4-1-1988

Emulation of a Complex Instruction Set Computer with a Reduced Instruction Set Computer

K. J. McNeley  
*Purdue University*

V. M. Milutinovic’  
*Purdue University*

Follow this and additional works at: https://docs.lib.purdue.edu/ecetr

https://docs.lib.purdue.edu/ecetr/600

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries. Please contact epubs@purdue.edu for additional information.
Emulation of a Complex Instruction Set Computer with a Reduced Instruction Set Computer

K. J. McNeley  
V. M. Milutinović

TR-EE 88-15  
April 1988

School of Electrical Engineering  
Purdue University  
West Lafayette, Indiana 47907

This research (especially in its final phases) was partially supported by NCR Corporation, World’s Headquarters, Dayton, Ohio.
Emulation of a Complex Instruction Set Computer With a Reduced Instruction Set Computer

K.J. McNeley and V.M. Milutinovic

School of Electrical Engineering
Purdue University
West Lafayette, Indiana 47907

ABSTRACT

This paper analyzes some of the difficulties of emulating a Complex Instruction Set Computer (CISC) with a Reduced Instruction Set Computer (RISC). It will be shown that although the speed advantage of a RISC is sacrificed, a CISC can be emulated with the exception of software constructs that support nonstandard hardware interfaces. Some concrete examples will be used to help illustrate the execution-time bottlenecks as well as to discuss possible solutions from an architectural point of view for both Silicon and Gallium Arsenide (GaAs). In addition, it will be shown that the most efficient method of emulation involves debugging compiled High-Level Language (HLL) source code on a CISC, and then recompiling the HLL code with a compiler that is familiar with the target RISC architecture.
Table of Contents

Section 1: INTRODUCTION

1.1 The Interest in GaAs 1
1.2 The Motivation Behind the Emulation of a CISC 1
1.3 Benefits Resulting from the Emulation 2

Section 2: EMULATION EFFICIENCY OF A CISC WITH A RISC 3

2.1 Problems Encountered in Emulation 3
   2.1.1 Addressing Mode Emulation 3
   2.1.2 Condition Code Support 4
   2.1.3 Trace Mode Emulation 4
   2.1.4 Software Support of Infrequently Used Instructions 5

2.2 A Solution to the Efficiency Problem 6

Section 3: THE TRANSLATOR'S SOPHISTICATION 7

3.1 Condition Code Optimization 7
   3.1.1 Locating Conditional Instructions 7
   3.1.2 Substituting RISC Supported Conditional Constructs 8
   3.1.3 A Default Solution to Condition Code Synthesis 8

3.2 The Impact of the Data Format on Efficiency 9
3.3 The Effect of Increasing the Register File Size 10
   3.3.1 Decreasing the LOAD/STORE Latency 10
   3.3.2 The Impact on the Instruction Format 11

Section 4: PIPELINE CONSIDERATIONS 12

4.1 Why Pipelining is an Important Consideration 12
4.2 Pipelining in the Silicon Environment 12
4.3 Pipelining in the GaAs Environment 13

Section 5: PACKING CONSIDERATIONS 14

5.1 Intra-instruction Packing 14
5.2 Inter-instruction Packing 14
Table of Contents

Section 6: OTHER ARCHITECTURAL CONSIDERATIONS 15

6.1 On-Package Cache Support 15
6.2 Coprocessor Support 16

6.2.1 Monitoring the Instruction Bus 16
6.2.2 Conditional Actions Based on Coprocessor Conditions 16
6.2.3 An Operand Transfer Scheme 17

6.3 Bit and Byte Field Support 17

Section 7: INSTRUCTION TRANSLATION 18

7.1 Translation From MC68020 Code to MIPS Code 18
7.2 Disassembly and Decomposition of the MC68020 Instructions 18
7.3 Condition Code Analysis and Interinstruction Optimization 19
7.4 Standard MIPS Optimization Mechanisms 19

