (double ; clk_m)
        {m2 = m1;}
        [2*M2]                          (1,2,0)

/*** result from pipeline ***/
%instr [s_Rad] Ra.d d                   (double ; clk_a)
        {$1 = a3;}
        [FWB]                           (1,0,0)   /* most bypasses exist */

%instr [s_Rmd] Rm.d d                   (double ; clk_m)
        {$1 = m2;}
        [FWB]                           (1,0,0)   /* most bypasses exist */

%instr  Tld.d                           (double ; clk_m, clk_t)
        {t = m2;}
        [T]                             (1,1,0)

%instr [s_frcpd] frcp.d d, d            (; clk_m)
        {$2 = fl_1 / $1;}
        [A1,A2,A3,M1,M2,M3,FWB; A1,A2,A3,M1,M2,M3,FWB;
                A1,A2,A3,M1,M2,M3,FWB; FWB]
        (1,4,0)

%instr  *fdivd d, d, d
        {$3 = $1 / $2;}
        []                              (0,0,0)

/*** conversions and other stuff ***/
%instr [s_fmlow] fmlow f, f, f          (int ; clk_m)        /* integer multiply */
        {$3 = $1 * $2;}
        [A1,A2,A3,M1,M2,M3,FWB,FQ; A1,A2,A3,M1,M2,M3,FWB,FQ;
                A1,A2,A3,M1,M2,M3,FWB,FQ; FWB]
        (1,3,0)

/* match this, func uses Acvtfi */
%instr  *cvtfi f, r                     ($1==float )
        {$2 = int ($1);}
        []                              (0,0,0)

/* convert to int */
%instr [s_Acvtfi] Acvtfi f              ($1==float ; clk_a)
        {a1 = int ($1);}
        [A1]                            (1,1,0)
```

```
%instr  *cvtdi  d,  r
        {$2  =  int ($1);}
        []                                (0,0,0)

/*  convert  to  int  */
%instr [s_Acvtdi]  Acvtdi  d              (;  clk_a)
        {a1  =  int ($1);}
        [A1]                              (1,1,0)

%instr  *cvtid  r,  d
        {$2  =  double ($1);}
        []                                (0,0,0)

%instr  Acvtdf  d                         (;  clk_a)
/*convert  from  double*/
        {a1  =  float ($1);}
        [A1]                              (1,1,0)

/*convert  from  float*/
%instr  Acvtfd  f                         ($1==float  &  $2==double ;  clk_a)
        {a1  =  double ($1);}
        [A1]                              (1,1,0)

%move   mov  r,  r
        {$2  =  $1;}
        [ALU;  WB]                        (1,1,0)

%move [s_fmovs]  fmov.s  f,  f            (float )
        {$2  =  $1;}
        [A1,A2,A3,M1,M2,M3,FWB]           (1,1,0)

%move [s_fmovd]  fmov.d  d,  d
        {$2  =  $1;}
        [A1,A2,A3,M1,M2,M3,FWB]           (1,1,0)

%nop  noop
        {}
        [ALU;  WB]                        (1,1,0)

/***  Auxiliary  latencies  ***/
%aux  fst.l  :  Rm.s        (1.$1==2.$1)  (2)        /*  extra  clock  than  normal  */
%aux  fst.l  :  Ra.s        (1.$1==2.$1)  (2)
%aux  fst.l  :  Rm.d        (1.$1==2.$1)  (2)
%aux  fst.l  :  Ra.d        (1.$1==2.$1)  (2)
%aux  fst.d  :  Rm.s        (1.$1==2.$1)  (2)
%aux  fst.d  :  Ra.s        (1.$1==2.$1)  (2)
%aux  fst.d  :  Rm.d        (1.$1==2.$1)  (2)
%aux  fst.d  :  Ra.d        (1.$1==2.$1)  (2)
%aux  Ra.s  :  M12.s        (1.$1==2.$1)  (1)        /*  no  bypass  to  MUL  $1  */
%aux  Ra.d  :  M12.d        (1.$1==2.$1)  (1)
%aux  Rm.s  :  M12.s        (1.$1==2.$1)  (1)
```

```
%aux  Rm.d : M12.d        (1.$1==2.$1) (1)

/* for loading 32-bit constant */
%glue #const32                          (int)
        {$1 ==> ((high($1) << 16) | low($1));}

/* for loading 32-bit address */
%glue #uconst32                         (int)
        {$1 ==> ((high($1) << 16) + low($1));}

/*** Transformations for relationals ***/
%glue r, r
        {($1 <= $2) ==> ~($1 > $2);}
%glue r, r
        {($1 >= $2) ==> ~($1 < $2);}
%glue #const16, r
        {($1 >= $2) ==> ~($1 < $2);}
%glue r, #const16
        {($1 >= $2) ==> ($1 > eval($2 - 1));}
%glue #const16, r
        {($1 <= $2) ==> (eval($2 - 1) < $1);}
%glue r, #const16
        {($1 <= $2) ==> ~($1 > $2);}
%glue #const16, r
        {($1 > $2) ==> ~(eval($1 - 1) < $2);}
%glue r, #const16
        {($1 < $2) ==> ~($1 > eval($2 - 1));}
%glue r, r
        {($1 != $2) ==> ~($1 == $2);}
%glue #const16, r
        {($1 != $2) ==> ~($1 == $2);}
%glue r, #const16
        {($1 != $2) ==> ~($2 == $1);}
%glue r, #const16
        {($1 == $2) ==> ($2 == $1);}
%glue f, f                              (float)
        {($1 <= $2) ==> ~~($1 <= $2);}
%glue f, f                              (float)
        {($1 >= $2) ==> ~~($2 <= $1);}
%glue f, f                              (float)
        {($1 != $2) ==> ~($1 == $2);}
%glue f, f                              (float)
        {($1 < $2) ==> ($2 > $1);}
%glue d, d
        {($1 <= $2) ==> ~~($1 <= $2);}
%glue d, d
        {($1 >= $2) ==> ~~($2 <= $1);}
%glue d, d
        {($1 != $2) ==> ~($1 == $2);}
%glue d, d
        {($1 < $2) ==> ($2 > $1);}
```

```
/*** For converts ***/
%glue r                          (int )
        {(float ($1)) ==> (float (double ($1)));}

/*** no (a − immediate) ***/
%glue r, #const16
        {($1 − $2) ==> ($1 + eval(−$2));}

/*** integer multiply ***/
%glue r, r                       (int )
        {($1 * $2) ==> (move (move ($1) * move ($2)));}

/*** char|short conversions; int in replacement ensures expr is typed
     correctly, dummy instr matches it
***/
%glue r                          ($1==char |short )
        {(int ($1)) ==> ((int ($1) << 16) >>> 16);}       /* >>> is arith >> */
%glue r                          ($1==char )
        {(short ($1)) ==> ((short ($1) << 16) >>> 16);} /* >>> is arith >> */

}
```

# Appendix E

# Sample Output

This appendix shows the machine code produced by Marion code generators for a C code fragment for the R2000, the 88000 and the i860. The RASE code generation strategy was used for each. The code fragment is part of Livermore Loop kernel 9.

```
for (i = 1;i<=i_1;++i)                                        for
    {
    px[i*25-25] = dm28*px[i*25-13] +
        dm27*px[i*25-14] +
        dm26*px[i*25-15] +
        dm25*px[i*25-16];
    }
```

The following table gives a key to the machine code produced by Marion:

| Marion machine code key | |
|---|---|
| Item | Meaning |
| [0] | issue cycle |
| - | first instruction on a cycle |
| * | instruction is a class |
| r | integer register |
| f | floating point register (if split register file) |
| r.0 | register allocated by code selector |
| r'3 | register allocated by register allocator |
| losy() | low half of relocatable address |
| hisy() | high half of relocatable address |

## E.1 R2000 Output

```
[0]-    LABELV 13
[0]     lw r'5, 32628, gp
[1]-    addi r'4, r.0, 25
[2]-    mult r'4, r'5, [lo]
[3]-    lui r'5, hisy(px-52)
[4]-    lui r'4, hisy(px-56)
[5]-    addiu r'5, r'5, losy(px-52)
[6]-    addiu r'4, r'4, losy(px-56)
[7]-    lui r'3, hisy(px-60)
[8]-    addiu r'3, r'3, losy(px-60)
[9]-    lwc1 f'6, 84, gp
[10]-   lwc1 f'5, 80, gp
[11]-   lui r'2, hisy(px-64)
[12]-   addiu r'2, r'2, losy(px-64)
[13]-   mflo r'1, [lo]
[14]-   sll r'1, r'1, 2
[15]-   addu r'5, r'1, r'5
[16]-   lwc1 f'4, 0, r'5
[17]-   addu r'5, r'1, r'4
[18]-   lwc1 f'3, 0, r'5
[19]-   addu r'5, r'1, r'3
[20]-   mul.s f'6, f'6, f'4
[21]-   lwc1 f'4, 0, r'5
[22]-   mul.s f'5, f'5, f'3
[23]-   addu r'5, r'1, r'2
[24]-   lwc1 f'3, 76, gp
[25]-   lw r'4, 32628, gp
[26]-   mul.s f'4, f'3, f'4
[27]-   lwc1 f'3, 0, r'5
[28]-   lwc1 f'2, 72, gp
[29]-   lui r'5, hisy(px-100)
[30]-   mul.s f'3, f'2, f'3
[31]-   add.s f'6, f'6, f'5
[32]-   addiu r'5, r'5, losy(px-100)
[33]-   addu r'5, r'1, r'5
[34]-   add.s f'6, f'6, f'4
[35]-   addi r'4, r'4, 1
[36]-   add.s f'6, f'6, f'3
[37]-   sw r'4, 32628, gp
[38]-   swc1 f'6, 0, r'5
```

## E.2 88000 Output

```
[0]-    LABELV 13
[0]     ld r'6, gp, 65432
[1]-    oru r'5, r.0, hisy(px-52)
[2]-    oru r'4, r.0, hisy(px-56)
[3]-    mul r'6, r'6, 25
[4]-    add r'5, r'5, losy(px-52)
[5]-    add r'4, r'4, losy(px-56)
[7]-    mak r'6, r'6, 0, 2
[8]-    ld r'5, r'6, r'5
[9]-    ld r'4, r'6, r'4
[10]-   ld r'3, gp, 84
[11]-   ld r'2, gp, 80
[12]-   oru r'1, r.0, hisy(px-60)
[13]-   fmul.s r'5, r'3, r'5
[14]-   fmul.s r'4, r'2, r'4
[15]-   add r'3, r'1, losy(px-60)
[16]-   ld r'3, r'6, r'3
[17]-   ld r'2, gp, 76
[20]-   fmul.s r'3, r'2, r'3
[21]-   oru r'2, r.0, hisy(px-64)
[22]-   fadd.s r'5, r'5, r'4
[23]-   add r'4, r'2, losy(px-64)
[24]-   ld r'4, r'6, r'4
[27]-   ld r'2, gp, 72
[28]-   fadd.s r'5, r'5, r'3
[29]-   ld r'3, gp, 65432
[30]-   fmul.s r'4, r'2, r'4
[31]-   oru r'2, r.0, hisy(px-100)
[33]-   add r'2, r'2, losy(px-100)
[34]-   add r'3, r'3, 1
[36]-   fadd.s r'5, r'5, r'4
[37]-   st r'3, gp, 65432
[41]-   st r'5, r'6, r'2
```

## E.3 i860 Output

```
[0]-     LABELV 13
[0]      ld.l 32628, gp, r'8
[1]-     or 25, r.0, r'7
[2]-     ixfr r'7, f'7
[3]-     ixfr r'8, f'6
[4]-     orh hisy(px-52), r.0, r'8
[5]-     adds losy(px-52), r'8, r'8
[6]-     fmlow f'7, f'6, f'7
[6]      fld.l 84, gp, f'6
[7]-     orh hisy(px-56), r.0, r'7
[8]-     orh hisy(px-60), r.0, r'6
[9]-     orh hisy(px-64), r.0, r'5
[10]-    fxfr f'7, r'30
[10]     adds losy(px-64), r'5, r'5
[11]-    adds losy(px-60), r'6, r'6
[12]-    shl 2, r'30, r'30
[13]-    fld.l r'8, r'30, f'7
[14]-    adds losy(px-56), r'7, r'8
[15]-    fld.l r'8, r'30, f'5
[16]-*   M12.s f'6, f'7, [m1]          "pfmul.s f'6, f'7, f'0"
[16]     fld.l 80, gp, f'7
[17]-*   M2.s [m2], [m1]               "pfmul.s f'0, f'0, f'0"
[17]     fld.l r'6, r'30, f'6
[18]-*   M3.s [m3], [m2]               "pfmul.s f'0, f'0, f'0"
[18]     fld.l 76, gp, f'4
[19]-*   Tld.s [t], [m3]               "Xr21apat.s f'0, f'0, f'0"
[19]     fld.l r'5, r'30, f'3
[20]-*   M12.s f'7, f'5, [m1]          "pfmul.s f'7, f'5, f'0"
[20]     fld.l 72, gp, f'7
[21]-*   M2.s [m2], [m1]               "pfmul.s f'0, f'0, f'0"
[21]     ld.l 32628, gp, r'8
[22]-*   M3.s [m3], [m2]               "pfmul.s f'0, f'0, f'0"
[22]     orh hisy(px-100), r.0, r'7
[23]-*   Atm.s [a1], [t], [m3]         "Xr2tmpar.s f'0, f'0, f'0"
[23]     adds losy(px-100), r'7, r'7
[24]-*   M12.s f'4, f'6, [m1]          "X12ampa.s f'4, f'6, f'0"
[24] *   A2 [a2], [a1]
[24]     adds 1, r'8, r'8
[25]-*   M2.s [m2], [m1]               "Xr21mpa.s f'0, f'0, f'0"
[25] *   A3 [a3], [a2]
[25]     st.l r'8, 32628, gp
[26]-*   M3.s [m3], [m2]               "pfmul.s f'0, f'0, f'0"
[27]-*   Aam.s [a1], [a3], [m3]        "X12ampa.s f'0, f'0, f'0"
[28]-*   M12.s f'7, f'3, [m1]          "X12ampa.s f'7, f'3, f'0"
[28] *   A2 [a2], [a1]
```