Section 8: CONCLUSION 20

Section 9: ACKNOWLEDGEMENTS 20

Section 10: REFERENCES 21

Section 11: TABLES AND FIGURES 23

Table 1. PERFORMANCE COMPARISON OF GaAs AND SILICON 24
Figure 1. SU-MIPS (4 MHz.) EXECUTION EFFICIENCY FOR 25
DIFFERENT COMPILED BENCHMARKS
Figure 2. MC68020 (10 MHz.) EXECUTION EFFICIENCY FOR 26
DIFFERENT COMPILED BENCHMARKS
Figure 3. COMPARISON OF HLL EXECUTION EFFICIENCIES AND CODE 27
SIZE FOR DIFFERENT METHODS OF CODE GENERATION
Figure 4. CACHE HIT RATIOS VERSUS CACHE SIZE 28
Table of Contents

Appendix A ADDRESSING MODE EMULATION

A.1 Register Allocation 28
A.2 Addressing modes supported by the MC68020 29
A.3 Register Definitions used in Translation 29
A.4 Addressing Mode Translation 30

Appendix B CONDITION CODE SYNTHESIS 32

B.1 Condition Code Bit Map 32
B.2 Basic Conditions Tested 32
B.3 Macros Preceding a Conditional Instruction 33
B.4 Condition Code Translation 35

Appendix C INSTRUCTION SET SYNTHESIS 45

C.1 Register Definitions used in Translation 45
C.2 Instruction Set Translation 46

Appendix D QUICKSORT BENCHMARK TRANSLATION 55

D.1 The Purpose of this Appendix 55
D.2 Pseudocode used for the Quicksort Benchmark 56
D.3 MC68020 Assembly Language Code for Quicksort 57
D.4 SU-MIPS Assembly Language Code for Quicksort 61
D.5 Translated MC68020 Assembly Language Code for Quicksort 68
INTRODUCTION

1.1 The Interest in GaAs

Gallium Arsenide (GaAs) has reached a level of integration where it is now feasible to design very high speed computer components of moderate scale. The current fabrication limitation is about 30K transistors per chip [HlinM84] and is expected to more than double by the year 2000. Since several 32-bit microprocessor designs exist today that require fewer than 60K transistors to implement [1], GaAs technology has attracted a great deal of scientific interest.

In addition to the speed advantage of about 20:1 which Enhancement/Depletion mode Metal Semiconductor Field Effect Transistors (E/D-MESFETS) offer [IkToM84], it has been shown that GaAs has the ability to withstand environmental conditions far more severe than its silicon counterpart. In particular, GaAs can withstand 10-100 million RADS and a temperature range of -200°C to +200°C [EdLiW85]. This is important when environmental constraints are critical such as those found in aerospace and military applications where radiation hardness is a primary concern.

One final property which makes GaAs a more attractive material than silicon is its ability to interface directly with optical fibers [Honey85]. This technology has the potential to be used for communications processing by using optical fibers as inputs and outputs in the same fashion that current silicon microprocessors use electrical connections. This also has the potential to help alleviate the off-chip bandwidth restriction and reduce the interface hardware required.

1.2 The Motivation Behind the Emulation of a CISC

The three factors mentioned above make this technology an exciting area for research and further development for special purpose applications. Unfortunately, some processor designs are not directly implementable on a single GaAs chip because of the differences between silicon and GaAs device physics. These differences can be found in Table 1. In summary, the four major differences are: (1) the low transistor count allowable, (2) the large ratio of off-chip propagation delays to on-chip switching delays, (3) the low on-chip gate fanin (and fanout), and (4) the low yield. It is clear from this discussion that some architectures will never be viable for a direct GaAs implementation on a single chip.

There are three basic approaches available for a GaAs microprocessor. They include: (1) functionally divide the processor and implement each function as a

---------------

1. Although GaAs cannot simply be considered as a fast silicon, the transistor count is a relatively good indicator of architectural complexity and is therefore important.
separate chip, (2) divide the processor in a bit-slice manner and implement each slice as a chip, and (3) implement the entire processor as a single, very simple chip. Although the first two approaches may be worthy of further study, it is generally agreed that the excessive communications between processor elements is prohibitive in a GaAs environment. Any implementation of a GaAs processor with a multi-chip configuration will be plagued with a degraded performance due to large off-chip propagation delays [MilFu84].