```
[29]-*   M2.s [m2], [m1]          "Xr21mpa.s f'0, f'0, f'0"
[29] *   A3 [a3], [a2]
[30]-*   M3.s [m3], [m2]          "pfmul.s f'0, f'0, f'0"
[31]-*   Aam.s [a1], [a3], [m3]   "X12ampa.s f'0, f'0, f'0"
[32]-*   A2 [a2], [a1]            "pfadd.s f'0, f'0, f'0"
[33]-*   A3 [a3], [a2]            "pfadd.s f'0, f'0, f'0"
[34]-*   Ra.s f'7, [a3]           "pfadd.s f'0, f'0, f'7"
[34]     fst.l f'7, r'7, r'30
```

# Appendix F

# Raw Data

This appendix presents the raw data used in the comparison of code generation strategies and in the architecture investigations. All data are CPU execution times calculated by Marion. The first three tables show the data used in the strategies comparison, organized by architecture. The rest of the tables show the data used in the architecture investigations. Each contains the data for one architecture, one code generation strategy and all register set sizes.

Table F.1: **R2000 raw data** for three code generation strategies.

| Program | Code Generation Strategy PP | IPS | RASE |
|---|---|---|---|
| Livermore | | | |
| Ker01 | 5.07e+07 | 5.07e+07 | 5.07e+07 |
| Ker02 | 6.38e+07 | 6.00e+07 | 6.00e+07 |
| Ker03 | 4.52e+07 | 4.52e+07 | 4.52e+07 |
| Ker04 | 4.39e+07 | 4.37e+07 | 4.37e+07 |
| Ker05 | 5.81e+07 | 5.81e+07 | 5.81e+07 |
| Ker06 | 3.87e+07 | 3.87e+07 | 3.87e+07 |
| Ker07 | 7.79e+07 | 5.89e+07 | 5.72e+07 |
| Ker08 | 1.05e+08 | 9.02e+07 | 8.98e+07 |
| Ker09 | 7.51e+07 | 6.43e+07 | 6.51e+07 |
| Ker10 | 1.09e+08 | 9.31e+07 | 9.31e+07 |
| Ker11 | 5.46e+07 | 5.46e+07 | 5.46e+07 |
| Ker12 | 5.80e+07 | 5.80e+07 | 5.80e+07 |
| Ker13 | 8.09e+07 | 7.45e+07 | 7.45e+07 |
| Ker14 | 9.00e+07 | 8.32e+07 | 8.32e+07 |
| Amean | 6.79e+07 | 6.24e+07 | 6.23e+07 |
| Nasker | | | |
| BTR | 1.80e+09 | 1.48e+09 | 1.47e+09 |
| CHO | 2.93e+09 | 2.83e+09 | 2.83e+09 |
| EMI | 6.22e+08 | 6.09e+08 | 6.04e+08 |
| FFT | 1.40e+09 | 1.35e+09 | 1.36e+09 |
| GMT | 1.78e+09 | 1.78e+09 | 1.78e+09 |
| MXM | 1.63e+09 | 1.54e+09 | 1.54e+09 |
| VPE | 5.33e+07 | 4.95e+07 | 4.95e+07 |
| Amean | 1.46e+09 | 1.38e+09 | 1.38e+09 |
| Optimized Nasker | | | |
| BTR | 1.22e+09 | 1.03e+09 | 1.01e+09 |
| CHO | 9.43e+08 | 8.76e+08 | 8.74e+08 |
| EMI | 6.26e+08 | 5.73e+08 | 5.69e+08 |
| FFT | 1.14e+09 | 1.13e+09 | 1.13e+09 |
| GMT | 1.03e+09 | 1.03e+09 | 1.03e+09 |
| MXM | 1.11e+09 | 9.35e+08 | 9.20e+08 |
| VPE | 5.33e+07 | 4.95e+07 | 4.95e+07 |
| Amean | 8.75e+08 | 8.03e+08 | 7.98e+08 |

| Program | Code Generation Strategy PP | IPS | RASE |
|---|---|---|---|
| Perfect | | | |
| ARC2D | 3.15e+10 | 2.98e+10 | 2.97e+10 |
| BDNA | 5.59e+09 | 5.03e+09 | 4.88e+09 |
| DYFESM | 6.66e+09 | 6.47e+09 | 6.47e+09 |
| FLO52 | 1.08e+10 | 1.03e+10 | 1.04e+10 |
| MDG | 1.26e+10 | 1.24e+10 | 1.24e+10 |
| MG3D | 1.29e+11 | 1.23e+11 | 1.22e+11 |
| QCD | 3.11e+09 | 3.00e+09 | 3.05e+09 |
| TRACK | 9.83e+08 | 9.62e+08 | 9.34e+08 |
| Amean | 2.50e+10 | 2.39e+10 | 2.37e+10 |
| Misc | | | |
| BOAST | 5.57e+09 | 5.05e+09 | 5.05e+09 |
| SPHOT | 8.65e+07 | 7.85e+07 | 8.00e+07 |
| WANAL1 | 1.18e+10 | 1.09e+10 | 1.09e+10 |
| NEURAL | 1.19e+09 | 1.04e+09 | 1.04e+09 |
| COSTSC | 1.19e+08 | 8.64e+07 | 8.68e+07 |
| Amean | 3.75e+09 | 3.43e+09 | 3.43e+09 |
| Int | | | |
| DHRYSTO | 1.42e+07 | 1.27e+07 | 1.35e+07 |
| INTEGER | 1.89e+08 | 1.81e+08 | 1.81e+08 |
| KNIGHT | 4.57e+06 | 3.90e+06 | 3.94e+06 |
| LCC | 2.03e+07 | 1.46e+07 | 1.56e+07 |
| SPARSE | 3.31e+06 | 2.67e+06 | 2.78e+06 |
| Amean | 4.63e+07 | 4.30e+07 | 4.34e+07 |

Table F.2: **88000 raw data** for three code generation strategies.

| Program | Code Generation Strategy | | |
|---|---|---|---|
| | PP | IPS | RASE |
| **Livermore** | | | |
| Ker01 | 6.12e+07 | 6.11e+07 | 6.11e+07 |
| Ker02 | 7.12e+07 | 5.96e+07 | 5.96e+07 |
| Ker03 | 5.50e+07 | 5.50e+07 | 5.50e+07 |
| Ker04 | 5.02e+07 | 4.96e+07 | 4.96e+07 |
| Ker05 | 5.82e+07 | 5.82e+07 | 5.82e+07 |
| Ker06 | 4.42e+07 | 4.41e+07 | 4.41e+07 |
| Ker07 | 1.00e+08 | 6.30e+07 | 6.38e+07 |
| Ker08 | 1.09e+08 | 8.64e+07 | 8.77e+07 |
| Ker09 | 8.37e+07 | 6.90e+07 | 6.82e+07 |
| Ker10 | 1.01e+08 | 8.53e+07 | 8.53e+07 |
| Ker11 | 5.23e+07 | 5.23e+07 | 5.23e+07 |
| Ker12 | 5.56e+07 | 5.56e+07 | 5.56e+07 |
| Ker13 | 8.29e+07 | 7.17e+07 | 7.17e+07 |
| Ker14 | 8.32e+07 | 7.80e+07 | 7.71e+07 |
| Amean | 7.20e+07 | 6.35e+07 | 6.35e+07 |
| **Nasker** | | | |
| BTR | 1.54e+09 | 1.24e+09 | 1.23e+09 |
| CHO | 1.51e+09 | 1.28e+09 | 1.28e+09 |
| EMI | 7.59e+08 | 6.49e+08 | 6.44e+08 |
| FFT | 1.61e+09 | 1.21e+09 | 1.21e+09 |
| GMT | 1.38e+09 | 1.25e+09 | 1.25e+09 |
| MXM | 1.40e+09 | 1.06e+09 | 1.05e+09 |
| VPE | 6.72e+07 | 5.77e+07 | 5.77e+07 |
| Amean | 1.18e+09 | 9.64e+08 | 9.60e+08 |
| **Optimized Nasker** | | | |
| BTR | 1.28e+09 | 1.07e+09 | 1.05e+09 |
| CHO | 1.04e+09 | 9.69e+08 | 9.68e+08 |
| EMI | 7.30e+08 | 6.69e+08 | 6.49e+08 |
| FFT | 1.44e+09 | 1.18e+09 | 1.17e+09 |
| GMT | 1.17e+09 | 1.12e+09 | 1.12e+09 |
| MXM | 1.19e+09 | 9.21e+08 | 9.21e+08 |
| VPE | 6.54e+07 | 5.77e+07 | 5.77e+07 |
| Amean | 9.88e+08 | 8.55e+08 | 8.48e+08 |

| Program | Code Generation Strategy | | |
|---|---|---|---|
| | PP | IPS | RASE |
| **Perfect** | | | |
| ARC2D | 3.40e+10 | 2.94e+10 | 2.73e+10 |
| BDNA | 9.07e+09 | 8.75e+09 | 7.67e+09 |
| DYFESM | 5.36e+09 | 4.83e+09 | 4.84e+09 |
| FLO52 | 9.27e+09 | 8.33e+09 | 8.31e+09 |
| MDG | 1.88e+10 | 1.80e+10 | 1.79e+10 |
| MG3D | 1.31e+11 | 1.11e+11 | 1.03e+11 |
| QCD | 3.47e+09 | 3.38e+09 | 3.14e+09 |
| TRACK | 1.50e+09 | 1.47e+09 | 1.41e+09 |
| Amean | 2.66e+10 | 2.31e+10 | 2.17e+10 |
| **Misc** | | | |
| BOAST | 7.30e+09 | 6.51e+09 | 6.46e+09 |
| SPHOT | 1.13e+08 | 1.04e+08 | 1.03e+08 |
| WANAL1 | 1.16e+10 | 1.01e+10 | 9.91e+09 |
| NEURAL | 1.36e+09 | 1.09e+09 | 1.10e+09 |
| COSTSC | 1.66e+08 | 1.26e+08 | 1.31e+08 |
| Amean | 4.11e+09 | 3.59e+09 | 3.54e+09 |
| **Int** | | | |
| DHRYSTO | 1.43e+07 | 1.34e+07 | 1.38e+07 |
| INTEGER | 1.64e+08 | 1.54e+08 | 1.54e+08 |
| KNIGHT | 3.80e+06 | 3.04e+06 | 3.07e+06 |
| LCC | 2.18e+07 | 1.57e+07 | 1.62e+07 |
| SPARSE | 3.87e+06 | 3.15e+06 | 3.21e+06 |
| Amean | 4.16e+07 | 3.79e+07 | 3.81e+07 |

Table F.3: **i860 raw data** for three code generation strategies.

| Program | Code Generation Strategy | | |
|---|---|---|---|
| | PP | IPS | RASE |
| **Livermore** | | | |
| Ker01 | 4.63e+07 | 4.63e+07 | 4.63e+07 |
| Ker02 | 5.21e+07 | 5.28e+07 | 5.21e+07 |
| Ker03 | 4.92e+07 | 4.92e+07 | 4.92e+07 |
| Ker04 | 4.55e+07 | 4.53e+07 | 4.53e+07 |
| Ker05 | 4.79e+07 | 4.99e+07 | 4.79e+07 |
| Ker06 | 3.77e+07 | 3.77e+07 | 3.77e+07 |
| Ker07 | 4.90e+07 | 4.90e+07 | 4.90e+07 |
| Ker08 | 7.72e+07 | 7.15e+07 | 7.02e+07 |
| Ker09 | 5.36e+07 | 5.28e+07 | 5.28e+07 |
| Ker10 | 9.47e+07 | 7.95e+07 | 7.95e+07 |
| Ker11 | 4.76e+07 | 4.76e+07 | 4.76e+07 |
| Ker12 | 5.06e+07 | 5.06e+07 | 5.06e+07 |
| Ker13 | 6.69e+07 | 6.63e+07 | 5.86e+07 |
| Ker14 | 7.27e+07 | 7.59e+07 | 6.99e+07 |
| Amean | 5.65e+07 | 5.53e+07 | 5.40e+07 |
| **Nasker** | | | |
| BTR | 1.36e+09 | 1.14e+09 | 1.12e+09 |
| CHO | 1.84e+09 | 1.74e+09 | 1.78e+09 |
| EMI | 6.25e+08 | 6.05e+08 | 5.94e+08 |
| FFT | 1.40e+09 | 1.46e+09 | 1.38e+09 |
| GMT | 1.45e+09 | 1.45e+09 | 1.50e+09 |
| MXM | 1.10e+09 | 1.05e+09 | 9.72e+08 |
| VPE | 4.56e+07 | 4.69e+07 | 5.03e+07 |
| Amean | 1.12e+09 | 1.07e+09 | 1.06e+09 |
| **Optimized Nasker** | | | |
| BTR | 1.03e+09 | 8.90e+08 | 8.79e+08 |
| CHO | 9.02e+08 | 8.36e+08 | 8.35e+08 |
| EMI | 5.92e+08 | 5.87e+08 | 5.82e+08 |
| FFT | 1.26e+09 | 1.15e+09 | 1.13e+09 |
| GMT | 9.52e+08 | 9.50e+08 | 9.51e+08 |
| MXM | 7.80e+08 | 7.00e+08 | 7.00e+08 |
| VPE | 4.56e+07 | 4.69e+07 | 5.03e+07 |
| Amean | 7.95e+08 | 7.37e+08 | 7.32e+08 |

| Program | Code Generation Strategy | | |
|---|---|---|---|
| | PP | IPS | RASE |
| **Perfect** | | | |
| ARC2D | 2.56e+10 | 2.26e+10 | 2.20e+10 |
| BDNA | 4.41e+09 | 4.47e+09 | 4.37e+09 |
| DYFESM | 5.48e+09 | 5.17e+09 | 5.18e+09 |
| FLO52 | 9.07e+09 | 8.38e+09 | 8.08e+09 |
| MDG | 1.29e+10 | 1.25e+10 | 1.25e+10 |
| MG3D | 1.07e+11 | 1.02e+11 | 9.89e+10 |
| QCD | 3.48e+09 | 3.32e+09 | 3.33e+09 |
| TRACK | 1.02e+09 | 9.88e+08 | 9.98e+08 |
| Amean | 2.11e+10 | 1.99e+10 | 1.94e+10 |
| **Misc** | | | |
| BOAST | 4.90e+09 | 4.53e+09 | 4.42e+09 |
| SPHOT | 8.89e+07 | 8.08e+07 | 7.99e+07 |
| WANAL1 | 8.62e+09 | 8.17e+09 | 8.14e+09 |
| NEURAL | 1.02e+09 | 9.18e+08 | 9.16e+08 |
| COSTSC | 1.13e+08 | 8.28e+07 | 8.28e+07 |
| Amean | 2.95e+09 | 2.76e+09 | 2.73e+09 |
| **Int** | | | |
| DHRYSTO | 1.51e+07 | 1.41e+07 | 1.46e+07 |
| INTEGER | 2.40e+08 | 2.15e+08 | 2.20e+08 |
| KNIGHT | 4.36e+06 | 3.77e+06 | 3.74e+06 |
| LCC | 2.31e+07 | 1.71e+07 | 1.79e+07 |
| SPARSE | 3.52e+06 | 2.88e+06 | 2.96e+06 |
| Amean | 5.72e+07 | 5.06e+07 | 5.18e+07 |

Table F.4: **A88sh Postpass raw data** for four register set sizes.

| Prog | Register Set Size 16 | 32 | 64 | 128 |
|---|---|---|---|---|
| Livermore | | | | |
| Ker01 | 7.59e+07 | 5.81e+07 | 5.81e+07 | 5.81e+07 |
| Ker02 | 7.47e+07 | 6.68e+07 | 5.78e+07 | 5.78e+07 |
| Ker03 | 5.31e+07 | 5.10e+07 | 5.10e+07 | 5.10e+07 |
| Ker04 | 5.15e+07 | 4.67e+07 | 4.63e+07 | 4.63e+07 |
| Ker05 | 7.05e+07 | 5.40e+07 | 5.40e+07 | 5.40e+07 |
| Ker06 | 4.53e+07 | 4.13e+07 | 4.12e+07 | 4.12e+07 |
| Ker07 | 1.07e+08 | 9.35e+07 | 6.30e+07 | 6.30e+07 |
| Ker08 | 1.29e+08 | 1.06e+08 | 9.20e+07 | 8.94e+07 |
| Ker09 | 8.28e+07 | 8.21e+07 | 6.90e+07 | 6.81e+07 |
| Ker10 | 1.01e+08 | 1.00e+08 | 8.53e+07 | 8.23e+07 |
| Ker11 | 5.47e+07 | 4.99e+07 | 4.99e+07 | 4.99e+07 |
| Ker12 | 5.81e+07 | 5.30e+07 | 5.30e+07 | 5.30e+07 |
| Ker13 | 8.86e+07 | 8.14e+07 | 7.22e+07 | 7.11e+07 |
| Ker14 | 9.53e+07 | 7.99e+07 | 7.39e+07 | 7.35e+07 |
| Amean | 7.77e+07 | 6.88e+07 | 6.19e+07 | 6.13e+07 |
| Nasker | | | | |
| BTR | 1.83e+09 | 1.49e+09 | 1.26e+09 | 1.22e+09 |
| CHO | 1.81e+09 | 1.45e+09 | 1.41e+09 | 1.38e+09 |
| EMI | 8.08e+08 | 7.32e+08 | 6.66e+08 | 6.36e+08 |
| FFT | 1.77e+09 | 1.59e+09 | 1.23e+09 | 1.22e+09 |
| GMT | 1.46e+09 | 1.34e+09 | 1.21e+09 | 1.21e+09 |
| MXM | 1.58e+09 | 1.35e+09 | 1.10e+09 | 1.02e+09 |
| VPE | 8.24e+07 | 6.51e+07 | 5.65e+07 | 5.74e+07 |
| Amean | 1.33e+09 | 1.15e+09 | 9.90e+08 | 9.63e+08 |
| Optimized Nasker | | | | |
| BTR | 1.52e+09 | 1.24e+09 | 1.06e+09 | 1.04e+09 |
| CHO | 1.25e+09 | 1.00e+09 | 9.53e+08 | 9.53e+08 |
| EMI | 7.85e+08 | 7.22e+08 | 7.01e+08 | 6.66e+08 |
| FFT | 1.66e+09 | 1.42e+09 | 1.19e+09 | 1.18e+09 |
| GMT | 1.42e+09 | 1.13e+09 | 1.17e+09 | 1.08e+09 |
| MXM | 1.43e+09 | 1.17e+09 | 9.05e+08 | 9.05e+08 |
| VPE | 8.00e+07 | 6.35e+07 | 5.65e+07 | 5.74e+07 |
| Amean | 1.16e+09 | 9.64e+08 | 8.62e+08 | 8.40e+08 |

| Prog | Register Set Size 16 | 32 | 64 | 128 |
|---|---|---|---|---|
| Perfect | | | | |
| ARC2D | 4.16e+10 | 3.30e+10 | 2.87e+10 | 2.68e+10 |
| BDNA | 1.16e+10 | 8.82e+09 | 7.19e+09 | 6.72e+09 |
| DYFESM | 5.50e+09 | 4.94e+09 | 4.72e+09 | 4.44e+09 |
| FLO52 | 1.05e+10 | 8.95e+09 | 8.34e+09 | 8.32e+09 |
| MDG | 1.84e+10 | 1.77e+10 | 1.75e+10 | 1.82e+10 |
| MG3D | 1.63e+11 | 1.25e+11 | 1.13e+11 | 1.03e+11 |
| QCD | 3.26e+09 | 3.23e+09 | 3.29e+09 | 3.42e+09 |
| TRACK | 1.54e+09 | 1.42e+09 | 1.40e+09 | 1.43e+09 |
| Amean | 3.19e+10 | 2.54e+10 | 2.30e+10 | 2.15e+10 |
| Misc | | | | |
| BOAST | 8.43e+09 | 7.01e+09 | 6.57e+09 | 6.42e+09 |
| SPHOT | 1.18e+08 | 1.09e+08 | 1.08e+08 | 1.07e+08 |
| WANAL1 | 1.33e+10 | 1.14e+10 | 1.02e+10 | 1.02e+10 |
| NEURAL | 1.44e+09 | 1.28e+09 | 1.15e+09 | 1.16e+09 |
| COSTSC | 1.58e+08 | 1.53e+08 | 1.36e+08 | 1.31e+08 |
| Amean | 4.69e+09 | 3.99e+09 | 3.63e+09 | 3.60e+09 |
| Int | | | | |
| DHRYSTO | 1.31e+07 | 1.33e+07 | 1.33e+07 | 1.33e+07 |
| INTEGER | 1.76e+08 | 1.64e+08 | 1.59e+08 | 1.54e+08 |
| KNIGHT | 3.71e+06 | 3.70e+06 | 3.54e+06 | 3.54e+06 |
| LCC | 1.76e+07 | 2.00e+07 | 2.37e+07 | 2.74e+07 |
| SPARSE | 3.45e+06 | 3.55e+06 | 3.87e+06 | 3.90e+06 |
| Amean | 4.28e+07 | 4.09e+07 | 4.07e+07 | 4.04e+07 |

Table F.5: **A88sh IPS raw data** for four register set sizes.

| Prog | Register Set Size 16 | 32 | 64 | 128 |
|---|---|---|---|---|
| Livermore | | | | |
| Ker01 | 5.81e+07 | 5.81e+07 | 5.81e+07 | 5.81e+07 |
| Ker02 | 5.78e+07 | 5.78e+07 | 5.78e+07 | 5.78e+07 |
| Ker03 | 5.10e+07 | 5.10e+07 | 5.10e+07 | 5.10e+07 |
| Ker04 | 4.63e+07 | 4.63e+07 | 4.63e+07 | 4.63e+07 |
| Ker05 | 5.40e+07 | 5.40e+07 | 5.40e+07 | 5.40e+07 |
| Ker06 | 4.12e+07 | 4.12e+07 | 4.12e+07 | 4.12e+07 |
| Ker07 | 6.30e+07 | 6.30e+07 | 6.30e+07 | 6.30e+07 |
| Ker08 | 1.22e+08 | 8.54e+07 | 8.54e+07 | 8.72e+07 |
| Ker09 | 6.74e+07 | 6.74e+07 | 6.74e+07 | 6.74e+07 |
| Ker10 | 9.10e+07 | 8.45e+07 | 8.45e+07 | 8.23e+07 |
| Ker11 | 4.99e+07 | 4.99e+07 | 4.99e+07 | 4.99e+07 |
| Ker12 | 5.30e+07 | 5.30e+07 | 5.30e+07 | 5.30e+07 |
| Ker13 | 7.75e+07 | 7.11e+07 | 7.11e+07 | 7.11e+07 |
| Ker14 | 7.63e+07 | 7.35e+07 | 7.35e+07 | 7.35e+07 |
| Amean | 6.49e+07 | 6.12e+07 | 6.12e+07 | 6.11e+07 |
| Nasker | | | | |
| BTR | 1.43e+09 | 1.22e+09 | 1.20e+09 | 1.20e+09 |
| CHO | 1.39e+09 | 1.25e+09 | 1.25e+09 | 1.25e+09 |
| EMI | 8.12e+08 | 6.16e+08 | 6.11e+08 | 6.26e+08 |
| FFT | 1.44e+09 | 1.23e+09 | 1.23e+09 | 1.23e+09 |
| GMT | 1.21e+09 | 1.21e+09 | 1.21e+09 | 1.21e+09 |
| MXM | 1.35e+09 | 1.02e+09 | 1.02e+09 | 1.02e+09 |
| VPE | 6.69e+07 | 5.64e+07 | 5.50e+07 | 5.64e+07 |
| Amean | 1.10e+09 | 9.43e+08 | 9.39e+08 | 9.42e+08 |
| Optimized Nasker | | | | |
| BTR | 1.33e+09 | 1.04e+09 | 1.03e+09 | 1.03e+09 |
| CHO | 9.90e+08 | 9.29e+08 | 9.29e+08 | 9.29e+08 |
| EMI | 7.97e+08 | 6.51e+08 | 6.46e+08 | 6.51e+08 |
| FFT | 1.32e+09 | 1.18e+09 | 1.18e+09 | 1.18e+09 |
| GMT | 1.09e+09 | 1.08e+09 | 1.08e+09 | 1.08e+09 |
| MXM | 1.03e+09 | 9.05e+08 | 9.05e+08 | 9.05e+08 |
| VPE | 6.78e+07 | 5.64e+07 | 5.50e+07 | 5.64e+07 |
| Amean | 9.46e+08 | 8.34e+08 | 8.32e+08 | 8.33e+08 |

| Prog | Register Set Size 16 | 32 | 64 | 128 |
|---|---|---|---|---|
| Perfect | | | | |
| ARC2D | 3.86e+10 | 2.83e+10 | 2.60e+10 | 2.58e+10 |
| BDNA | 1.12e+10 | 8.49e+09 | 7.36e+09 | 6.65e+09 |
| DYFESM | 4.41e+09 | 4.41e+09 | 4.41e+09 | 4.41e+09 |
| FLO52 | 9.04e+09 | 8.09e+09 | 8.04e+09 | 8.06e+09 |
| MDG | 1.73e+10 | 1.69e+10 | 1.68e+10 | 1.68e+10 |
| MG3D | 1.48e+11 | 1.06e+11 | 9.82e+10 | 9.85e+10 |
| QCD | 3.16e+09 | 3.11e+09 | 3.10e+09 | 3.11e+09 |
| TRACK | 1.50e+09 | 1.38e+09 | 1.35e+09 | 1.35e+09 |
| Amean | 2.92e+10 | 2.21e+10 | 2.07e+10 | 2.06e+10 |
| Misc | | | | |
| BOAST | 7.25e+09 | 6.25e+09 | 6.18e+09 | 6.18e+09 |
| SPHOT | 1.12e+08 | 1.01e+08 | 1.00e+08 | 1.00e+08 |
| WANAL1 | 1.12e+10 | 9.68e+09 | 9.47e+09 | 9.47e+09 |
| NEURAL | 1.33e+09 | 1.02e+09 | 1.02e+09 | 1.02e+09 |
| COSTSC | 1.46e+08 | 1.17e+08 | 1.15e+08 | 1.15e+08 |
| Amean | 4.01e+09 | 3.43e+09 | 3.38e+09 | 3.38e+09 |
| Int | | | | |
| DHRYSTO | 1.24e+07 | 1.24e+07 | 1.24e+07 | 1.24e+07 |
| INTEGER | 1.54e+08 | 1.54e+08 | 1.54e+08 | 1.54e+08 |
| KNIGHT | 3.01e+06 | 3.01e+06 | 3.01e+06 | 3.01e+06 |
| LCC | 1.45e+07 | 1.45e+07 | 1.45e+07 | 1.45e+07 |
| SPARSE | 2.95e+06 | 2.89e+06 | 2.88e+06 | 2.89e+06 |
| Amean | 3.74e+07 | 3.74e+07 | 3.74e+07 | 3.74e+07 |

Table F.6: **A88sh RASE raw data** for four register set sizes.

| Prog | Register Set Size 16 | 32 | 64 | 128 |
|---|---|---|---|---|
| Livermore | | | | |
| Ker01 | 5.81e+07 | 5.81e+07 | 5.81e+07 | 5.81e+07 |
| Ker02 | 5.78e+07 | 5.78e+07 | 5.78e+07 | 5.78e+07 |
| Ker03 | 5.10e+07 | 5.10e+07 | 5.10e+07 | 5.10e+07 |
| Ker04 | 4.63e+07 | 4.63e+07 | 4.63e+07 | 4.63e+07 |
| Ker05 | 5.40e+07 | 5.40e+07 | 5.40e+07 | 5.40e+07 |
| Ker06 | 4.12e+07 | 4.12e+07 | 4.12e+07 | 4.12e+07 |
| Ker07 | 6.30e+07 | 6.30e+07 | 6.30e+07 | 6.30e+07 |
| Ker08 | 9.07e+07 | 8.59e+07 | 8.59e+07 | 8.76e+07 |
| Ker09 | 6.74e+07 | 6.74e+07 | 6.74e+07 | 6.74e+07 |
| Ker10 | 8.53e+07 | 8.45e+07 | 8.45e+07 | 8.23e+07 |
| Ker11 | 4.99e+07 | 4.99e+07 | 4.99e+07 | 4.99e+07 |
| Ker12 | 5.30e+07 | 5.30e+07 | 5.30e+07 | 5.30e+07 |
| Ker13 | 7.17e+07 | 7.11e+07 | 7.11e+07 | 7.11e+07 |
| Ker14 | 7.79e+07 | 7.35e+07 | 7.35e+07 | 7.35e+07 |
| Amean | 6.19e+07 | 6.12e+07 | 6.12e+07 | 6.12e+07 |
| Nasker | | | | |
| BTR | 1.35e+09 | 1.21e+09 | 1.19e+09 | 1.19e+09 |
| CHO | 1.39e+09 | 1.26e+09 | 1.26e+09 | 1.26e+09 |
| EMI | 8.42e+08 | 6.16e+08 | 6.16e+08 | 6.16e+08 |
| FFT | 1.32e+09 | 1.23e+09 | 1.23e+09 | 1.23e+09 |
| GMT | 1.21e+09 | 1.21e+09 | 1.21e+09 | 1.21e+09 |
| MXM | 1.30e+09 | 1.03e+09 | 1.03e+09 | 1.03e+09 |
| VPE | 5.97e+07 | 5.65e+07 | 5.50e+07 | 5.65e+07 |
| Amean | 1.07e+09 | 9.45e+08 | 9.42e+08 | 9.42e+08 |
| Optimized Nasker | | | | |
| BTR | 1.14e+09 | 1.03e+09 | 1.02e+09 | 1.03e+09 |
| CHO | 9.92e+08 | 9.29e+08 | 9.29e+08 | 9.29e+08 |
| EMI | 8.77e+08 | 6.46e+08 | 6.46e+08 | 6.46e+08 |
| FFT | 1.28e+09 | 1.18e+09 | 1.18e+09 | 1.18e+09 |
| GMT | 1.08e+09 | 1.08e+09 | 1.08e+09 | 1.08e+09 |
| MXM | 1.02e+09 | 9.05e+08 | 9.05e+08 | 9.05e+08 |
| VPE | 5.97e+07 | 5.65e+07 | 5.50e+07 | 5.65e+07 |
| Amean | 9.21e+08 | 8.32e+08 | 8.31e+08 | 8.32e+08 |

| Prog | Register Set Size 16 | 32 | 64 | 128 |
|---|---|---|---|---|
| Perfect | | | | |
| ARC2D | 3.47e+10 | 2.65e+10 | 2.56e+10 | 2.56e+10 |
| BDNA | 1.24e+10 | 7.70e+09 | 6.85e+09 | 6.48e+09 |
| DYFESM | 4.41e+09 | 4.41e+09 | 4.41e+09 | 4.41e+09 |
| FLO52 | 8.47e+09 | 8.05e+09 | 8.04e+09 | 8.05e+09 |
| MDG | 1.71e+10 | 1.68e+10 | 1.68e+10 | 1.68e+10 |
| MG3D | 1.22e+11 | 1.00e+11 | 9.74e+10 | 9.74e+10 |
| QCD | 2.94e+09 | 2.92e+09 | 2.93e+09 | 2.93e+09 |
| TRACK | 1.54e+09 | 1.34e+09 | 1.33e+09 | 1.33e+09 |
| Amean | 2.54e+10 | 2.10e+10 | 2.04e+10 | 2.04e+10 |
| Misc | | | | |
| BOAST | 6.90e+09 | 6.23e+09 | 6.16e+09 | 6.16e+09 |
| SPHOT | 1.07e+08 | 9.98e+07 | 9.98e+07 | 9.98e+07 |
| WANAL1 | 9.81e+09 | 9.59e+09 | 9.59e+09 | 9.59e+09 |
| NEURAL | 1.24e+09 | 1.03e+09 | 1.03e+09 | 1.03e+09 |
| COSTSC | 1.50e+08 | 1.22e+08 | 1.15e+08 | 1.15e+08 |
| Amean | 3.64e+09 | 3.41e+09 | 3.40e+09 | 3.40e+09 |
| Int | | | | |
| DHRYSTO | 1.28e+07 | 1.28e+07 | 1.28e+07 | 1.28e+07 |
| INTEGER | 1.54e+08 | 1.54e+08 | 1.54e+08 | 1.54e+08 |
| KNIGHT | 3.04e+06 | 3.04e+06 | 3.04e+06 | 3.04e+06 |
| LCC | 1.50e+07 | 1.50e+07 | 1.50e+07 | 1.50e+07 |
| SPARSE | 3.02e+06 | 2.95e+06 | 2.95e+06 | 2.95e+06 |
| Amean | 3.76e+07 | 3.76e+07 | 3.76e+07 | 3.76e+07 |

Table F.7: **Ar2sh Postpass raw data** for four register set sizes.

| Prog | Register Set Size 16 | 32 | 64 | 128 |
|---|---|---|---|---|
| Livermore | | | | |
| Ker01 | 6.41e+07 | 5.22e+07 | 5.22e+07 | 5.22e+07 |
| Ker02 | 6.78e+07 | 5.74e+07 | 5.61e+07 | 5.61e+07 |
| Ker03 | 4.72e+07 | 4.52e+07 | 4.52e+07 | 4.52e+07 |
| Ker04 | 4.84e+07 | 4.38e+07 | 4.33e+07 | 4.33e+07 |
| Ker05 | 6.02e+07 | 5.40e+07 | 5.40e+07 | 5.40e+07 |
| Ker06 | 4.53e+07 | 3.76e+07 | 3.75e+07 | 3.75e+07 |
| Ker07 | 8.53e+07 | 7.46e+07 | 5.64e+07 | 5.64e+07 |
| Ker08 | 1.32e+08 | 1.08e+08 | 8.63e+07 | 8.11e+07 |
| Ker09 | 7.90e+07 | 7.67e+07 | 5.97e+07 | 5.97e+07 |
| Ker10 | 1.07e+08 | 1.00e+08 | 8.53e+07 | 8.23e+07 |
| Ker11 | 5.23e+07 | 4.99e+07 | 4.99e+07 | 4.99e+07 |
| Ker12 | 5.56e+07 | 5.30e+07 | 5.30e+07 | 5.30e+07 |
| Ker13 | 8.72e+07 | 8.09e+07 | 7.07e+07 | 6.92e+07 |
| Ker14 | 9.04e+07 | 7.31e+07 | 7.23e+07 | 7.23e+07 |
| Amean | 7.30e+07 | 6.47e+07 | 5.87e+07 | 5.80e+07 |
| Nasker | | | | |
| BTR | 2.16e+09 | 1.73e+09 | 1.49e+09 | 1.32e+09 |
| CHO | 2.94e+09 | 2.66e+09 | 2.60e+09 | 2.51e+09 |
| EMI | 6.97e+08 | 6.26e+08 | 6.11e+08 | 5.83e+08 |
| FFT | 1.85e+09 | 1.66e+09 | 1.37e+09 | 1.34e+09 |
| GMT | 1.74e+09 | 1.66e+09 | 1.61e+09 | 1.61e+09 |
| MXM | 2.01e+09 | 1.52e+09 | 1.38e+09 | 1.36e+09 |
| VPE | 6.83e+07 | 5.30e+07 | 4.56e+07 | 4.61e+07 |
| Amean | 1.64e+09 | 1.42e+09 | 1.30e+09 | 1.25e+09 |
| Optimized Nasker | | | | |
| BTR | 1.45e+09 | 1.19e+09 | 9.51e+08 | 9.23e+08 |
| CHO | 1.08e+09 | 8.40e+08 | 8.03e+08 | 8.03e+08 |
| EMI | 6.89e+08 | 6.25e+08 | 5.89e+08 | 5.88e+08 |
| FFT | 1.60e+09 | 1.39e+09 | 1.16e+09 | 1.15e+09 |
| GMT | 1.28e+09 | 9.46e+08 | 9.44e+08 | 9.01e+08 |
| MXM | 1.31e+09 | 1.12e+09 | 7.62e+08 | 7.62e+08 |
| VPE | 6.79e+07 | 5.30e+07 | 4.56e+07 | 4.61e+07 |
| Amean | 1.07e+09 | 8.81e+08 | 7.51e+08 | 7.39e+08 |

| Prog | Register Set Size 16 | 32 | 64 | 128 |
|---|---|---|---|---|
| Perfect | | | | |
| ARC2D | 4.43e+10 | 3.54e+10 | 3.01e+10 | 2.78e+10 |
| BDNA | 9.15e+09 | 6.88e+09 | 5.83e+09 | 5.35e+09 |
| DYFESM | 6.97e+09 | 6.15e+09 | 5.76e+09 | 5.71e+09 |
| FLO52 | 1.27e+10 | 1.06e+10 | 1.00e+10 | 9.67e+09 |
| MDG | 1.41e+10 | 1.35e+10 | 1.38e+10 | 1.47e+10 |
| MG3D | 1.62e+11 | 1.26e+11 | 1.14e+11 | 9.87e+10 |
| QCD | 3.36e+09 | 3.37e+09 | 3.45e+09 | 3.57e+09 |
| TRACK | 1.25e+09 | 1.16e+09 | 1.16e+09 | 1.20e+09 |
| Amean | 3.17e+10 | 2.54e+10 | 2.30e+10 | 2.08e+10 |
| Misc | | | | |
| BOAST | 7.03e+09 | 5.60e+09 | 5.19e+09 | 5.10e+09 |
| SPHOT | 8.84e+07 | 8.24e+07 | 8.12e+07 | 8.08e+07 |
| WANAL1 | 1.49e+10 | 1.20e+10 | 1.12e+10 | 1.12e+10 |
| NEURAL | 1.40e+09 | 1.22e+09 | 1.09e+09 | 1.10e+09 |
| COSTSC | 1.37e+08 | 1.31e+08 | 1.16e+08 | 1.12e+08 |
| Amean | 4.71e+09 | 3.81e+09 | 3.54e+09 | 3.52e+09 |
| Int | | | | |
| DHRYSTO | 1.35e+07 | 1.36e+07 | 1.36e+07 | 1.36e+07 |
| INTEGER | 1.91e+08 | 1.78e+08 | 1.66e+08 | 1.57e+08 |
| KNIGHT | 4.10e+06 | 4.08e+06 | 3.91e+06 | 3.91e+06 |
| LCC | 1.76e+07 | 2.00e+07 | 2.37e+07 | 2.74e+07 |
| SPARSE | 3.22e+06 | 3.33e+06 | 3.65e+06 | 3.67e+06 |
| Amean | 4.59e+07 | 4.38e+07 | 4.22e+07 | 4.11e+07 |

Table F.8: **Ar2sh IPS raw data** for four register set sizes.

| Prog | Register Set Size 16 | 32 | 64 | 128 |
|---|---|---|---|---|
| Livermore | | | | |
| Ker01 | 5.37e+07 | 5.37e+07 | 5.37e+07 | 5.37e+07 |
| Ker02 | 5.61e+07 | 5.61e+07 | 5.61e+07 | 5.61e+07 |
| Ker03 | 4.52e+07 | 4.52e+07 | 4.52e+07 | 4.52e+07 |
| Ker04 | 4.33e+07 | 4.33e+07 | 4.33e+07 | 4.33e+07 |
| Ker05 | 5.40e+07 | 5.40e+07 | 5.40e+07 | 5.40e+07 |
| Ker06 | 3.75e+07 | 3.75e+07 | 3.75e+07 | 3.75e+07 |
| Ker07 | 5.80e+07 | 5.80e+07 | 5.80e+07 | 5.80e+07 |
| Ker08 | 1.21e+08 | 8.06e+07 | 8.06e+07 | 8.24e+07 |
| Ker09 | 6.12e+07 | 6.12e+07 | 6.12e+07 | 6.12e+07 |
| Ker10 | 9.10e+07 | 8.45e+07 | 8.45e+07 | 8.23e+07 |
| Ker11 | 4.99e+07 | 4.99e+07 | 4.99e+07 | 4.99e+07 |
| Ker12 | 5.30e+07 | 5.30e+07 | 5.30e+07 | 5.30e+07 |
| Ker13 | 7.56e+07 | 7.02e+07 | 7.02e+07 | 7.02e+07 |
| Ker14 | 7.27e+07 | 7.23e+07 | 7.23e+07 | 7.23e+07 |
| Amean | 6.23e+07 | 5.85e+07 | 5.85e+07 | 5.85e+07 |
| Nasker | | | | |
| BTR | 1.62e+09 | 1.32e+09 | 1.30e+09 | 1.32e+09 |
| CHO | 2.54e+09 | 2.43e+09 | 2.43e+09 | 2.43e+09 |
| EMI | 6.84e+08 | 5.78e+08 | 5.83e+08 | 5.88e+08 |
| FFT | 1.58e+09 | 1.33e+09 | 1.34e+09 | 1.34e+09 |
| GMT | 1.62e+09 | 1.69e+09 | 1.61e+09 | 1.61e+09 |
| MXM | 1.74e+09 | 1.44e+09 | 1.44e+09 | 1.44e+09 |
| VPE | 5.33e+07 | 4.64e+07 | 4.59e+07 | 4.64e+07 |
| Amean | 1.41e+09 | 1.26e+09 | 1.25e+09 | 1.25e+09 |
| Optimized Nasker | | | | |
| BTR | 1.17e+09 | 9.24e+08 | 9.32e+08 | 9.30e+08 |
| CHO | 8.39e+08 | 7.79e+08 | 7.79e+08 | 7.79e+08 |
| EMI | 6.59e+08 | 5.78e+08 | 5.78e+08 | 5.78e+08 |
| FFT | 1.29e+09 | 1.15e+09 | 1.15e+09 | 1.15e+09 |
| GMT | 9.06e+08 | 9.01e+08 | 9.01e+08 | 9.01e+08 |
| MXM | 1.01e+09 | 7.77e+08 | 7.77e+08 | 7.77e+08 |
| VPE | 5.32e+07 | 4.64e+07 | 4.59e+07 | 4.64e+07 |
| Amean | 8.47e+08 | 7.36e+08 | 7.38e+08 | 7.37e+08 |

| Prog | Register Set Size 16 | 32 | 64 | 128 |
|---|---|---|---|---|
| Perfect | | | | |
| ARC2D | 4.05e+10 | 2.89e+10 | 2.71e+10 | 2.70e+10 |
| BDNA | 8.75e+09 | 6.40e+09 | 5.58e+09 | 5.43e+09 |
| DYFESM | 5.70e+09 | 5.69e+09 | 5.69e+09 | 5.69e+09 |
| FLO52 | 1.05e+10 | 9.50e+09 | 9.42e+09 | 9.42e+09 |
| MDG | 1.34e+10 | 1.29e+10 | 1.29e+10 | 1.29e+10 |
| MG3D | 1.42e+11 | 9.94e+10 | 9.82e+10 | 9.84e+10 |
| QCD | 2.96e+09 | 2.93e+09 | 2.92e+09 | 2.92e+09 |
| TRACK | 1.22e+09 | 1.12e+09 | 1.10e+09 | 1.09e+09 |
| Amean | 2.81e+10 | 2.09e+10 | 2.04e+10 | 2.04e+10 |
| Misc | | | | |
| BOAST | 5.91e+09 | 5.03e+09 | 4.99e+09 | 4.99e+09 |
| SPHOT | 8.22e+07 | 7.48e+07 | 7.48e+07 | 7.48e+07 |
| WANAL1 | 1.28e+10 | 1.06e+10 | 1.06e+10 | 1.06e+10 |
| NEURAL | 1.44e+09 | 1.00e+09 | 1.00e+09 | 1.00e+09 |
| COSTSC | 1.24e+08 | 9.56e+07 | 9.38e+07 | 9.38e+07 |
| Amean | 4.07e+09 | 3.36e+09 | 3.35e+09 | 3.35e+09 |
| Int | | | | |
| DHRYSTO | 1.28e+07 | 1.28e+07 | 1.28e+07 | 1.28e+07 |
| INTEGER | 1.57e+08 | 1.57e+08 | 1.57e+08 | 1.57e+08 |
| KNIGHT | 3.38e+06 | 3.38e+06 | 3.38e+06 | 3.38e+06 |
| LCC | 1.45e+07 | 1.45e+07 | 1.45e+07 | 1.45e+07 |
| SPARSE | 2.72e+06 | 2.66e+06 | 2.66e+06 | 2.66e+06 |
| Amean | 3.81e+07 | 3.81e+07 | 3.81e+07 | 3.81e+07 |

Table F.9: **Ar2sh RASE raw data** for four register set sizes.

| Prog | Register Set Size 16 | 32 | 64 | 128 |
|---|---|---|---|---|
| Livermore | | | | |
| Ker01 | 5.22e+07 | 5.22e+07 | 5.22e+07 | 5.22e+07 |
| Ker02 | 5.61e+07 | 5.61e+07 | 5.61e+07 | 5.61e+07 |
| Ker03 | 4.52e+07 | 4.52e+07 | 4.52e+07 | 4.52e+07 |
| Ker04 | 4.33e+07 | 4.33e+07 | 4.33e+07 | 4.33e+07 |
| Ker05 | 5.40e+07 | 5.40e+07 | 5.40e+07 | 5.40e+07 |
| Ker06 | 3.75e+07 | 3.75e+07 | 3.75e+07 | 3.75e+07 |
| Ker07 | 5.72e+07 | 5.72e+07 | 5.72e+07 | 5.72e+07 |