Because of the aforementioned design considerations, the only architecture which is transistor count compatible with GaAs is known as a Reduced Instruction Set Computer (RISC) architecture. The RISC design philosophy consists of three basic design principles [Katev83]: (1) identify the machine instructions most frequently used by the target application, (2) optimize the datapath and timing for these frequent instructions, and (3) incorporate other frequent instructions into the instruction set only if they fit into the already elaborated VLSI scheme. Since RISC architectures are the only architecture in which a single chip computer can be implemented in GaAs, what needs to be shown is that the reduced instruction set primitives can fully emulate all of the elaborate capabilities of a Complex Instruction Set Computer (CISC) architecture.

It will be shown in this paper that a RISC can emulate a CISC with the exception of software constructs for hardware dependent capabilities such as dynamic memory sizing, nonstandard bus arbitration, etc. Of course, these constructs could be supported on new designs if they were needed by including the required hardware. It will also be shown that the most efficient method of emulation is accomplished by debugging compiled High-Level Language (HLL) source code on a CISC and then recompiling the debugged code into RISC machine code using a compiler that understands the resources available on the target RISC architecture. This technique will support the maximum execution-time speed up possible from GaAs technology.

1.3 Benefits Resulting from the Emulation

Several benefits exist as a direct consequence of the emulation of a CISC with a RISC architecture in GaAs. Four advantages are outlined below:

1. A possibility exists to replace CISC architectures where tolerance for excessive radiation or wide temperature variations is required.

2. A possibility exists to capture a market share of CISC type application programs in the general purpose computing arena where the execution speed improvement of a GaAs RISC is desired.

3. The emulation allows portability of CISC programs to the RISC environment to increase the number of application programs readily available to RISC architectures.

4. Software productivity is enhanced in speed critical environments because programs can first be debugged using CISC aids available in hardware (for example, trace mode) and then executed on a RISC architecture which achieves speed by reducing its overhead.
The only restriction is that the two architectures must have some basic similarities such as the same address space and the same data bus size. It is also advantageous to have the same memory map if I/O devices and coprocessors are memory mapped. If this is not the case, the I/O drivers will require some address translation or possibly, recoding.

EMULATION EFFICIENCY OF A CISC WITH A RISC

2.1 Problems Encountered in Emulation

Although it would appear advantageous to directly translate a program written using a CISC instruction set into one using a RISC instruction set, close scrutiny reveals that this endeavor is almost futile. The chief advantage of a RISC is the benefit of an execution-time decrease in machine code due to a shorter machine clock cycle [ClaSt80]. Unfortunately, this speed advantage can not be fully exploited when emulating a CISC using a machine code translation scheme. This is due to several architectural bottlenecks. The rest of this section is devoted to an explanation of these bottlenecks.

2.1.1 Addressing Mode Emulation

One of the architectural design philosophies in a RISC is the LOAD/STORE [2] concept of main memory addressing. A minimum number of off-chip operand references are used because of a large (and often times windowed) register file. This means that memory bandwidth is less of a restriction in this type of architecture than in most CISCs. When translation is considered however, the operand fetch latency time must be evaluated. This is especially true for CISC supported addressing modes.

Appendix A illustrates one example of an addressing mode emulation using the Stanford University MIPS (SU-MIPS) [PrGrH84] [GiGrH83] to emulate the Motorola 68020 (MC68020®) [Motor84]. The assembly code has been packed and optimized to allow a maximum utilization of hardware resources [Gross83] within the known context. The point of this emulation is that it can take as many as eight RISC instructions to do a simple arithmetic ADD if some indirect addressing modes are used. As a consequence, a RISC loses its speed advantage over a CISC implemented in silicon in because of the memory operand fetch latency.

--------

2. LOAD/STORE architectures minimize the operand fetch latency by prefetching all operands and storing them in registers. Since registers are generally the fastest level in the memory hierarchy, only a small operand fetch delay occurs compared with other levels of storage.
2.1.2 Condition Code Support

Although some RISC architectures, like the University of California at Berkeley RISC (UCB-RISC) [Katev83], support condition codes, many others do not. Instead, these RISCs often have a compare and branch instruction to simulate the two step process of testing a condition and executing a conditional action. This reduces the execution-time needed for a conditional action by a factor of two [3]. Even when a RISC architecture supports condition codes, support generally includes a rather small subset as compared with most CISCs.