| Ker08 | 9.68e+07 | 8.02e+07 | 8.02e+07 | 8.19e+07 |
| Ker09 | 5.97e+07 | 5.97e+07 | 5.97e+07 | 5.97e+07 |
| Ker10 | 8.45e+07 | 8.45e+07 | 8.45e+07 | 8.23e+07 |
| Ker11 | 4.99e+07 | 4.99e+07 | 4.99e+07 | 4.99e+07 |
| Ker12 | 5.30e+07 | 5.30e+07 | 5.30e+07 | 5.30e+07 |
| Ker13 | 6.92e+07 | 6.92e+07 | 6.92e+07 | 6.92e+07 |
| Ker14 | 7.23e+07 | 7.23e+07 | 7.23e+07 | 7.23e+07 |
| Amean | 5.93e+07 | 5.82e+07 | 5.82e+07 | 5.81e+07 |
| Nasker | | | | |
| BTR | 1.59e+09 | 1.29e+09 | 1.29e+09 | 1.29e+09 |
| CHO | 2.58e+09 | 2.43e+09 | 2.43e+09 | 2.43e+09 |
| EMI | 7.51e+08 | 5.78e+08 | 5.88e+08 | 5.88e+08 |
| FFT | 1.45e+09 | 1.33e+09 | 1.34e+09 | 1.34e+09 |
| GMT | 1.61e+09 | 1.61e+09 | 1.61e+09 | 1.61e+09 |
| MXM | 1.52e+09 | 1.36e+09 | 1.36e+09 | 1.36e+09 |
| VPE | 5.03e+07 | 4.61e+07 | 4.50e+07 | 4.61e+07 |
| Amean | 1.36e+09 | 1.23e+09 | 1.24e+09 | 1.24e+09 |
| Optimized Nasker | | | | |
| BTR | 1.07e+09 | 9.18e+08 | 9.08e+08 | 9.09e+08 |
| CHO | 8.42e+08 | 7.79e+08 | 7.79e+08 | 7.79e+08 |
| EMI | 7.83e+08 | 5.78e+08 | 5.98e+08 | 5.98e+08 |
| FFT | 1.26e+09 | 1.17e+09 | 1.15e+09 | 1.15e+09 |
| GMT | 9.04e+08 | 9.00e+08 | 9.00e+08 | 9.00e+08 |
| MXM | 9.98e+08 | 7.62e+08 | 7.62e+08 | 7.62e+08 |
| VPE | 5.03e+07 | 4.61e+07 | 4.50e+07 | 4.61e+07 |
| Amean | 8.44e+08 | 7.36e+08 | 7.35e+08 | 7.35e+08 |

| Prog | Register Set Size 16 | 32 | 64 | 128 |
|---|---|---|---|---|
| Perfect | | | | |
| ARC2D | 3.94e+10 | 2.82e+10 | 2.68e+10 | 2.68e+10 |
| BDNA | 9.28e+09 | 5.95e+09 | 5.42e+09 | 5.35e+09 |
| DYFESM | 5.69e+09 | 5.69e+09 | 5.69e+09 | 5.69e+09 |
| FLO52 | 1.00e+10 | 9.48e+09 | 9.30e+09 | 9.30e+09 |
| MDG | 1.32e+10 | 1.30e+10 | 1.29e+10 | 1.30e+10 |
| MG3D | 1.26e+11 | 9.70e+10 | 9.69e+10 | 9.69e+10 |
| QCD | 3.00e+09 | 2.97e+09 | 2.96e+09 | 2.96e+09 |
| TRACK | 1.27e+09 | 1.10e+09 | 1.09e+09 | 1.09e+09 |
| Amean | 2.60e+10 | 2.04e+10 | 2.01e+10 | 2.01e+10 |
| Misc | | | | |
| BOAST | 5.93e+09 | 4.92e+09 | 4.91e+09 | 4.90e+09 |
| SPHOT | 7.73e+07 | 7.47e+07 | 7.47e+07 | 7.47e+07 |
| WANAL1 | 1.19e+10 | 1.04e+10 | 1.04e+10 | 1.04e+10 |
| NEURAL | 1.22e+09 | 9.73e+08 | 9.73e+08 | 9.73e+08 |
| COSTSC | 1.31e+08 | 1.01e+08 | 9.53e+07 | 9.53e+07 |
| Amean | 3.85e+09 | 3.29e+09 | 3.29e+09 | 3.29e+09 |
| Int | | | | |
| DHRYSTO | 1.32e+07 | 1.32e+07 | 1.32e+07 | 1.32e+07 |
| INTEGER | 1.57e+08 | 1.57e+08 | 1.57e+08 | 1.57e+08 |
| KNIGHT | 3.41e+06 | 3.41e+06 | 3.41e+06 | 3.41e+06 |
| LCC | 1.50e+07 | 1.50e+07 | 1.50e+07 | 1.50e+07 |
| SPARSE | 2.79e+06 | 2.73e+06 | 2.73e+06 | 2.73e+06 |
| Amean | 3.83e+07 | 3.83e+07 | 3.83e+07 | 3.83e+07 |

Table F.10: **Ar2shb Postpass raw data** for four register set sizes.

| Prog | Register Set Size | | | |
|---|---|---|---|---|
| | 16 | 32 | 64 | 128 |
| **Livermore** | | | | |
| Ker01 | 6.11e+07 | 4.48e+07 | 4.48e+07 | 4.48e+07 |
| Ker02 | 6.53e+07 | 5.74e+07 | 5.23e+07 | 5.23e+07 |
| Ker03 | 4.34e+07 | 4.13e+07 | 4.13e+07 | 4.13e+07 |
| Ker04 | 4.69e+07 | 4.07e+07 | 4.03e+07 | 4.03e+07 |
| Ker05 | 6.02e+07 | 4.99e+07 | 4.99e+07 | 4.99e+07 |
| Ker06 | 4.28e+07 | 3.64e+07 | 3.62e+07 | 3.62e+07 |
| Ker07 | 7.95e+07 | 6.63e+07 | 4.90e+07 | 4.90e+07 |
| Ker08 | 1.20e+08 | 9.50e+07 | 7.76e+07 | 7.15e+07 |
| Ker09 | 6.74e+07 | 6.51e+07 | 5.43e+07 | 5.35e+07 |
| Ker10 | 1.00e+08 | 9.39e+07 | 7.88e+07 | 7.80e+07 |
| Ker11 | 5.00e+07 | 4.75e+07 | 4.75e+07 | 4.75e+07 |
| Ker12 | 5.31e+07 | 5.05e+07 | 5.05e+07 | 5.05e+07 |
| Ker13 | 7.90e+07 | 7.27e+07 | 6.35e+07 | 6.29e+07 |
| Ker14 | 8.16e+07 | 6.38e+07 | 6.38e+07 | 6.38e+07 |
| Amean | 6.79e+07 | 5.89e+07 | 5.36e+07 | 5.30e+07 |
| **Nasker** | | | | |
| BTR | 2.15e+09 | 1.63e+09 | 1.32e+09 | 1.26e+09 |
| CHO | 2.94e+09 | 2.66e+09 | 2.60e+09 | 2.51e+09 |
| EMI | 6.27e+08 | 5.70e+08 | 5.26e+08 | 5.13e+08 |
| FFT | 1.72e+09 | 1.48e+09 | 1.29e+09 | 1.25e+09 |
| GMT | 1.74e+09 | 1.61e+09 | 1.61e+09 | 1.61e+09 |
| MXM | 2.01e+09 | 1.52e+09 | 1.38e+09 | 1.36e+09 |
| VPE | 6.50e+07 | 4.84e+07 | 4.19e+07 | 4.24e+07 |
| Amean | 1.61e+09 | 1.36e+09 | 1.25e+09 | 1.22e+09 |
| **Optimized Nasker** | | | | |
| BTR | 1.32e+09 | 1.10e+09 | 8.84e+08 | 8.44e+08 |
| CHO | 1.08e+09 | 8.39e+08 | 8.03e+08 | 8.03e+08 |
| EMI | 5.94e+08 | 5.64e+08 | 5.13e+08 | 5.03e+08 |
| FFT | 1.49e+09 | 1.22e+09 | 1.07e+09 | 1.06e+09 |
| GMT | 1.20e+09 | 9.45e+08 | 9.02e+08 | 9.00e+08 |
| MXM | 1.23e+09 | 1.01e+09 | 7.30e+08 | 7.30e+08 |
| VPE | 6.44e+07 | 4.84e+07 | 4.19e+07 | 4.22e+07 |
| Amean | 9.97e+08 | 8.18e+08 | 7.06e+08 | 6.97e+08 |

| Prog | Register Set Size | | | |
|---|---|---|---|---|
| | 16 | 32 | 64 | 128 |
| **Perfect** | | | | |
| ARC2D | 4.19e+10 | 3.32e+10 | 2.81e+10 | 2.61e+10 |
| BDNA | 8.32e+09 | 5.87e+09 | 4.90e+09 | 4.54e+09 |
| DYFESM | 6.93e+09 | 6.04e+09 | 5.75e+09 | 5.70e+09 |
| FLO52 | 1.24e+10 | 1.05e+10 | 9.75e+09 | 9.54e+09 |
| MDG | 1.31e+10 | 1.28e+10 | 1.27e+10 | 1.36e+10 |
| MG3D | 1.49e+11 | 1.16e+11 | 1.06e+11 | 9.53e+10 |
| QCD | 3.22e+09 | 3.22e+09 | 3.31e+09 | 3.44e+09 |
| TRACK | 1.19e+09 | 1.08e+09 | 1.08e+09 | 1.12e+09 |
| Amean | 2.95e+10 | 2.36e+10 | 2.14e+10 | 1.99e+10 |
| **Misc** | | | | |
| BOAST | 6.47e+09 | 5.01e+09 | 4.58e+09 | 4.48e+09 |
| SPHOT | 8.34e+07 | 7.65e+07 | 7.56e+07 | 7.60e+07 |
| WANAL1 | 1.35e+10 | 1.07e+10 | 1.02e+10 | 1.03e+10 |
| NEURAL | 1.34e+09 | 1.10e+09 | 1.02e+09 | 1.02e+09 |
| COSTSC | 1.31e+08 | 1.24e+08 | 1.10e+08 | 1.05e+08 |
| Amean | 4.30e+09 | 3.40e+09 | 3.20e+09 | 3.20e+09 |
| **Int** | | | | |
| DHRYSTO | 1.35e+07 | 1.36e+07 | 1.36e+07 | 1.36e+07 |
| INTEGER | 1.91e+08 | 1.78e+08 | 1.66e+08 | 1.57e+08 |
| KNIGHT | 4.10e+06 | 4.08e+06 | 3.91e+06 | 3.91e+06 |
| LCC | 1.76e+07 | 2.00e+07 | 2.37e+07 | 2.74e+07 |
| SPARSE | 3.18e+06 | 3.29e+06 | 3.61e+06 | 3.63e+06 |
| Amean | 4.59e+07 | 4.38e+07 | 4.22e+07 | 4.11e+07 |

Table F.11: **Ar2shb IPS raw data** for four register set sizes.

| Prog | Register Set Size 16 | 32 | 64 | 128 |
|---|---|---|---|---|
| Livermore | | | | |
| Ker01 | 4.48e+07 | 4.48e+07 | 4.48e+07 | 4.48e+07 |
| Ker02 | 5.23e+07 | 5.23e+07 | 5.23e+07 | 5.23e+07 |
| Ker03 | 4.13e+07 | 4.13e+07 | 4.13e+07 | 4.13e+07 |
| Ker04 | 4.03e+07 | 4.03e+07 | 4.03e+07 | 4.03e+07 |
| Ker05 | 4.98e+07 | 4.98e+07 | 4.98e+07 | 4.98e+07 |
| Ker06 | 3.62e+07 | 3.62e+07 | 3.62e+07 | 3.62e+07 |
| Ker07 | 4.90e+07 | 4.90e+07 | 4.90e+07 | 4.90e+07 |
| Ker08 | 1.11e+08 | 7.02e+07 | 7.10e+07 | 7.10e+07 |
| Ker09 | 5.51e+07 | 5.43e+07 | 5.43e+07 | 5.43e+07 |
| Ker10 | 8.46e+07 | 7.80e+07 | 7.80e+07 | 7.80e+07 |
| Ker11 | 4.75e+07 | 4.75e+07 | 4.75e+07 | 4.75e+07 |
| Ker12 | 5.05e+07 | 5.05e+07 | 5.05e+07 | 5.05e+07 |
| Ker13 | 6.93e+07 | 6.29e+07 | 6.29e+07 | 6.29e+07 |
| Ker14 | 6.42e+07 | 6.42e+07 | 6.42e+07 | 6.42e+07 |
| Amean | 5.68e+07 | 5.29e+07 | 5.30e+07 | 5.30e+07 |
| Nasker | | | | |
| BTR | 1.53e+09 | 1.27e+09 | 1.26e+09 | 1.24e+09 |
| CHO | 2.54e+09 | 2.43e+09 | 2.43e+09 | 2.43e+09 |
| EMI | 6.79e+08 | 5.13e+08 | 5.13e+08 | 5.13e+08 |
| FFT | 1.40e+09 | 1.24e+09 | 1.25e+09 | 1.25e+09 |
| GMT | 1.62e+09 | 1.69e+09 | 1.61e+09 | 1.61e+09 |
| MXM | 1.71e+09 | 1.44e+09 | 1.44e+09 | 1.44e+09 |
| VPE | 4.96e+07 | 4.21e+07 | 4.19e+07 | 4.22e+07 |
| Amean | 1.36e+09 | 1.23e+09 | 1.22e+09 | 1.22e+09 |
| Optimized Nasker | | | | |
| BTR | 1.06e+09 | 8.35e+08 | 8.35e+08 | 8.32e+08 |
| CHO | 8.39e+08 | 7.79e+08 | 7.79e+08 | 7.79e+08 |
| EMI | 7.14e+08 | 5.03e+08 | 5.18e+08 | 5.08e+08 |
| FFT | 1.16e+09 | 1.08e+09 | 1.07e+09 | 1.07e+09 |
| GMT | 9.08e+08 | 9.00e+08 | 9.00e+08 | 9.00e+08 |
| MXM | 9.51e+08 | 7.30e+08 | 7.30e+08 | 7.30e+08 |
| VPE | 4.94e+07 | 4.19e+07 | 4.19e+07 | 4.19e+07 |
| Amean | 8.12e+08 | 6.96e+08 | 6.96e+08 | 6.94e+08 |

| Prog | Register Set Size 16 | 32 | 64 | 128 |
|---|---|---|---|---|
| Perfect | | | | |
| ARC2D | 3.85e+10 | 2.75e+10 | 2.54e+10 | 2.54e+10 |
| BDNA | 8.23e+09 | 6.50e+09 | 5.04e+09 | 4.50e+09 |
| DYFESM | 5.68e+09 | 5.67e+09 | 5.67e+09 | 5.67e+09 |
| FLO52 | 1.03e+10 | 9.41e+09 | 9.31e+09 | 9.30e+09 |
| MDG | 1.26e+10 | 1.20e+10 | 1.20e+10 | 1.20e+10 |
| MG3D | 1.36e+11 | 9.94e+10 | 9.38e+10 | 9.36e+10 |
| QCD | 3.09e+09 | 3.08e+09 | 3.07e+09 | 3.07e+09 |
| TRACK | 1.16e+09 | 1.06e+09 | 1.03e+09 | 1.03e+09 |
| Amean | 2.69e+10 | 2.06e+10 | 1.94e+10 | 1.93e+10 |
| Misc | | | | |
| BOAST | 5.42e+09 | 4.33e+09 | 4.29e+09 | 4.28e+09 |
| SPHOT | 7.87e+07 | 6.79e+07 | 6.83e+07 | 6.83e+07 |
| WANAL1 | 1.08e+10 | 9.45e+09 | 9.35e+09 | 9.35e+09 |
| NEURAL | 1.35e+09 | 9.18e+08 | 9.18e+08 | 9.18e+08 |
| COSTSC | 1.22e+08 | 9.33e+07 | 8.94e+07 | 8.94e+07 |
| Amean | 3.55e+09 | 2.97e+09 | 2.94e+09 | 2.94e+09 |
| Int | | | | |
| DHRYSTO | 1.28e+07 | 1.28e+07 | 1.28e+07 | 1.28e+07 |
| INTEGER | 1.57e+08 | 1.57e+08 | 1.57e+08 | 1.57e+08 |
| KNIGHT | 3.38e+06 | 3.38e+06 | 3.38e+06 | 3.38e+06 |
| LCC | 1.45e+07 | 1.45e+07 | 1.45e+07 | 1.45e+07 |
| SPARSE | 2.68e+06 | 2.62e+06 | 2.62e+06 | 2.62e+06 |
| Amean | 3.81e+07 | 3.81e+07 | 3.81e+07 | 3.81e+07 |

Table F.12: **Ar2shb RASE raw data** for four register set sizes.

| Prog | Register Set Size 16 | 32 | 64 | 128 |
|---|---|---|---|---|
| Livermore | | | | |
| Ker01 | 4.48e+07 | 4.48e+07 | 4.48e+07 | 4.48e+07 |
| Ker02 | 5.23e+07 | 5.23e+07 | 5.23e+07 | 5.23e+07 |
| Ker03 | 4.13e+07 | 4.13e+07 | 4.13e+07 | 4.13e+07 |
| Ker04 | 4.03e+07 | 4.03e+07 | 4.03e+07 | 4.03e+07 |
| Ker05 | 4.98e+07 | 4.98e+07 | 4.98e+07 | 4.98e+07 |
| Ker06 | 3.62e+07 | 3.62e+07 | 3.62e+07 | 3.62e+07 |
| Ker07 | 4.90e+07 | 4.90e+07 | 4.90e+07 | 4.90e+07 |
| Ker08 | 8.72e+07 | 6.89e+07 | 7.10e+07 | 7.10e+07 |
| Ker09 | 5.35e+07 | 5.35e+07 | 5.35e+07 | 5.35e+07 |
| Ker10 | 7.80e+07 | 7.80e+07 | 7.80e+07 | 7.80e+07 |
| Ker11 | 4.75e+07 | 4.75e+07 | 4.75e+07 | 4.75e+07 |
| Ker12 | 5.05e+07 | 5.05e+07 | 5.05e+07 | 5.05e+07 |
| Ker13 | 6.29e+07 | 6.29e+07 | 6.29e+07 | 6.29e+07 |
| Ker14 | 6.38e+07 | 6.38e+07 | 6.38e+07 | 6.38e+07 |
| Amean | 5.41e+07 | 5.28e+07 | 5.29e+07 | 5.29e+07 |
| Nasker | | | | |
| BTR | 1.49e+09 | 1.24e+09 | 1.23e+09 | 1.23e+09 |
| CHO | 2.58e+09 | 2.43e+09 | 2.43e+09 | 2.43e+09 |
| EMI | 6.80e+08 | 5.18e+08 | 5.28e+08 | 5.28e+08 |
| FFT | 1.34e+09 | 1.24e+09 | 1.25e+09 | 1.25e+09 |
| GMT | 1.61e+09 | 1.61e+09 | 1.61e+09 | 1.61e+09 |
| MXM | 1.44e+09 | 1.36e+09 | 1.36e+09 | 1.36e+09 |
| VPE | 4.49e+07 | 4.17e+07 | 4.17e+07 | 4.17e+07 |
| Amean | 1.31e+09 | 1.21e+09 | 1.21e+09 | 1.21e+09 |
| Optimized Nasker | | | | |
| BTR | 9.61e+08 | 8.34e+08 | 8.34e+08 | 8.32e+08 |
| CHO | 8.42e+08 | 7.79e+08 | 7.79e+08 | 7.79e+08 |
| EMI | 7.40e+08 | 5.03e+08 | 5.13e+08 | 5.13e+08 |
| FFT | 1.14e+09 | 1.08e+09 | 1.06e+09 | 1.06e+09 |
| GMT | 9.03e+08 | 9.01e+08 | 9.00e+08 | 9.00e+08 |
| MXM | 9.51e+08 | 7.30e+08 | 7.30e+08 | 7.30e+08 |
| VPE | 4.49e+07 | 4.17e+07 | 4.17e+07 | 4.17e+07 |
| Amean | 7.97e+08 | 6.96e+08 | 6.94e+08 | 6.94e+08 |

| Prog | Register Set Size 16 | 32 | 64 | 128 |
|---|---|---|---|---|
| Perfect | | | | |
| ARC2D | 3.73e+10 | 2.73e+10 | 2.51e+10 | 2.51e+10 |
| BDNA | 9.36e+09 | 5.87e+09 | 4.85e+09 | 4.40e+09 |
| DYFESM | 5.68e+09 | 5.67e+09 | 5.67e+09 | 5.67e+09 |
| FLO52 | 9.93e+09 | 9.35e+09 | 9.18e+09 | 9.18e+09 |
| MDG | 1.25e+10 | 1.20e+10 | 1.20e+10 | 1.20e+10 |
| MG3D | 1.16e+11 | 9.39e+10 | 9.29e+10 | 9.29e+10 |
| QCD | 2.97e+09 | 2.96e+09 | 2.95e+09 | 2.95e+09 |
| TRACK | 1.22e+09 | 1.04e+09 | 1.02e+09 | 1.02e+09 |
| Amean | 2.44e+10 | 1.98e+10 | 1.92e+10 | 1.92e+10 |
| Misc | | | | |
| BOAST | 5.44e+09 | 4.31e+09 | 4.29e+09 | 4.29e+09 |
| SPHOT | 7.21e+07 | 6.89e+07 | 6.89e+07 | 6.89e+07 |
| WANAL1 | 1.10e+10 | 9.30e+09 | 9.30e+09 | 9.30e+09 |
| NEURAL | 1.20e+09 | 9.20e+08 | 9.20e+08 | 9.20e+08 |
| COSTSC | 1.23e+08 | 9.61e+07 | 8.91e+07 | 8.91e+07 |
| Amean | 3.57e+09 | 2.94e+09 | 2.93e+09 | 2.93e+09 |
| Int | | | | |
| DHRYSTO | 1.32e+07 | 1.32e+07 | 1.32e+07 | 1.32e+07 |
| INTEGER | 1.57e+08 | 1.57e+08 | 1.57e+08 | 1.57e+08 |
| KNIGHT | 3.41e+06 | 3.41e+06 | 3.41e+06 | 3.41e+06 |
| LCC | 1.50e+07 | 1.50e+07 | 1.50e+07 | 1.50e+07 |
| SPARSE | 2.75e+06 | 2.69e+06 | 2.69e+06 | 2.69e+06 |
| Amean | 3.83e+07 | 3.83e+07 | 3.83e+07 | 3.83e+07 |

Table F.13: **Ar2sp Postpass raw data for the Livermore and Nasker groups** for six register set sizes.

| Program | Register Set Size | | | | | |
|---|---|---|---|---|---|---|
| | 16 | 32 | 32/16 | 64 | 64/32 | 128 |
| | | | Livermore | | | |
| Ker01 | 7.31e+07 | 4.63e+07 | 4.48e+07 | 4.48e+07 | 4.48e+07 | 4.48e+07 |
| Ker02 | 7.81e+07 | 5.63e+07 | 5.23e+07 | 5.23e+07 | 5.23e+07 | 5.23e+07 |
| Ker03 | 5.91e+07 | 4.14e+07 | 4.13e+07 | 4.13e+07 | 4.13e+07 | 4.13e+07 |
| Ker04 | 6.62e+07 | 4.07e+07 | 4.07e+07 | 4.07e+07 | 4.03e+07 | 4.03e+07 |
| Ker05 | 7.47e+07 | 5.20e+07 | 4.99e+07 | 4.99e+07 | 4.99e+07 | 4.99e+07 |
| Ker06 | 6.09e+07 | 3.79e+07 | 3.62e+07 | 3.62e+07 | 3.62e+07 | 3.62e+07 |
| Ker07 | 8.12e+07 | 6.80e+07 | 5.47e+07 | 5.06e+07 | 4.90e+07 | 4.90e+07 |
| Ker08 | 1.41e+08 | 1.04e+08 | 8.55e+07 | 8.28e+07 | 6.93e+07 | 7.19e+07 |
| Ker09 | 7.60e+07 | 6.59e+07 | 5.43e+07 | 5.43e+07 | 5.35e+07 | 5.35e+07 |
| Ker10 | 1.29e+08 | 9.61e+07 | 9.90e+07 | 9.90e+07 | 7.80e+07 | 7.80e+07 |
| Ker11 | 7.12e+07 | 5.00e+07 | 4.75e+07 | 4.75e+07 | 4.75e+07 | 4.75e+07 |
| Ker12 | 7.57e+07 | 5.31e+07 | 5.05e+07 | 5.05e+07 | 5.05e+07 | 5.05e+07 |
| Ker13 | 1.12e+08 | 8.14e+07 | 7.51e+07 | 7.51e+07 | 6.54e+07 | 6.54e+07 |
| Ker14 | 1.01e+08 | 8.12e+07 | 6.75e+07 | 6.75e+07 | 6.63e+07 | 6.63e+07 |
| Amean | 8.57e+07 | 6.24e+07 | 5.71e+07 | 5.66e+07 | 5.32e+07 | 5.33e+07 |
| | | | Nasker | | | |
| BTR | 2.16e+09 | 1.81e+09 | 1.45e+09 | 1.48e+09 | 1.28e+09 | 1.31e+09 |
| CHO | 3.40e+09 | 2.88e+09 | 2.72e+09 | 2.72e+09 | 2.57e+09 | 2.57e+09 |
| EMI | 7.09e+08 | 5.69e+08 | 5.59e+08 | 5.49e+08 | 5.34e+08 | 5.14e+08 |
| FFT | 1.89e+09 | 1.42e+09 | 1.30e+09 | 1.29e+09 | 1.30e+09 | 1.29e+09 |
| GMT | 2.62e+09 | 1.74e+09 | 1.61e+09 | 1.61e+09 | 1.61e+09 | 1.61e+09 |
| MXM | 2.31e+09 | 1.87e+09 | 1.52e+09 | 1.52e+09 | 1.36e+09 | 1.36e+09 |
| VPE | 6.72e+07 | 5.15e+07 | 4.42e+07 | 4.39e+07 | 4.27e+07 | 4.23e+07 |
| Amean | 1.88e+09 | 1.48e+09 | 1.31e+09 | 1.32e+09 | 1.24e+09 | 1.24e+09 |
| | | | Optimized Nasker | | | |
| BTR | 1.34e+09 | 1.23e+09 | 1.03e+09 | 9.94e+08 | 8.57e+08 | 8.63e+08 |
| CHO | 1.44e+09 | 9.95e+08 | 8.27e+08 | 8.26e+08 | 8.03e+08 | 8.04e+08 |
| EMI | 6.93e+08 | 5.36e+08 | 5.29e+08 | 5.24e+08 | 5.14e+08 | 4.98e+08 |
| FFT | 1.68e+09 | 1.22e+09 | 1.07e+09 | 1.07e+09 | 1.08e+09 | 1.07e+09 |
| GMT | 1.62e+09 | 1.07e+09 | 9.02e+08 | 9.02e+08 | 9.01e+08 | 9.01e+08 |
| MXM | 1.36e+09 | 1.03e+09 | 8.73e+08 | 8.73e+08 | 7.62e+08 | 7.62e+08 |
| VPE | 6.88e+07 | 5.15e+07 | 4.42e+07 | 4.39e+07 | 4.27e+07 | 4.10e+07 |
| Amean | 1.17e+09 | 8.76e+08 | 7.54e+08 | 7.48e+08 | 7.09e+08 | 7.06e+08 |

Table F.14: **Ar2sp Postpass raw data for the Perfect, Misc and Int groups** for six register set sizes.

| Program | Register Set Size | | | | | |
|---|---|---|---|---|---|---|
| | 16 | 32 | 32/16 | 64 | 64/32 | 128 |
| | | | Perfect | | | |
| ARC2D | 4.71e+10 | 3.38e+10 | 2.92e+10 | 2.86e+10 | 2.67e+10 | 2.64e+10 |
| BDNA | 9.31e+09 | 6.50e+09 | 6.38e+09 | 5.19e+09 | 5.20e+09 | 4.64e+09 |
| DYFESM | 8.63e+09 | 6.10e+09 | 6.04e+09 | 6.04e+09 | 5.71e+09 | 5.71e+09 |
| FLO52 | 1.42e+10 | 1.12e+10 | 1.00e+10 | 9.99e+09 | 9.57e+09 | 9.58e+09 |
| MDG | 1.56e+10 | 1.26e+10 | 1.25e+10 | 1.26e+10 | 1.25e+10 | 1.29e+10 |
| MG3D | 1.78e+11 | 1.26e+11 | 1.08e+11 | 1.04e+11 | 1.01e+11 | 9.77e+10 |
| QCD | 3.77e+09 | 3.07e+09 | 3.04e+09 | 3.07e+09 | 3.13e+09 | 3.14e+09 |
| TRACK | 1.23e+09 | 1.10e+09 | 1.09e+09 | 1.08e+09 | 1.10e+09 | 1.11e+09 |
| Amean | 3.47e+10 | 2.50e+10 | 2.20e+10 | 2.13e+10 | 2.06e+10 | 2.01e+10 |
| | | | Misc | | | |
| BOAST | 6.61e+09 | 5.22e+09 | 4.78e+09 | 4.67e+09 | 4.54e+09 | 4.50e+09 |
| SPHOT | 9.50e+07 | 7.94e+07 | 8.07e+07 | 7.73e+07 | 7.72e+07 | 7.51e+07 |
| WANAL1 | 1.60e+10 | 1.24e+10 | 1.06e+10 | 1.02e+10 | 1.02e+10 | 1.03e+10 |
| NEURAL | 1.38e+09 | 1.15e+09 | 1.03e+09 | 1.01e+09 | 1.02e+09 | 1.03e+09 |
| COSTSC | 1.36e+08 | 1.25e+08 | 1.18e+08 | 1.13e+08 | 1.10e+08 | 1.07e+08 |
| Amean | 4.84e+09 | 3.79e+09 | 3.32e+09 | 3.21e+09 | 3.19e+09 | 3.20e+09 |
| | | | Int | | | |
| DHRYSTO | 1.42e+07 | 1.35e+07 | 1.36e+07 | 1.36e+07 | 1.36e+07 | 1.36e+07 |
| INTEGER | 1.96e+08 | 1.91e+08 | 1.78e+08 | 1.78e+08 | 1.66e+08 | 1.66e+08 |
| KNIGHT | 4.58e+06 | 4.10e+06 | 4.08e+06 | 4.08e+06 | 3.91e+06 | 3.91e+06 |
| LCC | 1.69e+07 | 1.77e+07 | 2.01e+07 | 2.01e+07 | 2.38e+07 | 2.39e+07 |
| SPARSE | 3.30e+06 | 3.16e+06 | 3.31e+06 | 3.34e+06 | 3.58e+06 | 3.59e+06 |
| Amean | 4.70e+07 | 4.59e+07 | 4.38e+07 | 4.38e+07 | 4.22e+07 | 4.22e+07 |

Table F.15: **Ar2sp IPS raw data for the Livermore and Nasker groups** for six register set sizes.

| Program | Register Set Size | | | | | |
|---|---|---|---|---|---|---|
| | 16 | 32 | 32/16 | 64 | 64/32 | 128 |
| Livermore | | | | | | |
| Ker01 | 5.53e+07 | 4.48e+07 | 4.48e+07 | 4.48e+07 | 4.48e+07 | 4.48e+07 |
| Ker02 | 6.11e+07 | 5.23e+07 | 5.23e+07 | 5.23e+07 | 5.23e+07 | 5.23e+07 |
| Ker03 | 4.74e+07 | 4.13e+07 | 4.13e+07 | 4.13e+07 | 4.13e+07 | 4.13e+07 |
| Ker04 | 6.14e+07 | 4.03e+07 | 4.03e+07 | 4.03e+07 | 4.03e+07 | 4.03e+07 |
| Ker05 | 5.83e+07 | 4.98e+07 | 4.98e+07 | 4.98e+07 | 4.98e+07 | 4.98e+07 |
| Ker06 | 4.85e+07 | 3.62e+07 | 3.62e+07 | 3.62e+07 | 3.62e+07 | 3.62e+07 |
| Ker07 | 5.73e+07 | 4.90e+07 | 4.90e+07 | 4.90e+07 | 4.90e+07 | 4.90e+07 |
| Ker08 | 1.18e+08 | 7.59e+07 | 7.10e+07 | 7.10e+07 | 6.84e+07 | 7.10e+07 |
| Ker09 | 6.60e+07 | 5.35e+07 | 5.35e+07 | 5.35e+07 | 5.35e+07 | 5.35e+07 |
| Ker10 | 1.13e+08 | 8.74e+07 | 7.80e+07 | 7.80e+07 | 7.80e+07 | 7.80e+07 |
| Ker11 | 5.72e+07 | 4.75e+07 | 4.75e+07 | 4.75e+07 | 4.75e+07 | 4.75e+07 |
| Ker12 | 6.08e+07 | 5.05e+07 | 5.05e+07 | 5.05e+07 | 5.05e+07 | 5.05e+07 |
| Ker13 | 1.08e+08 | 6.64e+07 | 6.48e+07 | 6.48e+07 | 6.48e+07 | 6.48e+07 |
| Ker14 | 8.33e+07 | 6.63e+07 | 6.63e+07 | 6.63e+07 | 6.63e+07 | 6.63e+07 |
| Amean | 7.11e+07 | 5.44e+07 | 5.32e+07 | 5.32e+07 | 5.30e+07 | 5.32e+07 |
| Nasker | | | | | | |
| BTR | 1.81e+09 | 1.40e+09 | 1.24e+09 | 1.24e+09 | 1.24e+09 | 1.25e+09 |
| CHO | 2.97e+09 | 2.54e+09 | 2.44e+09 | 2.44e+09 | 2.44e+09 | 2.44e+09 |
| EMI | 7.14e+08 | 5.14e+08 | 5.23e+08 | 5.13e+08 | 5.13e+08 | 5.13e+08 |
| FFT | 1.66e+09 | 1.35e+09 | 1.25e+09 | 1.24e+09 | 1.25e+09 | 1.25e+09 |
| GMT | 1.70e+09 | 1.61e+09 | 1.61e+09 | 1.61e+09 | 1.61e+09 | 1.61e+09 |
| MXM | 1.93e+09 | 1.70e+09 | 1.41e+09 | 1.41e+09 | 1.41e+09 | 1.41e+09 |
| VPE | 5.83e+07 | 4.26e+07 | 4.13e+07 | 4.13e+07 | 4.24e+07 | 4.13e+07 |
| Amean | 1.55e+09 | 1.31e+09 | 1.22e+09 | 1.21e+09 | 1.22e+09 | 1.22e+09 |
| Optimized Nasker | | | | | | |
| BTR | 1.16e+09 | 8.91e+08 | 8.33e+08 | 8.44e+08 | 8.34e+08 | 8.33e+08 |
| CHO | 1.18e+09 | 8.46e+08 | 7.79e+08 | 7.79e+08 | 7.79e+08 | 7.79e+08 |
| EMI | 6.18e+08 | 5.04e+08 | 5.03e+08 | 5.03e+08 | 5.03e+08 | 5.03e+08 |
| FFT | 1.42e+09 | 1.10e+09 | 1.06e+09 | 1.06e+09 | 1.08e+09 | 1.06e+09 |
| GMT | 1.29e+09 | 9.03e+08 | 9.00e+08 | 9.00e+08 | 9.00e+08 | 9.00e+08 |
| MXM | 1.22e+09 | 8.10e+08 | 7.46e+08 | 7.46e+08 | 7.46e+08 | 7.46e+08 |
| VPE | 5.98e+07 | 4.26e+07 | 4.13e+07 | 4.13e+07 | 4.19e+07 | 4.13e+07 |
| Amean | 9.93e+08 | 7.28e+08 | 6.95e+08 | 6.96e+08 | 6.98e+08 | 6.95e+08 |

Table F.16: **Ar2sp IPS raw data for the Perfect, Misc and Int groups** for six register set sizes.

| Program | Register Set Size | | | | | |
|---|---|---|---|---|---|---|
| | 16 | 32 | 32/16 | 64 | 64/32 | 128 |
| Perfect | | | | | | |
| ARC2D | 4.45e+10 | 2.99e+10 | 2.63e+10 | 2.54e+10 | 2.54e+10 | 2.53e+10 |
| BDNA | 9.69e+09 | 6.65e+09 | 6.31e+09 | 5.38e+09 | 5.20e+09 | 4.66e+09 |
| DYFESM | 8.29e+09 | 5.67e+09 | 5.67e+09 | 5.67e+09 | 5.67e+09 | 5.67e+09 |
| FLO52 | 1.27e+10 | 9.80e+09 | 9.32e+09 | 9.32e+09 | 9.31e+09 | 9.29e+09 |
| MDG | 1.47e+10 | 1.21e+10 | 1.20e+10 | 1.20e+10 | 1.21e+10 | 1.21e+10 |
| MG3D | 1.64e+11 | 1.05e+11 | 9.92e+10 | 9.40e+10 | 9.38e+10 | 9.41e+10 |
| QCD | 3.47e+09 | 2.95e+09 | 2.94e+09 | 2.93e+09 | 2.94e+09 | 2.92e+09 |
| TRACK | 1.18e+09 | 1.06e+09 | 1.06e+09 | 1.03e+09 | 1.04e+09 | 1.03e+09 |
| Amean | 3.23e+10 | 2.16e+10 | 2.04e+10 | 1.95e+10 | 1.94e+10 | 1.94e+10 |
| Misc | | | | | | |
| BOAST | 6.11e+09 | 4.47e+09 | 4.36e+09 | 4.30e+09 | 4.29e+09 | 4.29e+09 |
| SPHOT | 9.02e+07 | 7.31e+07 | 7.31e+07 | 6.89e+07 | 6.89e+07 | 6.89e+07 |
| WANAL1 | 1.21e+10 | 9.44e+09 | 9.44e+09 | 9.37e+09 | 9.37e+09 | 9.37e+09 |
| NEURAL | 1.30e+09 | 1.00e+09 | 8.91e+08 | 9.04e+08 | 9.04e+08 | 9.04e+08 |
| COSTSC | 1.29e+08 | 9.24e+07 | 8.83e+07 | 8.23e+07 | 8.23e+07 | 8.23e+07 |
| Amean | 3.95e+09 | 3.02e+09 | 2.97e+09 | 2.95e+09 | 2.94e+09 | 2.94e+09 |
| Int | | | | | | |
| DHRYSTO | 1.34e+07 | 1.28e+07 | 1.28e+07 | 1.28e+07 | 1.28e+07 | 1.28e+07 |
| INTEGER | 1.58e+08 | 1.57e+08 | 1.57e+08 | 1.57e+08 | 1.57e+08 | 1.57e+08 |
| KNIGHT | 4.14e+06 | 3.38e+06 | 3.38e+06 | 3.38e+06 | 3.38e+06 | 3.38e+06 |
| LCC | 1.46e+07 | 1.44e+07 | 1.44e+07 | 1.44e+07 | 1.44e+07 | 1.44e+07 |
| SPARSE | 2.84e+06 | 2.60e+06 | 2.59e+06 | 2.60e+06 | 2.60e+06 | 2.60e+06 |
| Amean | 3.86e+07 | 3.80e+07 | 3.80e+07 | 3.80e+07 | 3.80e+07 | 3.80e+07 |

Table F.17: **Ar2sp RASE raw data for the Livermore and Nasker groups** for six register set sizes.

| Program | Register Set Size | | | | | |
|---|---|---|---|---|---|---|
| | 16 | 32 | 32/16 | 64 | 64/32 | 128 |
| Livermore | | | | | | |
| Ker01 | 5.38e+07 | 4.48e+07 | 4.48e+07 | 4.48e+07 | 4.48e+07 | 4.48e+07 |
| Ker02 | 5.97e+07 | 5.23e+07 | 5.23e+07 | 5.23e+07 | 5.23e+07 | 5.23e+07 |
| Ker03 | 4.74e+07 | 4.13e+07 | 4.13e+07 | 4.13e+07 | 4.13e+07 | 4.13e+07 |
| Ker04 | 5.10e+07 | 4.03e+07 | 4.03e+07 | 4.03e+07 | 4.03e+07 | 4.03e+07 |
| Ker05 | 5.62e+07 | 4.98e+07 | 4.98e+07 | 4.98e+07 | 4.98e+07 | 4.98e+07 |
| Ker06 | 4.23e+07 | 3.62e+07 | 3.62e+07 | 3.62e+07 | 3.62e+07 | 3.62e+07 |
| Ker07 | 5.48e+07 | 4.90e+07 | 4.90e+07 | 4.90e+07 | 4.90e+07 | 4.90e+07 |
| Ker08 | 1.17e+08 | 7.15e+07 | 7.10e+07 | 7.10e+07 | 6.89e+07 | 7.10e+07 |
| Ker09 | 6.36e+07 | 5.35e+07 | 5.35e+07 | 5.35e+07 | 5.35e+07 | 5.35e+07 |
| Ker10 | 1.00e+08 | 7.80e+07 | 7.80e+07 | 7.80e+07 | 7.80e+07 | 7.80e+07 |
| Ker11 | 5.48e+07 | 4.75e+07 | 4.75e+07 | 4.75e+07 | 4.75e+07 | 4.75e+07 |
| Ker12 | 5.82e+07 | 5.05e+07 | 5.05e+07 | 5.05e+07 | 5.05e+07 | 5.05e+07 |
| Ker13 | 7.86e+07 | 6.49e+07 | 6.48e+07 | 6.48e+07 | 6.48e+07 | 6.48e+07 |
| Ker14 | 7.80e+07 | 6.63e+07 | 6.63e+07 | 6.63e+07 | 6.63e+07 | 6.63e+07 |
| Amean | 6.54e+07 | 5.33e+07 | 5.32e+07 | 5.32e+07 | 5.31e+07 | 5.32e+07 |
| Nasker | | | | | | |
| BTR | 2.07e+09 | 1.45e+09 | 1.27e+09 | 1.23e+09 | 1.24e+09 | 1.24e+09 |
| CHO | 2.65e+09 | 2.57e+09 | 2.43e+09 | 2.43e+09 | 2.43e+09 | 2.43e+09 |
| EMI | 8.82e+08 | 5.39e+08 | 5.28e+08 | 5.18e+08 | 5.18e+08 | 5.28e+08 |
| FFT | 1.55e+09 | 1.33e+09 | 1.24e+09 | 1.24e+09 | 1.25e+09 | 1.25e+09 |
| GMT | 1.92e+09 | 1.61e+09 | 1.61e+09 | 1.61e+09 | 1.61e+09 | 1.61e+09 |
| MXM | 2.20e+09 | 1.41e+09 | 1.36e+09 | 1.36e+09 | 1.36e+09 | 1.36e+09 |
| VPE | 5.11e+07 | 4.22e+07 | 4.13e+07 | 4.13e+07 | 4.22e+07 | 4.10e+07 |
| Amean | 1.62e+09 | 1.28e+09 | 1.21e+09 | 1.20e+09 | 1.21e+09 | 1.21e+09 |
| Optimized Nasker | | | | | | |
| BTR | 1.24e+09 | 8.69e+08 | 8.40e+08 | 8.37e+08 | 8.33e+08 | 8.34e+08 |
| CHO | 1.08e+09 | 8.39e+08 | 7.79e+08 | 7.79e+08 | 7.79e+08 | 7.79e+08 |
| EMI | 7.87e+08 | 5.24e+08 | 5.13e+08 | 5.03e+08 | 5.03e+08 | 5.13e+08 |
| FFT | 1.41e+09 | 1.12e+09 | 1.06e+09 | 1.06e+09 | 1.08e+09 | 1.06e+09 |
| GMT | 1.11e+09 | 9.00e+08 | 9.00e+08 | 9.00e+08 | 9.01e+08 | 9.01e+08 |
| MXM | 1.20e+09 | 7.94e+08 | 7.30e+08 | 7.30e+08 | 7.30e+08 | 7.30e+08 |
| VPE | 5.05e+07 | 4.22e+07 | 4.13e+07 | 4.13e+07 | 4.22e+07 | 4.10e+07 |
| Amean | 9.83e+08 | 7.27e+08 | 6.95e+08 | 6.93e+08 | 6.95e+08 | 6.94e+08 |

Table F.18: **Ar2sp RASE raw data for the Perfect, Misc and Int groups** for six register set sizes.

| Program | Register Set Size | | | | | |
|---|---|---|---|---|---|---|
| | 16 | 32 | 32/16 | 64 | 64/32 | 128 |
| Perfect | | | | | | |
| ARC2D | 4.55e+10 | 2.65e+10 | 2.59e+10 | 2.51e+10 | 2.51e+10 | 2.51e+10 |
| BDNA | 1.08e+10 | 6.00e+09 | 5.60e+09 | 5.05e+09 | 4.66e+09 | 4.49e+09 |
| DYFESM | 6.39e+09 | 5.67e+09 | 5.67e+09 | 5.67e+09 | 5.67e+09 | 5.67e+09 |
| FLO52 | 1.14e+10 | 9.57e+09 | 9.19e+09 | 9.19e+09 | 9.18e+09 | 9.18e+09 |
| MDG | 1.49e+10 | 1.21e+10 | 1.21e+10 | 1.21e+10 | 1.21e+10 | 1.21e+10 |
| MG3D | 1.54e+11 | 9.90e+10 | 9.43e+10 | 9.30e+10 | 9.30e+10 | 9.30e+10 |
| QCD | 3.40e+09 | 2.99e+09 | 2.99e+09 | 2.99e+09 | 2.97e+09 | 2.97e+09 |
| TRACK | 1.20e+09 | 1.06e+09 | 1.05e+09 | 1.02e+09 | 1.02e+09 | 1.02e+09 |
| Amean | 3.09e+10 | 2.04e+10 | 1.96e+10 | 1.93e+10 | 1.92e+10 | 1.92e+10 |
| Misc | | | | | | |
| BOAST | 6.20e+09 | 4.40e+09 | 4.30e+09 | 4.29e+09 | 4.28e+09 | 4.28e+09 |
| SPHOT | 8.52e+07 | 6.94e+07 | 6.94e+07 | 6.99e+07 | 6.99e+07 | 6.99e+07 |
| WANAL1 | 1.04e+10 | 9.30e+09 | 9.30e+09 | 9.30e+09 | 9.30e+09 | 9.30e+09 |
| NEURAL | 1.28e+09 | 9.36e+08 | 9.01e+08 | 9.01e+08 | 9.01e+08 | 9.01e+08 |
| COSTSC | 1.31e+08 | 9.36e+07 | 9.25e+07 | 8.43e+07 | 8.43e+07 | 8.43e+07 |
| Amean | 3.62e+09 | 2.96e+09 | 2.93e+09 | 2.93e+09 | 2.93e+09 | 2.93e+09 |
| Int | | | | | | |
| DHRYSTO | 1.41e+07 | 1.33e+07 | 1.33e+07 | 1.33e+07 | 1.33e+07 | 1.33e+07 |
| INTEGER | 1.57e+08 | 1.57e+08 | 1.57e+08 | 1.57e+08 | 1.57e+08 | 1.57e+08 |
| KNIGHT | 4.42e+06 | 3.45e+06 | 3.45e+06 | 3.45e+06 | 3.45e+06 | 3.45e+06 |
| LCC | 1.68e+07 | 1.56e+07 | 1.56e+07 | 1.56e+07 | 1.56e+07 | 1.56e+07 |
| SPARSE | 3.08e+06 | 2.72e+06 | 2.71e+06 | 2.71e+06 | 2.71e+06 | 2.71e+06 |
| Amean | 3.91e+07 | 3.84e+07 | 3.84e+07 | 3.84e+07 | 3.84e+07 | 3.84e+07 |

Table F.19: **Ar2spf Postpass raw data for the Livermore and Nasker groups** for six register set sizes.

| Program | Register Set Size | | | | | |
|---|---|---|---|---|---|---|
| | 16 | 32 | 32/16 | 64 | 64/32 | 128 |
| Livermore | | | | | | |
| Ker01 | 7.31e+07 | 4.63e+07 | 4.48e+07 | 4.48e+07 | 4.48e+07 | 4.48e+07 |
| Ker02 | 7.61e+07 | 5.45e+07 | 5.05e+07 | 5.05e+07 | 5.05e+07 | 5.05e+07 |
| Ker03 | 5.91e+07 | 4.14e+07 | 4.13e+07 | 4.13e+07 | 4.13e+07 | 4.13e+07 |
| Ker04 | 6.62e+07 | 4.07e+07 | 4.07e+07 | 4.07e+07 | 4.03e+07 | 4.03e+07 |
| Ker05 | 7.47e+07 | 5.20e+07 | 4.99e+07 | 4.99e+07 | 4.99e+07 | 4.99e+07 |
| Ker06 | 6.09e+07 | 3.79e+07 | 3.62e+07 | 3.62e+07 | 3.62e+07 | 3.62e+07 |
| Ker07 | 8.12e+07 | 6.80e+07 | 5.47e+07 | 5.06e+07 | 4.90e+07 | 4.90e+07 |
| Ker08 | 1.33e+08 | 1.08e+08 | 9.29e+07 | 9.11e+07 | 8.20e+07 | 8.02e+07 |
| Ker09 | 7.75e+07 | 6.82e+07 | 5.74e+07 | 5.74e+07 | 5.66e+07 | 5.66e+07 |
| Ker10 | 1.27e+08 | 9.68e+07 | 9.68e+07 | 9.68e+07 | 7.87e+07 | 8.09e+07 |
| Ker11 | 7.12e+07 | 5.00e+07 | 4.75e+07 | 4.75e+07 | 4.75e+07 | 4.75e+07 |
| Ker12 | 7.57e+07 | 5.31e+07 | 5.05e+07 | 5.05e+07 | 5.05e+07 | 5.05e+07 |
| Ker13 | 1.12e+08 | 8.14e+07 | 7.51e+07 | 7.51e+07 | 6.54e+07 | 6.54e+07 |
| Ker14 | 1.01e+08 | 8.12e+07 | 6.75e+07 | 6.75e+07 | 6.63e+07 | 6.63e+07 |
| Amean | 8.49e+07 | 6.28e+07 | 5.76e+07 | 5.71e+07 | 5.42e+07 | 5.42e+07 |
| Nasker | | | | | | |
| BTR | 2.02e+09 | 1.72e+09 | 1.47e+09 | 1.44e+09 | 1.21e+09 | 1.19e+09 |
| CHO | 2.68e+09 | 1.99e+09 | 1.72e+09 | 1.71e+09 | 1.65e+09 | 1.65e+09 |
| EMI | 7.25e+08 | 6.01e+08 | 5.92e+08 | 5.39e+08 | 5.31e+08 | 5.16e+08 |
| FFT | 1.86e+09 | 1.32e+09 | 1.22e+09 | 1.23e+09 | 1.23e+09 | 1.23e+09 |
| GMT | 2.41e+09 | 1.41e+09 | 1.32e+09 | 1.28e+09 | 1.28e+09 | 1.28e+09 |
| MXM | 2.03e+09 | 1.69e+09 | 1.44e+09 | 1.27e+09 | 1.16e+09 | 1.16e+09 |
| VPE | 6.72e+07 | 5.15e+07 | 4.42e+07 | 4.39e+07 | 4.24e+07 | 4.10e+07 |
| Amean | 1.68e+09 | 1.25e+09 | 1.12e+09 | 1.07e+09 | 1.01e+09 | 1.01e+09 |
| Optimized Nasker | | | | | | |
| BTR | 1.36e+09 | 1.23e+09 | 1.01e+09 | 1.01e+09 | 8.67e+08 | 8.57e+08 |
| CHO | 1.44e+09 | 9.96e+08 | 8.27e+08 | 8.26e+08 | 8.02e+08 | 8.03e+08 |
| EMI | 7.10e+08 | 5.56e+08 | 5.29e+08 | 5.19e+08 | 5.08e+08 | 4.98e+08 |
| FFT | 1.68e+09 | 1.21e+09 | 1.07e+09 | 1.07e+09 | 1.08e+09 | 1.07e+09 |
| GMT | 1.49e+09 | 1.11e+09 | 9.02e+08 | 9.01e+08 | 9.01e+08 | 9.01e+08 |
| MXM | 1.36e+09 | 1.03e+09 | 8.73e+08 | 8.73e+08 | 7.62e+08 | 7.62e+08 |
| VPE | 6.88e+07 | 5.15e+07 | 4.42e+07 | 4.36e+07 | 4.21e+07 | 4.10e+07 |
| Amean | 1.16e+09 | 8.83e+08 | 7.51e+08 | 7.49e+08 | 7.09e+08 | 7.05e+08 |

Table F.20: **Ar2spf Postpass raw data for the Perfect, Misc and Int groups** for six register set sizes.

| Program | Register Set Size | | | | | |
|---|---|---|---|---|---|---|
| | 16 | 32 | 32/16 | 64 | 64/32 | 128 |
| | | | Perfect | | | |
| ARC2D | 4.29e+10 | 3.12e+10 | 2.69e+10 | 2.51e+10 | 2.39e+10 | 2.32e+10 |
| BDNA | 9.30e+09 | 6.48e+09 | 6.33e+09 | 5.25e+09 | 5.27e+09 | 4.80e+09 |
| DYFESM | 8.87e+09 | 5.24e+09 | 5.11e+09 | 5.11e+09 | 4.87e+09 | 4.87e+09 |
| FLO52 | 1.33e+10 | 1.02e+10 | 9.01e+09 | 8.88e+09 | 8.55e+09 | 8.54e+09 |
| MDG | 1.56e+10 | 1.26e+10 | 1.26e+10 | 1.25e+10 | 1.25e+10 | 1.29e+10 |
| MG3D | 1.70e+11 | 1.16e+11 | 1.00e+11 | 9.65e+10 | 9.39e+10 | 9.12e+10 |
| QCD | 3.53e+09 | 2.77e+09 | 2.79e+09 | 2.82e+09 | 2.86e+09 | 2.88e+09 |
| TRACK | 1.22e+09 | 1.09e+09 | 1.09e+09 | 1.07e+09 | 1.09e+09 | 1.10e+09 |
| Amean | 3.31e+10 | 2.32e+10 | 2.05e+10 | 1.97e+10 | 1.91e+10 | 1.87e+10 |
| | | | Misc | | | |
| BOAST | 6.67e+09 | 5.29e+09 | 4.94e+09 | 4.78e+09 | 4.62e+09 | 4.56e+09 |
| SPHOT | 9.45e+07 | 7.74e+07 | 7.90e+07 | 7.50e+07 | 7.45e+07 | 7.37e+07 |
| WANAL1 | 1.52e+10 | 1.15e+10 | 9.44e+09 | 8.93e+09 | 8.74e+09 | 8.64e+09 |
| NEURAL | 1.40e+09 | 1.24e+09 | 1.08e+09 | 1.07e+09 | 1.07e+09 | 1.10e+09 |
| COSTSC | 1.36e+08 | 1.25e+08 | 1.18e+08 | 1.13e+08 | 1.10e+08 | 1.07e+08 |
| Amean | 4.70e+09 | 3.65e+09 | 3.13e+09 | 2.99e+09 | 2.92e+09 | 2.90e+09 |
| | | | Int | | | |
| DHRYSTO | 1.39e+07 | 1.32e+07 | 1.34e+07 | 1.34e+07 | 1.34e+07 | 1.34e+07 |
| INTEGER | 1.42e+08 | 1.38e+08 | 1.28e+08 | 1.28e+08 | 1.24e+08 | 1.24e+08 |
| KNIGHT | 4.56e+06 | 4.08e+06 | 3.97e+06 | 3.97e+06 | 3.86e+06 | 3.86e+06 |
| LCC | 1.64e+07 | 1.71e+07 | 1.96e+07 | 1.96e+07 | 2.33e+07 | 2.34e+07 |
| SPARSE | 3.29e+06 | 3.14e+06 | 3.30e+06 | 3.32e+06 | 3.56e+06 | 3.58e+06 |
| Amean | 3.60e+07 | 3.51e+07 | 3.37e+07 | 3.37e+07 | 3.36e+07 | 3.36e+07 |

Table F.21: **Ar2spf IPS raw data for the Livermore and Nasker groups** for six register set sizes.

| Program | Register Set Size 16 | 32 | 32/16 | 64 | 64/32 | 128 |
|---|---|---|---|---|---|---|
| Livermore | | | | | | |
| Ker01 | 5.53e+07 | 4.48e+07 | 4.48e+07 | 4.48e+07 | 4.48e+07 | 4.48e+07 |
| Ker02 | 5.77e+07 | 5.05e+07 | 5.05e+07 | 5.05e+07 | 5.05e+07 | 5.05e+07 |
| Ker03 | 4.74e+07 | 4.13e+07 | 4.13e+07 | 4.13e+07 | 4.13e+07 | 4.13e+07 |
| Ker04 | 6.14e+07 | 4.03e+07 | 4.03e+07 | 4.03e+07 | 4.03e+07 | 4.03e+07 |
| Ker05 | 5.83e+07 | 4.98e+07 | 4.98e+07 | 4.98e+07 | 4.98e+07 | 4.98e+07 |
| Ker06 | 4.85e+07 | 3.62e+07 | 3.62e+07 | 3.62e+07 | 3.62e+07 | 3.62e+07 |
| Ker07 | 5.73e+07 | 4.90e+07 | 4.90e+07 | 4.90e+07 | 4.90e+07 | 4.90e+07 |
| Ker08 | 1.18e+08 | 8.41e+07 | 7.98e+07 | 7.98e+07 | 8.06e+07 | 7.98e+07 |
| Ker09 | 6.68e+07 | 5.66e+07 | 5.66e+07 | 5.66e+07 | 5.66e+07 | 5.66e+07 |
| Ker10 | 1.16e+08 | 9.03e+07 | 8.09e+07 | 8.09e+07 | 7.87e+07 | 8.09e+07 |
| Ker11 | 5.72e+07 | 4.75e+07 | 4.75e+07 | 4.75e+07 | 4.75e+07 | 4.75e+07 |
| Ker12 | 6.08e+07 | 5.05e+07 | 5.05e+07 | 5.05e+07 | 5.05e+07 | 5.05e+07 |
| Ker13 | 9.84e+07 | 6.64e+07 | 6.48e+07 | 6.48e+07 | 6.48e+07 | 6.48e+07 |
| Ker14 | 8.33e+07 | 6.63e+07 | 6.63e+07 | 6.63e+07 | 6.63e+07 | 6.63e+07 |
| Amean | 7.05e+07 | 5.53e+07 | 5.42e+07 | 5.42e+07 | 5.41e+07 | 5.42e+07 |
| Nasker | | | | | | |
| BTR | 1.58e+09 | 1.23e+09 | 1.19e+09 | 1.20e+09 | 1.19e+09 | 1.19e+09 |
| CHO | 1.83e+09 | 1.58e+09 | 1.53e+09 | 1.53e+09 | 1.53e+09 | 1.53e+09 |
| EMI | 7.30e+08 | 5.13e+08 | 5.23e+08 | 5.13e+08 | 5.13e+08 | 5.13e+08 |
| FFT | 1.52e+09 | 1.21e+09 | 1.19e+09 | 1.19e+09 | 1.20e+09 | 1.20e+09 |
| GMT | 1.79e+09 | 1.28e+09 | 1.28e+09 | 1.28e+09 | 1.28e+09 | 1.28e+09 |
| MXM | 1.68e+09 | 1.21e+09 | 1.14e+09 | 1.14e+09 | 1.14e+09 | 1.14e+09 |
| VPE | 5.92e+07 | 4.26e+07 | 4.13e+07 | 4.13e+07 | 4.24e+07 | 4.13e+07 |
| Amean | 1.31e+09 | 1.01e+09 | 9.85e+08 | 9.85e+08 | 9.85e+08 | 9.85e+08 |
| Optimized Nasker | | | | | | |
| BTR | 1.17e+09 | 8.98e+08 | 8.42e+08 | 8.53e+08 | 8.44e+08 | 8.42e+08 |
| CHO | 1.16e+09 | 8.46e+08 | 7.78e+08 | 7.78e+08 | 7.78e+08 | 7.78e+08 |
| EMI | 6.16e+08 | 5.03e+08 | 5.03e+08 | 5.03e+08 | 5.03e+08 | 5.03e+08 |
| FFT | 1.41e+09 | 1.13e+09 | 1.06e+09 | 1.06e+09 | 1.08e+09 | 1.06e+09 |
| GMT | 1.28e+09 | 9.03e+08 | 9.00e+08 | 9.00e+08 | 9.01e+08 | 9.01e+08 |
| MXM | 1.22e+09 | 8.10e+08 | 7.46e+08 | 7.46e+08 | 7.46e+08 | 7.46e+08 |
| VPE | 5.87e+07 | 4.26e+07 | 4.13e+07 | 4.13e+07 | 4.19e+07 | 4.13e+07 |
| Amean | 9.88e+08 | 7.33e+08 | 6.96e+08 | 6.97e+08 | 6.99e+08 | 6.96e+08 |

Table F.22: **Ar2spf IPS raw data for the Perfect, Misc and Int groups** for six register set sizes.

| Program | Register Set Size | | | | | |
|---|---|---|---|---|---|---|
| | 16 | 32 | 32/16 | 64 | 64/32 | 128 |
| Perfect | | | | | | |
| ARC2D | 3.69e+10 | 2.49e+10 | 2.31e+10 | 2.21e+10 | 2.21e+10 | 2.21e+10 |
| BDNA | 9.56e+09 | 6.65e+09 | 6.43e+09 | 5.43e+09 | 5.08e+09 | 4.70e+09 |
| DYFESM | 6.05e+09 | 4.62e+09 | 4.62e+09 | 4.62e+09 | 4.62e+09 | 4.62e+09 |
| FLO52 | 1.00e+10 | 8.60e+09 | 8.39e+09 | 8.40e+09 | 8.40e+09 | 8.40e+09 |
| MDG | 1.45e+10 | 1.21e+10 | 1.20e+10 | 1.20e+10 | 1.21e+10 | 1.21e+10 |
| MG3D | 1.50e+11 | 9.52e+10 | 9.12e+10 | 8.59e+10 | 8.59e+10 | 8.59e+10 |
| QCD | 3.13e+09 | 2.66e+09 | 2.66e+09 | 2.65e+09 | 2.66e+09 | 2.65e+09 |
| TRACK | 1.17e+09 | 1.06e+09 | 1.06e+09 | 1.03e+09 | 1.04e+09 | 1.02e+09 |
| Amean | 2.89e+10 | 1.95e+10 | 1.87e+10 | 1.78e+10 | 1.77e+10 | 1.77e+10 |
| Misc | | | | | | |
| BOAST | 5.96e+09 | 4.51e+09 | 4.44e+09 | 4.35e+09 | 4.35e+09 | 4.35e+09 |
| SPHOT | 8.60e+07 | 6.62e+07 | 6.61e+07 | 6.73e+07 | 6.73e+07 | 6.73e+07 |
| WANAL1 | 1.23e+10 | 8.62e+09 | 8.71e+09 | 8.64e+09 | 8.64e+09 | 8.64e+09 |
| NEURAL | 1.15e+09 | 9.83e+08 | 9.44e+08 | 9.44e+08 | 9.44e+08 | 9.44e+08 |
| COSTSC | 1.28e+08 | 9.24e+07 | 8.83e+07 | 8.23e+07 | 8.23e+07 | 8.23e+07 |
| Amean | 3.92e+09 | 2.85e+09 | 2.85e+09 | 2.82e+09 | 2.82e+09 | 2.82e+09 |
| Int | | | | | | |
| DHRYSTO | 1.30e+07 | 1.25e+07 | 1.25e+07 | 1.25e+07 | 1.25e+07 | 1.25e+07 |
| INTEGER | 1.19e+08 | 1.19e+08 | 1.19e+08 | 1.19e+08 | 1.19e+08 | 1.19e+08 |
| KNIGHT | 4.23e+06 | 3.28e+06 | 3.28e+06 | 3.28e+06 | 3.28e+06 | 3.28e+06 |
| LCC | 1.41e+07 | 1.39e+07 | 1.39e+07 | 1.39e+07 | 1.39e+07 | 1.39e+07 |
| SPARSE | 2.82e+06 | 2.58e+06 | 2.58e+06 | 2.58e+06 | 2.58e+06 | 2.58e+06 |
| Amean | 3.06e+07 | 3.03e+07 | 3.03e+07 | 3.03e+07 | 3.03e+07 | 3.03e+07 |

Table F.23: **Ar2spf RASE raw data for the Livermore and Nasker groups** for six register set sizes.

| Program | Register Set Size | | | | | |
|---|---|---|---|---|---|---|
| | 16 | 32 | 32/16 | 64 | 64/32 | 128 |
| Livermore | | | | | | |
| Ker01 | 5.38e+07 | 4.48e+07 | 4.48e+07 | 4.48e+07 | 4.48e+07 | 4.48e+07 |
| Ker02 | 5.62e+07 | 5.05e+07 | 5.05e+07 | 5.05e+07 | 5.05e+07 | 5.05e+07 |
| Ker03 | 4.74e+07 | 4.13e+07 | 4.13e+07 | 4.13e+07 | 4.13e+07 | 4.13e+07 |
| Ker04 | 5.10e+07 | 4.03e+07 | 4.03e+07 | 4.03e+07 | 4.03e+07 | 4.03e+07 |
| Ker05 | 5.62e+07 | 4.98e+07 | 4.98e+07 | 4.98e+07 | 4.98e+07 | 4.98e+07 |
| Ker06 | 4.23e+07 | 3.62e+07 | 3.62e+07 | 3.62e+07 | 3.62e+07 | 3.62e+07 |
| Ker07 | 5.48e+07 | 4.90e+07 | 4.90e+07 | 4.90e+07 | 4.90e+07 | 4.90e+07 |
| Ker08 | 1.00e+08 | 8.11e+07 | 8.06e+07 | 8.06e+07 | 8.19e+07 | 8.06e+07 |
| Ker09 | 6.68e+07 | 5.66e+07 | 5.66e+07 | 5.66e+07 | 5.66e+07 | 5.66e+07 |
| Ker10 | 9.97e+07 | 8.09e+07 | 8.09e+07 | 8.09e+07 | 7.87e+07 | 8.09e+07 |
| Ker11 | 5.48e+07 | 4.75e+07 | 4.75e+07 | 4.75e+07 | 4.75e+07 | 4.75e+07 |
| Ker12 | 5.82e+07 | 5.05e+07 | 5.05e+07 | 5.05e+07 | 5.05e+07 | 5.05e+07 |
| Ker13 | 7.86e+07 | 6.49e+07 | 6.48e+07 | 6.48e+07 | 6.48e+07 | 6.48e+07 |
| Ker14 | 7.80e+07 | 6.63e+07 | 6.63e+07 | 6.63e+07 | 6.63e+07 | 6.63e+07 |
| Amean | 6.41e+07 | 5.43e+07 | 5.42e+07 | 5.42e+07 | 5.42e+07 | 5.42e+07 |
| Nasker | | | | | | |
| BTR | 1.62e+09 | 1.21e+09 | 1.18e+09 | 1.18e+09 | 1.18e+09 | 1.18e+09 |
| CHO | 1.94e+09 | 1.57e+09 | 1.54e+09 | 1.54e+09 | 1.54e+09 | 1.54e+09 |
| EMI | 8.74e+08 | 5.38e+08 | 5.28e+08 | 5.18e+08 | 5.18e+08 | 5.28e+08 |
| FFT | 1.52e+09 | 1.23e+09 | 1.19e+09 | 1.19e+09 | 1.20e+09 | 1.19e+09 |
| GMT | 1.49e+09 | 1.28e+09 | 1.28e+09 | 1.28e+09 | 1.28e+09 | 1.28e+09 |
| MXM | 1.77e+09 | 1.24e+09 | 1.11e+09 | 1.11e+09 | 1.11e+09 | 1.11e+09 |
| VPE | 5.11e+07 | 4.22e+07 | 4.13e+07 | 4.13e+07 | 4.22e+07 | 4.10e+07 |
| Amean | 1.32e+09 | 1.02e+09 | 9.81e+08 | 9.80e+08 | 9.81e+08 | 9.81e+08 |
| Optimized Nasker | | | | | | |
| BTR | 1.25e+09 | 8.83e+08 | 8.49e+08 | 8.47e+08 | 8.43e+08 | 8.43e+08 |
| CHO | 1.08e+09 | 8.39e+08 | 7.78e+08 | 7.78e+08 | 7.78e+08 | 7.78e+08 |
| EMI | 7.84e+08 | 5.23e+08 | 5.13e+08 | 5.03e+08 | 5.03e+08 | 5.13e+08 |
| FFT | 1.41e+09 | 1.10e+09 | 1.06e+09 | 1.06e+09 | 1.08e+09 | 1.06e+09 |
| GMT | 1.11e+09 | 9.00e+08 | 9.00e+08 | 9.00e+08 | 9.01e+08 | 9.01e+08 |
| MXM | 1.20e+09 | 7.94e+08 | 7.30e+08 | 7.30e+08 | 7.30e+08 | 7.30e+08 |
| VPE | 5.05e+07 | 4.22e+07 | 4.13e+07 | 4.22e+07 | 4.16e+07 | 4.10e+07 |
| Amean | 9.84e+08 | 7.26e+08 | 6.96e+08 | 6.94e+08 | 6.97e+08 | 6.95e+08 |

Table F.24: **Ar2spf RASE raw data for the Perfect, Misc and Int groups** for six register set sizes.

| Program | Register Set Size | | | | | |
|---|---|---|---|---|---|---|
| | 16 | 32 | 32/16 | 64 | 64/32 | 128 |
| Perfect | | | | | | |
| ARC2D | 4.12e+10 | 2.37e+10 | 2.23e+10 | 2.18e+10 | 2.18e+10 | 2.18e+10 |
| BDNA | 1.06e+10 | 6.09e+09 | 5.68e+09 | 5.14e+09 | 4.69e+09 | 4.51e+09 |
| DYFESM | 6.01e+09 | 4.61e+09 | 4.61e+09 | 4.61e+09 | 4.61e+09 | 4.61e+09 |
| FLO52 | 9.37e+09 | 8.64e+09 | 8.40e+09 | 8.39e+09 | 8.40e+09 | 8.39e+09 |
| MDG | 1.49e+10 | 1.21e+10 | 1.21e+10 | 1.21e+10 | 1.20e+10 | 1.21e+10 |
| MG3D | 1.44e+11 | 9.08e+10 | 8.68e+10 | 8.54e+10 | 8.54e+10 | 8.54e+10 |
| QCD | 3.11e+09 | 2.71e+09 | 2.71e+09 | 2.71e+09 | 2.69e+09 | 2.70e+09 |
| TRACK | 1.19e+09 | 1.05e+09 | 1.05e+09 | 1.02e+09 | 1.02e+09 | 1.01e+09 |
| Amean | 2.88e+10 | 1.87e+10 | 1.80e+10 | 1.76e+10 | 1.76e+10 | 1.76e+10 |
| Misc | | | | | | |
| BOAST | 6.12e+09 | 4.46e+09 | 4.37e+09 | 4.36e+09 | 4.36e+09 | 4.36e+09 |
| SPHOT | 8.44e+07 | 6.87e+07 | 6.87e+07 | 6.81e+07 | 6.81e+07 | 6.81e+07 |
| WANAL1 | 9.58e+09 | 8.62e+09 | 8.62e+09 | 8.62e+09 | 8.62e+09 | 8.62e+09 |
| NEURAL | 1.15e+09 | 9.59e+08 | 9.51e+08 | 9.52e+08 | 9.52e+08 | 9.52e+08 |
| COSTSC | 1.31e+08 | 9.36e+07 | 9.25e+07 | 8.43e+07 | 8.43e+07 | 8.43e+07 |
| Amean | 3.41e+09 | 2.84e+09 | 2.82e+09 | 2.82e+09 | 2.82e+09 | 2.82e+09 |
| Int | | | | | | |
| DHRYSTO | 1.38e+07 | 1.30e+07 | 1.30e+07 | 1.30e+07 | 1.30e+07 | 1.30e+07 |
| INTEGER | 1.19e+08 | 1.19e+08 | 1.19e+08 | 1.19e+08 | 1.19e+08 | 1.19e+08 |
| KNIGHT | 4.31e+06 | 3.35e+06 | 3.35e+06 | 3.35e+06 | 3.35e+06 | 3.35e+06 |
| LCC | 1.61e+07 | 1.51e+07 | 1.51e+07 | 1.51e+07 | 1.51e+07 | 1.51e+07 |
| SPARSE | 3.06e+06 | 2.70e+06 | 2.70e+06 | 2.70e+06 | 2.70e+06 | 2.70e+06 |
| Amean | 3.13e+07 | 3.06e+07 | 3.06e+07 | 3.06e+07 | 3.06e+07 | 3.06e+07 |

Table F.25: **A88shL Postpass raw data** for four register set sizes.

| Prog | Register Set Size 16 | 32 | 64 | 128 |
|---|---|---|---|---|
| Livermore | | | | |
| Ker01 | 1.61e+08 | 9.14e+07 | 9.14e+07 | 9.14e+07 |
| Ker02 | 1.27e+08 | 1.18e+08 | 8.30e+07 | 8.30e+07 |
| Ker03 | 1.01e+08 | 9.86e+07 | 9.86e+07 | 9.86e+07 |
| Ker04 | 1.02e+08 | 9.15e+07 | 8.54e+07 | 8.54e+07 |
| Ker05 | 1.13e+08 | 1.04e+08 | 1.04e+08 | 1.04e+08 |
| Ker06 | 9.36e+07 | 7.47e+07 | 7.45e+07 | 7.45e+07 |
| Ker07 | 1.87e+08 | 1.57e+08 | 6.84e+07 | 6.84e+07 |
| Ker08 | 2.57e+08 | 1.77e+08 | 9.96e+07 | 9.18e+07 |
| Ker09 | 1.60e+08 | 1.39e+08 | 7.74e+07 | 7.65e+07 |
| Ker10 | 2.09e+08 | 1.68e+08 | 1.42e+08 | 1.41e+08 |
| Ker11 | 1.07e+08 | 1.05e+08 | 1.05e+08 | 1.05e+08 |
| Ker12 | 1.14e+08 | 1.12e+08 | 1.12e+08 | 1.12e+08 |
| Ker13 | 1.87e+08 | 1.64e+08 | 1.14e+08 | 1.12e+08 |
| Ker14 | 2.07e+08 | 1.37e+08 | 1.31e+08 | 1.28e+08 |
| Amean | 1.52e+08 | 1.24e+08 | 9.90e+07 | 9.80e+07 |
| Nasker | | | | |
| BTR | 3.62e+09 | 2.32e+09 | 1.70e+09 | 1.41e+09 |
| CHO | 3.33e+09 | 2.16e+09 | 2.10e+09 | 1.96e+09 |
| EMI | 1.43e+09 | 1.09e+09 | 9.76e+08 | 8.70e+08 |
| FFT | 3.53e+09 | 2.59e+09 | 1.97e+09 | 1.80e+09 |
| GMT | 2.45e+09 | 1.90e+09 | 1.60e+09 | 1.60e+09 |
| MXM | 3.01e+09 | 1.72e+09 | 1.15e+09 | 1.10e+09 |
| VPE | 1.51e+08 | 9.53e+07 | 7.13e+07 | 7.03e+07 |
| Amean | 2.50e+09 | 1.70e+09 | 1.37e+09 | 1.26e+09 |
| Optimized Nasker | | | | |
| BTR | 3.03e+09 | 1.95e+09 | 1.30e+09 | 1.21e+09 |
| CHO | 2.33e+09 | 1.56e+09 | 1.46e+09 | 1.46e+09 |
| EMI | 1.33e+09 | 1.20e+09 | 9.61e+08 | 8.85e+08 |
| FFT | 3.31e+09 | 2.17e+09 | 1.87e+09 | 1.81e+09 |
| GMT | 2.48e+09 | 1.69e+09 | 1.56e+09 | 1.47e+09 |
| MXM | 3.10e+09 | 1.65e+09 | 9.41e+08 | 9.40e+08 |
| VPE | 1.51e+08 | 9.02e+07 | 7.13e+07 | 7.03e+07 |
| Amean | 2.25e+09 | 1.47e+09 | 1.17e+09 | 1.12e+09 |

| Prog | Register Set Size 16 | 32 | 64 | 128 |
|---|---|---|---|---|
| Perfect | | | | |
| ARC2D | 7.41e+10 | 4.91e+10 | 3.95e+10 | 3.59e+10 |
| BDNA | 1.92e+10 | 1.23e+10 | 9.38e+09 | 8.22e+09 |
| DYFESM | 1.12e+10 | 9.77e+09 | 8.82e+09 | 8.77e+09 |
| FLO52 | 1.91e+10 | 1.39e+10 | 1.21e+10 | 1.18e+10 |
| MDG | 3.39e+10 | 3.10e+10 | 3.00e+10 | 3.04e+10 |
| MG3D | 3.39e+11 | 2.15e+11 | 1.78e+11 | 1.52e+11 |
| QCD | 6.75e+09 | 6.09e+09 | 5.78e+09 | 5.56e+09 |
| TRACK | 2.82e+09 | 2.39e+09 | 2.28e+09 | 2.25e+09 |
| Amean | 6.33e+10 | 4.24e+10 | 3.57e+10 | 3.19e+10 |
| Misc | | | | |
| BOAST | 1.52e+10 | 1.11e+10 | 9.73e+09 | 9.38e+09 |
| SPHOT | 1.83e+08 | 1.55e+08 | 1.43e+08 | 1.42e+08 |
| WANAL1 | 2.33e+10 | 1.69e+10 | 1.47e+10 | 1.45e+10 |
| NEURAL | 3.00e+09 | 2.38e+09 | 1.98e+09 | 1.97e+09 |
| COSTSC | 3.60e+08 | 3.24e+08 | 2.93e+08 | 2.80e+08 |
| Amean | 8.41e+09 | 6.17e+09 | 5.37e+09 | 5.25e+09 |
| Int | | | | |
| DHRYSTO | 2.53e+07 | 2.47e+07 | 2.47e+07 | 2.47e+07 |
| INTEGER | 1.82e+08 | 1.68e+08 | 1.63e+08 | 1.54e+08 |
| KNIGHT | 6.75e+06 | 5.57e+06 | 4.23e+06 | 4.23e+06 |
| LCC | 3.75e+07 | 3.86e+07 | 4.09e+07 | 4.11e+07 |
| SPARSE | 7.88e+06 | 7.31e+06 | 7.25e+06 | 7.04e+06 |
| Amean | 5.19e+07 | 4.88e+07 | 4.80e+07 | 4.62e+07 |

Table F.26: **A88shL IPS raw data** for four register set sizes.

| Prog | Register Set Size 16 | 32 | 64 | 128 |
|---|---|---|---|---|
| Livermore | | | | |
| Ker01 | 9.14e+07 | 9.14e+07 | 9.14e+07 | 9.14e+07 |
| Ker02 | 8.31e+07 | 8.30e+07 | 8.30e+07 | 8.30e+07 |
| Ker03 | 9.86e+07 | 9.86e+07 | 9.86e+07 | 9.86e+07 |
| Ker04 | 8.54e+07 | 8.54e+07 | 8.54e+07 | 8.54e+07 |
| Ker05 | 1.04e+08 | 1.04e+08 | 1.04e+08 | 1.04e+08 |
| Ker06 | 7.45e+07 | 7.45e+07 | 7.45e+07 | 7.45e+07 |
| Ker07 | 8.91e+07 | 7.01e+07 | 7.01e+07 | 7.01e+07 |
| Ker08 | 3.74e+08 | 9.09e+07 | 9.09e+07 | 9.04e+07 |
| Ker09 | 1.07e+08 | 7.88e+07 | 7.88e+07 | 7.88e+07 |
| Ker10 | 2.74e+08 | 1.42e+08 | 1.41e+08 | 1.41e+08 |
| Ker11 | 1.05e+08 | 1.05e+08 | 1.05e+08 | 1.05e+08 |
| Ker12 | 1.12e+08 | 1.12e+08 | 1.12e+08 | 1.12e+08 |
| Ker13 | 1.86e+08 | 1.13e+08 | 1.13e+08 | 1.13e+08 |
| Ker14 | 1.57e+08 | 1.28e+08 | 1.28e+08 | 1.28e+08 |
| Amean | 1.39e+08 | 9.83e+07 | 9.83e+07 | 9.82e+07 |
| Nasker | | | | |
| BTR | 2.94e+09 | 1.43e+09 | 1.35e+09 | 1.36e+09 |
| CHO | 2.52e+09 | 1.58e+09 | 1.58e+09 | 1.58e+09 |
| EMI | 1.35e+09 | 8.75e+08 | 8.85e+08 | 8.70e+08 |
| FFT | 2.89e+09 | 1.79e+09 | 1.80e+09 | 1.80e+09 |
| GMT | 1.61e+09 | 1.60e+09 | 1.60e+09 | 1.60e+09 |
| MXM | 2.88e+09 | 1.11e+09 | 1.11e+09 | 1.11e+09 |
| VPE | 1.26e+08 | 6.79e+07 | 6.77e+07 | 6.79e+07 |
| Amean | 2.05e+09 | 1.21e+09 | 1.20e+09 | 1.20e+09 |
| Optimized Nasker | | | | |
| BTR | 2.88e+09 | 1.25e+09 | 1.19e+09 | 1.20e+09 |
| CHO | 2.27e+09 | 1.36e+09 | 1.36e+09 | 1.36e+09 |
| EMI | 2.01e+09 | 8.65e+08 | 8.85e+08 | 8.75e+08 |
| FFT | 3.03e+09 | 1.72e+09 | 1.81e+09 | 1.81e+09 |
| GMT | 1.50e+09 | 1.47e+09 | 1.47e+09 | 1.47e+09 |
| MXM | 2.33e+09 | 9.56e+08 | 9.56e+08 | 9.56e+08 |
| VPE | 1.25e+08 | 6.79e+07 | 6.77e+07 | 6.79e+07 |
| Amean | 2.02e+09 | 1.10e+09 | 1.11e+09 | 1.11e+09 |

| Prog | Register Set Size 16 | 32 | 64 | 128 |
|---|---|---|---|---|
| Perfect | | | | |
| ARC2D | 8.27e+10 | 4.15e+10 | 3.48e+10 | 3.45e+10 |
| BDNA | 2.37e+10 | 1.49e+10 | 9.83e+09 | 7.88e+09 |
| DYFESM | 8.96e+09 | 8.74e+09 | 8.74e+09 | 8.74e+09 |
| FLO52 | 1.60e+10 | 1.14e+10 | 1.12e+10 | 1.12e+10 |
| MDG | 3.32e+10 | 3.00e+10 | 2.94e+10 | 2.94e+10 |
| MG3D | 3.65e+11 | 1.83e+11 | 1.45e+11 | 1.45e+11 |
| QCD | 6.58e+09 | 6.12e+09 | 6.08e+09 | 6.09e+09 |
| TRACK | 2.78e+09 | 2.43e+09 | 2.21e+09 | 2.21e+09 |
| Amean | 6.74e+10 | 3.73e+10 | 3.09e+10 | 3.06e+10 |
| Misc | | | | |
| BOAST | 1.45e+10 | 9.28e+09 | 8.88e+09 | 8.86e+09 |
| SPHOT | 1.74e+08 | 1.34e+08 | 1.34e+08 | 1.34e+08 |
| WANAL1 | 1.73e+10 | 1.24e+10 | 1.27e+10 | 1.27e+10 |
| NEURAL | 3.20e+09 | 1.86e+09 | 1.86e+09 | 1.86e+09 |
| COSTSC | 3.52e+08 | 2.70e+08 | 2.32e+08 | 2.32e+08 |
| Amean | 7.11e+09 | 4.79e+09 | 4.76e+09 | 4.76e+09 |
| Int | | | | |
| DHRYSTO | 2.45e+07 | 2.40e+07 | 2.40e+07 | 2.40e+07 |
| INTEGER | 1.54e+08 | 1.54e+08 | 1.54e+08 | 1.54e+08 |
| KNIGHT | 3.70e+06 | 3.70e+06 | 3.70e+06 | 3.70e+06 |
| LCC | 2.77e+07 | 2.77e+07 | 2.77e+07 | 2.77e+07 |
| SPARSE | 6.29e+06 | 6.04e+06 | 6.04e+06 | 6.04e+06 |
| Amean | 4.32e+07 | 4.31e+07 | 4.31e+07 | 4.31e+07 |

Table F.27: **A88shL RASE raw data** for four register set sizes.

| Prog | Register Set Size 16 | 32 | 64 | 128 |
|---|---|---|---|---|
| Livermore | | | | |
| Ker01 | 9.14e+07 | 9.14e+07 | 9.14e+07 | 9.14e+07 |
| Ker02 | 8.30e+07 | 8.30e+07 | 8.30e+07 | 8.30e+07 |
| Ker03 | 9.86e+07 | 9.86e+07 | 9.86e+07 | 9.86e+07 |
| Ker04 | 8.54e+07 | 8.54e+07 | 8.54e+07 | 8.54e+07 |
| Ker05 | 1.04e+08 | 1.04e+08 | 1.04e+08 | 1.04e+08 |
| Ker06 | 7.45e+07 | 7.45e+07 | 7.45e+07 | 7.45e+07 |
| Ker07 | 7.01e+07 | 7.01e+07 | 7.01e+07 | 7.01e+07 |
| Ker08 | 1.52e+08 | 9.04e+07 | 9.04e+07 | 9.22e+07 |
| Ker09 | 7.74e+07 | 7.65e+07 | 7.65e+07 | 7.65e+07 |
| Ker10 | 1.43e+08 | 1.41e+08 | 1.41e+08 | 1.41e+08 |
| Ker11 | 1.05e+08 | 1.05e+08 | 1.05e+08 | 1.05e+08 |
| Ker12 | 1.12e+08 | 1.12e+08 | 1.12e+08 | 1.12e+08 |
| Ker13 | 1.56e+08 | 1.17e+08 | 1.13e+08 | 1.13e+08 |
| Ker14 | 1.42e+08 | 1.28e+08 | 1.28e+08 | 1.28e+08 |
| Amean | 1.07e+08 | 9.83e+07 | 9.81e+07 | 9.82e+07 |
| Nasker | | | | |
| BTR | 1.97e+09 | 1.38e+09 | 1.34e+09 | 1.35e+09 |
| CHO | 2.22e+09 | 1.58e+09 | 1.58e+09 | 1.58e+09 |
| EMI | 1.35e+09 | 8.80e+08 | 8.70e+08 | 8.70e+08 |
| FFT | 2.49e+09 | 1.79e+09 | 1.80e+09 | 1.80e+09 |
| GMT | 1.61e+09 | 1.60e+09 | 1.60e+09 | 1.60e+09 |
| MXM | 1.83e+09 | 1.10e+09 | 1.10e+09 | 1.10e+09 |
| VPE | 9.65e+07 | 6.79e+07 | 6.77e+07 | 6.79e+07 |
| Amean | 1.65e+09 | 1.20e+09 | 1.19e+09 | 1.20e+09 |
| Optimized Nasker | | | | |
| BTR | 1.73e+09 | 1.19e+09 | 1.16e+09 | 1.18e+09 |
| CHO | 1.55e+09 | 1.36e+09 | 1.36e+09 | 1.36e+09 |
| EMI | 1.34e+09 | 8.75e+08 | 8.75e+08 | 8.75e+08 |
| FFT | 2.31e+09 | 1.73e+09 | 1.81e+09 | 1.81e+09 |
| GMT | 1.48e+09 | 1.47e+09 | 1.47e+09 | 1.47e+09 |
| MXM | 1.54e+09 | 9.40e+08 | 9.40e+08 | 9.40e+08 |
| VPE | 9.65e+07 | 6.79e+07 | 6.77e+07 | 6.79e+07 |
| Amean | 1.44e+09 | 1.09e+09 | 1.10e+09 | 1.10e+09 |

| Prog | Register Set Size 16 | 32 | 64 | 128 |
|---|---|---|---|---|
| Perfect | | | | |
| ARC2D | 6.36e+10 | 3.60e+10 | 3.43e+10 | 3.43e+10 |
| BDNA | 2.15e+10 | 1.05e+10 | 8.29e+09 | 7.64e+09 |
| DYFESM | 8.78e+09 | 8.74e+09 | 8.74e+09 | 8.74e+09 |
| FLO52 | 1.25e+10 | 1.12e+10 | 1.12e+10 | 1.12e+10 |
| MDG | 3.30e+10 | 3.00e+10 | 2.98e+10 | 2.98e+10 |
| MG3D | 2.70e+11 | 1.55e+11 | 1.44e+11 | 1.44e+11 |
| QCD | 5.57e+09 | 5.27e+09 | 5.27e+09 | 5.27e+09 |
| TRACK | 2.76e+09 | 2.19e+09 | 2.15e+09 | 2.15e+09 |
| Amean | 5.22e+10 | 3.24e+10 | 3.05e+10 | 3.04e+10 |
| Misc | | | | |
| BOAST | 1.28e+10 | 9.03e+09 | 8.84e+09 | 8.84e+09 |
| SPHOT | 1.53e+08 | 1.34e+08 | 1.34e+08 | 1.34e+08 |
| WANAL1 | 1.73e+10 | 1.24e+10 | 1.24e+10 | 1.24e+10 |
| NEURAL | 2.46e+09 | 1.86e+09 | 1.86e+09 | 1.86e+09 |
| COSTSC | 3.39e+08 | 2.73e+08 | 2.32e+08 | 2.32e+08 |
| Amean | 6.61e+09 | 4.74e+09 | 4.69e+09 | 4.69e+09 |
| Int | | | | |
| DHRYSTO | 2.44e+07 | 2.44e+07 | 2.44e+07 | 2.44e+07 |
| INTEGER | 1.54e+08 | 1.54e+08 | 1.54e+08 | 1.54e+08 |
| KNIGHT | 3.74e+06 | 3.74e+06 | 3.74e+06 | 3.74e+06 |
| LCC | 2.81e+07 | 2.81e+07 | 2.81e+07 | 2.81e+07 |
| SPARSE | 6.42e+06 | 6.07e+06 | 6.07e+06 | 6.07e+06 |
| Amean | 4.33e+07 | 4.33e+07 | 4.33e+07 | 4.33e+07 |

Table F.28: **Ar2spL Postpass raw data for the Livermore and Nasker groups** for six register set sizes.

| Program | Register Set Size | | | | | |
|---|---|---|---|---|---|---|
| | 16 | 32 | 32/16 | 64 | 64/32 | 128 |
| Livermore | | | | | | |
| Ker01 | 1.21e+08 | 7.82e+07 | 7.66e+07 | 7.66e+07 | 7.66e+07 | 7.66e+07 |
| Ker02 | 1.85e+08 | 8.92e+07 | 7.62e+07 | 7.62e+07 | 7.62e+07 | 7.62e+07 |
| Ker03 | 1.09e+08 | 8.89e+07 | 8.89e+07 | 8.89e+07 | 8.89e+07 | 8.89e+07 |
| Ker04 | 1.49e+08 | 7.95e+07 | 7.94e+07 | 7.94e+07 | 7.78e+07 | 7.78e+07 |
| Ker05 | 1.56e+08 | 9.62e+07 | 9.40e+07 | 9.40e+07 | 9.40e+07 | 9.40e+07 |
| Ker06 | 1.46e+08 | 7.00e+07 | 6.83e+07 | 6.83e+07 | 6.83e+07 | 6.83e+07 |
| Ker07 | 1.37e+08 | 1.01e+08 | 7.58e+07 | 7.01e+07 | 5.93e+07 | 5.93e+07 |
| Ker08 | 2.57e+08 | 1.49e+08 | 1.19e+08 | 1.10e+08 | 9.40e+07 | 7.79e+07 |
| Ker09 | 1.14e+08 | 9.05e+07 | 7.89e+07 | 7.58e+07 | 7.50e+07 | 7.50e+07 |
| Ker10 | 2.61e+08 | 2.12e+08 | 2.02e+08 | 2.02e+08 | 1.93e+08 | 1.93e+08 |
| Ker11 | 1.77e+08 | 1.03e+08 | 9.78e+07 | 9.78e+07 | 9.78e+07 | 9.78e+07 |
| Ker12 | 1.84e+08 | 1.07e+08 | 1.04e+08 | 1.04e+08 | 1.04e+08 | 1.04e+08 |
| Ker13 | 2.81e+08 | 1.48e+08 | 1.22e+08 | 1.22e+08 | 1.06e+08 | 1.06e+08 |
| Ker14 | 2.33e+08 | 1.56e+08 | 1.27e+08 | 1.27e+08 | 1.25e+08 | 1.21e+08 |
| Amean | 1.79e+08 | 1.12e+08 | 1.01e+08 | 9.94e+07 | 9.54e+07 | 9.40e+07 |
| Nasker | | | | | | |
| BTR | 4.18e+09 | 2.57e+09 | 1.96e+09 | 1.92e+09 | 1.53e+09 | 1.49e+09 |
| CHO | 5.71e+09 | 3.92e+09 | 3.49e+09 | 3.49e+09 | 3.20e+09 | 3.20e+09 |
| EMI | 1.33e+09 | 1.02e+09 | 8.94e+08 | 8.58e+08 | 7.98e+08 | 7.18e+08 |
| FFT | 4.11e+09 | 2.19e+09 | 2.19e+09 | 1.97e+09 | 1.97e+09 | 1.97e+09 |
| GMT | 5.23e+09 | 2.51e+09 | 2.01e+09 | 2.00e+09 | 2.00e+09 | 2.00e+09 |
| MXM | 4.58e+09 | 3.07e+09 | 2.16e+09 | 2.16e+09 | 1.52e+09 | 1.52e+09 |
| VPE | 1.46e+08 | 8.16e+07 | 6.22e+07 | 5.81e+07 | 5.27e+07 | 5.29e+07 |
| Amean | 3.61e+09 | 2.19e+09 | 1.82e+09 | 1.78e+09 | 1.58e+09 | 1.56e+09 |
| Optimized Nasker | | | | | | |
| BTR | 2.72e+09 | 1.70e+09 | 1.36e+09 | 1.32e+09 | 1.00e+09 | 1.01e+09 |
| CHO | 3.85e+09 | 2.05e+09 | 1.33e+09 | 1.33e+09 | 1.28e+09 | 1.29e+09 |
| EMI | 1.34e+09 | 8.48e+08 | 7.94e+08 | 8.03e+08 | 7.33e+08 | 6.98e+08 |
| FFT | 3.71e+09 | 2.20e+09 | 1.69e+09 | 1.76e+09 | 1.68e+09 | 1.68e+09 |
| GMT | 4.27e+09 | 1.76e+09 | 1.25e+09 | 1.25e+09 | 1.25e+09 | 1.25e+09 |
| MXM | 3.48e+09 | 1.57e+09 | 1.05e+09 | 1.05e+09 | 8.45e+08 | 8.45e+08 |
| VPE | 1.52e+08 | 8.16e+07 | 6.22e+07 | 5.81e+07 | 5.27e+07 | 5.27e+07 |
| Amean | 2.79e+09 | 1.46e+09 | 1.08e+09 | 1.08e+09 | 9.77e+08 | 9.75e+08 |

Table F.29: **Ar2spL Postpass raw data for the Perfect, Misc and Int groups** for six register set sizes.

| Program | Register Set Size | | | | | |
|---|---|---|---|---|---|---|
| | 16 | 32 | 32/16 | 64 | 64/32 | 128 |
| Perfect | | | | | | |
| ARC2D | 8.80e+10 | 5.26e+10 | 4.16e+10 | 3.96e+10 | 3.64e+10 | 3.57e+10 |
| BDNA | 1.79e+10 | 1.03e+10 | 9.97e+09 | 7.20e+09 | 7.21e+09 | 6.21e+09 |
| DYFESM | 1.65e+10 | 1.06e+10 | 1.04e+10 | 1.04e+10 | 1.00e+10 | 1.00e+10 |
| FLO52 | 2.56e+10 | 1.71e+10 | 1.44e+10 | 1.42e+10 | 1.36e+10 | 1.36e+10 |
| MDG | 3.31e+10 | 2.73e+10 | 2.70e+10 | 2.65e+10 | 2.64e+10 | 2.64e+10 |
| MG3D | 3.81e+11 | 2.17e+11 | 1.83e+11 | 1.63e+11 | 1.54e+11 | 1.47e+11 |
| QCD | 7.53e+09 | 5.55e+09 | 5.44e+09 | 5.35e+09 | 5.38e+09 | 5.36e+09 |
| TRACK | 2.57e+09 | 2.09e+09 | 2.05e+09 | 1.98e+09 | 1.99e+09 | 1.98e+09 |
| Amean | 7.15e+10 | 4.28e+10 | 3.67e+10 | 3.35e+10 | 3.19e+10 | 3.08e+10 |
| Misc | | | | | | |
| BOAST | 1.38e+10 | 9.50e+09 | 8.26e+09 | 7.82e+09 | 7.60e+09 | 7.44e+09 |
| SPHOT | 1.68e+08 | 1.35e+08 | 1.33e+08 | 1.18e+08 | 1.18e+08 | 1.13e+08 |
| WANAL1 | 2.87e+10 | 1.94e+10 | 1.67e+10 | 1.49e+10 | 1.53e+10 | 1.51e+10 |
| NEURAL | 3.04e+09 | 2.43e+09 | 2.05e+09 | 1.99e+09 | 1.95e+09 | 1.91e+09 |
| COSTSC | 3.71e+08 | 3.09e+08 | 2.87e+08 | 2.83e+08 | 2.70e+08 | 2.64e+08 |
| Amean | 9.22e+09 | 6.35e+09 | 5.49e+09 | 5.02e+09 | 5.05e+09 | 4.97e+09 |
| Int | | | | | | |
| DHRYSTO | 3.22e+07 | 2.55e+07 | 2.51e+07 | 2.51e+07 | 2.51e+07 | 2.51e+07 |
| INTEGER | 2.03e+08 | 1.97e+08 | 1.82e+08 | 1.82e+08 | 1.70e+08 | 1.70e+08 |
| KNIGHT | 9.49e+06 | 7.15e+06 | 5.90e+06 | 5.90e+06 | 4.61e+06 | 4.61e+06 |
| LCC | 4.02e+07 | 3.76e+07 | 3.86e+07 | 3.87e+07 | 4.10e+07 | 4.11e+07 |
| SPARSE | 8.93e+06 | 7.40e+06 | 6.99e+06 | 7.00e+06 | 6.85e+06 | 6.85e+06 |
| Amean | 5.88e+07 | 5.49e+07 | 5.17e+07 | 5.17e+07 | 4.95e+07 | 4.95e+07 |

Table F.30: **Ar2spL IPS raw data for the Livermore and Nasker groups** for six register set sizes.

| Program | Register Set Size | | | | | |
|---|---|---|---|---|---|---|
| | 16 | 32 | 32/16 | 64 | 64/32 | 128 |
| Livermore | | | | | | |
| Ker01 | 8.72e+07 | 7.66e+07 | 7.66e+07 | 7.66e+07 | 7.66e+07 | 7.66e+07 |
| Ker02 | 1.53e+08 | 7.62e+07 | 7.62e+07 | 7.62e+07 | 7.62e+07 | 7.62e+07 |
| Ker03 | 1.67e+08 | 8.89e+07 | 8.89e+07 | 8.89e+07 | 8.89e+07 | 8.89e+07 |
| Ker04 | 1.49e+08 | 7.78e+07 | 7.78e+07 | 7.78e+07 | 7.78e+07 | 7.78e+07 |
| Ker05 | 2.11e+08 | 9.40e+07 | 9.40e+07 | 9.40e+07 | 9.40e+07 | 9.40e+07 |
| Ker06 | 1.11e+08 | 6.83e+07 | 6.83e+07 | 6.83e+07 | 6.83e+07 | 6.83e+07 |
| Ker07 | 1.07e+08 | 5.93e+07 | 5.93e+07 | 5.93e+07 | 5.93e+07 | 5.93e+07 |
| Ker08 | 3.27e+08 | 1.38e+08 | 7.69e+07 | 7.69e+07 | 7.47e+07 | 7.69e+07 |
| Ker09 | 1.28e+08 | 6.88e+07 | 6.88e+07 | 6.19e+07 | 6.19e+07 | 6.19e+07 |
| Ker10 | 3.23e+08 | 1.57e+08 | 1.28e+08 | 1.28e+08 | 1.28e+08 | 1.28e+08 |
| Ker11 | 2.01e+08 | 9.78e+07 | 9.78e+07 | 9.78e+07 | 9.78e+07 | 9.78e+07 |
| Ker12 | 2.14e+08 | 1.04e+08 | 1.04e+08 | 1.04e+08 | 1.04e+08 | 1.04e+08 |
| Ker13 | 3.80e+08 | 1.22e+08 | 1.01e+08 | 1.01e+08 | 1.01e+08 | 1.01e+08 |
| Ker14 | 1.99e+08 | 1.25e+08 | 1.10e+08 | 1.10e+08 | 1.10e+08 | 1.10e+08 |
| Amean | 1.97e+08 | 9.67e+07 | 8.77e+07 | 8.72e+07 | 8.70e+07 | 8.72e+07 |
| Nasker | | | | | | |
| BTR | 4.58e+09 | 2.11e+09 | 1.51e+09 | 1.40e+09 | 1.43e+09 | 1.43e+09 |
| CHO | 5.60e+09 | 3.40e+09 | 2.79e+09 | 2.79e+09 | 2.79e+09 | 2.79e+09 |
| EMI | 1.84e+09 | 7.19e+08 | 7.17e+08 | 7.22e+08 | 7.22e+08 | 7.22e+08 |
| FFT | 4.35e+09 | 2.08e+09 | 1.78e+09 | 1.78e+09 | 1.78e+09 | 1.78e+09 |
| GMT | 4.28e+09 | 2.01e+09 | 2.00e+09 | 2.00e+09 | 2.00e+09 | 2.00e+09 |
| MXM | 4.30e+09 | 1.80e+09 | 1.55e+09 | 1.55e+09 | 1.55e+09 | 1.55e+09 |
| VPE | 1.74e+08 | 5.53e+07 | 5.19e+07 | 5.19e+07 | 5.25e+07 | 5.19e+07 |
| Amean | 3.59e+09 | 1.74e+09 | 1.49e+09 | 1.47e+09 | 1.47e+09 | 1.47e+09 |
| Optimized Nasker | | | | | | |
| BTR | 3.19e+09 | 1.29e+09 | 9.80e+08 | 9.73e+08 | 9.67e+08 | 9.62e+08 |
| CHO | 4.13e+09 | 1.64e+09 | 1.19e+09 | 1.19e+09 | 1.19e+09 | 1.19e+09 |
| EMI | 1.72e+09 | 7.04e+08 | 6.97e+08 | 7.02e+08 | 6.97e+08 | 6.97e+08 |
| FFT | 4.20e+09 | 1.90e+09 | 1.59e+09 | 1.59e+09 | 1.62e+09 | 1.61e+09 |
| GMT | 2.58e+09 | 1.25e+09 | 1.25e+09 | 1.25e+09 | 1.25e+09 | 1.25e+09 |
| MXM | 3.57e+09 | 1.70e+09 | 8.13e+08 | 8.13e+08 | 8.13e+08 | 8.13e+08 |
| VPE | 1.74e+08 | 5.53e+07 | 5.19e+07 | 5.19e+07 | 5.19e+07 | 5.19e+07 |
| Amean | 2.79e+09 | 1.22e+09 | 9.39e+08 | 9.39e+08 | 9.41e+08 | 9.39e+08 |

Table F.31: **Ar2spL IPS raw data for the Perfect, Misc and Int groups** for six register set sizes.

| Program | Register Set Size | | | | | |
|---|---|---|---|---|---|---|
| | 16 | 32 | 32/16 | 64 | 64/32 | 128 |
| Perfect | | | | | | |
| ARC2D | 1.11e+11 | 5.05e+10 | 3.60e+10 | 3.37e+10 | 3.37e+10 | 3.32e+10 |
| BDNA | 2.42e+10 | 1.28e+10 | 1.34e+10 | 9.17e+09 | 8.03e+09 | 6.22e+09 |
| DYFESM | 1.78e+10 | 1.00e+10 | 9.97e+09 | 9.97e+09 | 9.97e+09 | 9.97e+09 |
| FLO52 | 2.81e+10 | 1.53e+10 | 1.31e+10 | 1.31e+10 | 1.29e+10 | 1.30e+10 |
| MDG | 3.45e+10 | 2.61e+10 | 2.59e+10 | 2.44e+10 | 2.44e+10 | 2.43e+10 |
| MG3D | 4.70e+11 | 1.94e+11 | 1.60e+11 | 1.37e+11 | 1.37e+11 | 1.37e+11 |
| QCD | 7.07e+09 | 5.58e+09 | 5.51e+09 | 5.40e+09 | 5.46e+09 | 5.41e+09 |
| TRACK | 2.71e+09 | 2.10e+09 | 2.07e+09 | 1.93e+09 | 1.93e+09 | 1.89e+09 |
| Amean | 8.69e+10 | 3.95e+10 | 3.32e+10 | 2.93e+10 | 2.92e+10 | 2.89e+10 |
| Misc | | | | | | |
| BOAST | 1.56e+10 | 8.29e+09 | 7.68e+09 | 7.11e+09 | 7.07e+09 | 7.05e+09 |
| SPHOT | 1.59e+08 | 1.12e+08 | 1.12e+08 | 1.06e+08 | 1.06e+08 | 1.06e+08 |
| WANAL1 | 2.69e+10 | 1.35e+10 | 1.35e+10 | 1.29e+10 | 1.29e+10 | 1.29e+10 |
| NEURAL | 3.35e+09 | 2.02e+09 | 1.84e+09 | 1.81e+09 | 1.81e+09 | 1.81e+09 |
| COSTSC | 3.63e+08 | 2.19e+08 | 1.93e+08 | 1.60e+08 | 1.60e+08 | 1.60e+08 |
| Amean | 9.27e+09 | 4.83e+09 | 4.66e+09 | 4.42e+09 | 4.41e+09 | 4.41e+09 |
| Int | | | | | | |
| DHRYSTO | 3.01e+07 | 2.45e+07 | 2.44e+07 | 2.45e+07 | 2.45e+07 | 2.45e+07 |
| INTEGER | 1.62e+08 | 1.57e+08 | 1.57e+08 | 1.57e+08 | 1.57e+08 | 1.57e+08 |
| KNIGHT | 9.04e+06 | 4.07e+06 | 4.07e+06 | 4.07e+06 | 4.07e+06 | 4.07e+06 |
| LCC | 3.08e+07 | 2.78e+07 | 2.78e+07 | 2.78e+07 | 2.78e+07 | 2.78e+07 |
| SPARSE | 7.15e+06 | 5.78e+06 | 5.78e+06 | 5.77e+06 | 5.77e+06 | 5.77e+06 |
| Amean | 4.78e+07 | 4.38e+07 | 4.38e+07 | 4.38e+07 | 4.38e+07 | 4.38e+07 |

Table F.32: **Ar2spL RASE raw data for the Livermore and Nasker groups** for six register set sizes.

| Program | Register Set Size 16 | 32 | 32/16 | 64 | 64/32 | 128 |
|---|---|---|---|---|---|---|
| Livermore | | | | | | |
| Ker01 | 8.12e+07 | 7.66e+07 | 7.66e+07 | 7.66e+07 | 7.66e+07 | 7.66e+07 |
| Ker02 | 8.78e+07 | 7.62e+07 | 7.62e+07 | 7.62e+07 | 7.62e+07 | 7.62e+07 |
| Ker03 | 9.29e+07 | 8.89e+07 | 8.89e+07 | 8.89e+07 | 8.89e+07 | 8.89e+07 |
| Ker04 | 8.96e+07 | 7.78e+07 | 7.78e+07 | 7.78e+07 | 7.78e+07 | 7.78e+07 |
| Ker05 | 1.00e+08 | 9.40e+07 | 9.40e+07 | 9.40e+07 | 9.40e+07 | 9.40e+07 |
| Ker06 | 9.90e+07 | 6.83e+07 | 6.83e+07 | 6.83e+07 | 6.83e+07 | 6.83e+07 |
| Ker07 | 6.61e+07 | 5.94e+07 | 5.93e+07 | 5.93e+07 | 5.93e+07 | 5.93e+07 |
| Ker08 | 1.66e+08 | 1.85e+08 | 7.65e+07 | 7.65e+07 | 7.47e+07 | 7.65e+07 |
| Ker09 | 7.97e+07 | 6.27e+07 | 6.19e+07 | 6.19e+07 | 6.19e+07 | 6.19e+07 |
| Ker10 | 1.51e+08 | 1.28e+08 | 1.28e+08 | 1.28e+08 | 1.28e+08 | 1.28e+08 |
| Ker11 | 1.05e+08 | 9.78e+07 | 9.78e+07 | 9.78e+07 | 9.78e+07 | 9.78e+07 |
| Ker12 | 1.12e+08 | 1.04e+08 | 1.04e+08 | 1.04e+08 | 1.04e+08 | 1.04e+08 |
| Ker13 | 1.71e+08 | 1.20e+08 | 1.01e+08 | 1.01e+08 | 1.01e+08 | 1.01e+08 |
| Ker14 | 1.23e+08 | 1.15e+08 | 1.10e+08 | 1.10e+08 | 1.10e+08 | 1.10e+08 |
| Amean | 1.09e+08 | 9.67e+07 | 8.72e+07 | 8.72e+07 | 8.70e+07 | 8.72e+07 |
| Nasker | | | | | | |
| BTR | 3.71e+09 | 1.89e+09 | 1.60e+09 | 1.53e+09 | 1.55e+09 | 1.54e+09 |
| CHO | 4.13e+09 | 3.18e+09 | 2.79e+09 | 2.79e+09 | 2.79e+09 | 2.79e+09 |
| EMI | 1.74e+09 | 7.78e+08 | 7.22e+08 | 7.17e+08 | 7.17e+08 | 7.18e+08 |
| FFT | 2.44e+09 | 2.14e+09 | 1.78e+09 | 1.78e+09 | 1.78e+09 | 1.78e+09 |
| GMT | 3.14e+09 | 2.00e+09 | 2.00e+09 | 2.00e+09 | 2.00e+09 | 2.00e+09 |
| MXM | 4.85e+09 | 2.11e+09 | 1.51e+09 | 1.51e+09 | 1.51e+09 | 1.51e+09 |
| VPE | 1.06e+08 | 5.34e+07 | 5.17e+07 | 5.17e+07 | 5.24e+07 | 5.17e+07 |
| Amean | 2.87e+09 | 1.74e+09 | 1.49e+09 | 1.48e+09 | 1.49e+09 | 1.48e+09 |
| Optimized Nasker | | | | | | |
| BTR | 2.82e+09 | 1.23e+09 | 1.03e+09 | 9.97e+08 | 9.88e+08 | 9.86e+08 |
| CHO | 2.85e+09 | 1.33e+09 | 1.19e+09 | 1.19e+09 | 1.19e+09 | 1.19e+09 |
| EMI | 1.63e+09 | 7.58e+08 | 7.02e+08 | 6.97e+08 | 6.97e+08 | 6.97e+08 |
| FFT | 2.30e+09 | 1.92e+09 | 1.59e+09 | 1.59e+09 | 1.62e+09 | 1.61e+09 |
| GMT | 2.97e+09 | 1.25e+09 | 1.25e+09 | 1.25e+09 | 1.25e+09 | 1.25e+09 |
| MXM | 3.11e+09 | 9.57e+08 | 8.13e+08 | 8.13e+08 | 8.13e+08 | 8.13e+08 |
| VPE | 9.90e+07 | 5.34e+07 | 5.17e+07 | 5.17e+07 | 5.24e+07 | 5.17e+07 |
| Amean | 2.25e+09 | 1.07e+09 | 9.47e+08 | 9.41e+08 | 9.44e+08 | 9.43e+08 |

Table F.33: **Ar2spL RASE raw data for the Perfect, Misc and Int groups** for six register set sizes.

| Program | Register Set Size | | | | | |
|---|---|---|---|---|---|---|
| | 16 | 32 | 32/16 | 64 | 64/32 | 128 |
| Perfect | | | | | | |
| ARC2D | 8.69e+10 | 3.71e+10 | 3.48e+10 | 3.35e+10 | 3.34e+10 | 3.32e+10 |
| BDNA | 2.43e+10 | 1.01e+10 | 9.27e+09 | 6.59e+09 | 6.21e+09 | 5.79e+09 |
| DYFESM | 1.08e+10 | 9.97e+09 | 9.96e+09 | 9.96e+09 | 9.96e+09 | 9.96e+09 |
| FLO52 | 1.92e+10 | 1.38e+10 | 1.29e+10 | 1.29e+10 | 1.29e+10 | 1.29e+10 |
| MDG | 3.06e+10 | 2.50e+10 | 2.46e+10 | 2.43e+10 | 2.43e+10 | 2.43e+10 |
| MG3D | 3.56e+11 | 2.30e+11 | 1.47e+11 | 1.40e+11 | 1.40e+11 | 1.40e+11 |
| QCD | 6.58e+09 | 5.34e+09 | 5.22e+09 | 5.19e+09 | 5.20e+09 | 5.20e+09 |
| TRACK | 2.49e+09 | 2.02e+09 | 1.99e+09 | 1.88e+09 | 1.87e+09 | 1.85e+09 |
| Amean | 6.71e+10 | 4.17e+10 | 3.07e+10 | 2.93e+10 | 2.92e+10 | 2.92e+10 |
| Misc | | | | | | |
| BOAST | 1.35e+10 | 7.57e+09 | 7.18e+09 | 7.09e+09 | 7.08e+09 | 7.08e+09 |
| SPHOT | 1.45e+08 | 1.09e+08 | 1.09e+08 | 1.07e+08 | 1.07e+08 | 1.07e+08 |
| WANAL1 | 2.76e+10 | 1.30e+10 | 1.30e+10 | 1.30e+10 | 1.30e+10 | 1.30e+10 |
| NEURAL | 3.05e+09 | 1.94e+09 | 1.82e+09 | 1.82e+09 | 1.82e+09 | 1.82e+09 |
| COSTSC | 3.20e+08 | 2.12e+08 | 1.99e+08 | 1.62e+08 | 1.62e+08 | 1.62e+08 |
| Amean | 8.92e+09 | 4.57e+09 | 4.46e+09 | 4.44e+09 | 4.43e+09 | 4.43e+09 |
| Int | | | | | | |
| DHRYSTO | 3.17e+07 | 2.49e+07 | 2.49e+07 | 2.49e+07 | 2.49e+07 | 2.49e+07 |
| INTEGER | 1.57e+08 | 1.57e+08 | 1.57e+08 | 1.57e+08 | 1.57e+08 | 1.57e+08 |
| KNIGHT | 9.60e+06 | 4.15e+06 | 4.15e+06 | 4.15e+06 | 4.15e+06 | 4.15e+06 |
| LCC | 3.77e+07 | 2.85e+07 | 2.84e+07 | 2.84e+07 | 2.84e+07 | 2.84e+07 |
| SPARSE | 7.93e+06 | 5.84e+06 | 5.82e+06 | 5.82e+06 | 5.82e+06 | 5.82e+06 |
| Amean | 4.88e+07 | 4.41e+07 | 4.41e+07 | 4.41e+07 | 4.41e+07 | 4.41e+07 |