Condition codes are set during the execution of virtually every instruction a CISC executes. Consequently, either an elaborate translation mechanism is required to check when the condition codes are actually used, or every instruction decomposition must synthesize the effect of the executed instruction on the condition code register. An example of a condition code synthesis is shown in Appendix B. As with examples used throughout this paper, the MC68020 and the SU-MIPS were used as the CISC/RISC pair. The generalized translation sequence assumes the existence of a sophisticated translator and occurs in a three step process as follows:

1. The translator locates all conditional instructions and retraces the job stream to determine the last instruction or instructions which set the condition code bits used in the decision. In situations where the condition code bits used in a conditional action do not lie within the confines of a labeled routine, any procedure that has a branch to that label must be checked.

2. The translator then replaces the conditional instruction with a corresponding compare and branch instruction if the RISC supports the condition tested and the sources for the comparison can be determined.

3. Alternately, the macros which set the required bits are inserted. This is the default when the translator cannot, for one reason or another, determine where the required condition code bits are set. An example of this is when a conditional branch vectors program flow based on an entry in a lookup table, and consequently, the destination procedure is indeterminant at translation time. Since the condition code bits required by the target procedure are also indeterminant, all bits in the condition code register must be appropriately set.

2.1.3 Trace Mode Emulation

Hardware costs have undergone a dramatic decrease in the last decade primarily due to advances in VLSI fabrication. As a result of the evolution of low cost integrated circuits, a large portion of project development costs lie in software development - especially during the program debugging phase. In

---

3. The overall execution-time actually increases by 1.1% when considering the entire program. The execution time is reduced only when considering an individual conditional instruction with an explicit test for condition.
response to this situation, hardware manufacturers are beginning to include a 
trace mode for program checkout and support for functions like setting program 
breakpoints, virtual machine support, etc. This concept may be well suited to 
the CISC philosophy but it violates the RISC philosophy of supporting only 
simple, fast instructions.

The software emulation of a trace mode is very expensive in terms of both 
exection-time and memory space. As a result, it has been ignored during the 
emulation process discussed in this paper. Instead it is assumed that program 
debugging will be done in a CISC environment and then recompiled for the RISC 
environment. The validity of this assumption lies in making hardware/software 
trade-offs between GaAs and silicon environments. Both RISCs and CISCs have 
specific advantages and therefore both types of architectures should be 
exploited. If it is advantageous to support a trace mode in a development 
environment, the complexity of this support should be migrated into silicon since 
it is known that a RISC's software is not efficient for this task. Moreover, in 
the case of the SU-MIPS, trace capabilities are virtually meaningless after code 
reorganization is complete because instruction boundaries are not always 
preserved.

2.1.4 Software Support of Infrequently Used Instructions

One of the main arguments heard from RISC proponents (in the CISC vs. 
RISC debate) is that compiler writers generally do not use the full capabilities 
of a complex instruction set [Wulf81]. This is either because the complex 
instruction has a longer execution-time than the corresponding primitives [4], or 
the complex instruction doesn't quite fit the particular need in all cases and 
extensive analysis would be required to determine when it makes sense to use it 
[5]. As a result, it is believed that a large portion of the complex instructions 
are not used except by assembly language programmers.

The question that arises from consideration of the emulation of infrequently 
used instructions is: if these particular instructions are used so infrequently, then 
why does it matter if they are execution-time expensive? The reason is that 
this paper analyzes the emulation as a whole. For example, the instruction 
decomposition for bit field operations is very execution-time expensive. This can 
be used to justify an external formatting processor which can provide data in 
the format requested rather than trying to do bit manipulations. Inclusion of 
this type of hardware has the additional advantage of normalizing a mantissa or 
stripping off the exponent from a floating point coprocessor. What is important 
in the design is the overall system architecture, not the individual pieces of it.

-------------

4. "Primitive" indicates one of the RISC instructions used to emulate a CISC 
instruction. The VAX autoincrement addressing mode is the classic example of a 
longer execution time for a CISC instruction.

5. For example, consider hardware support of a CASE statement or a 
PROCEDURE call. Invariably it either supports only one language well, or is so 
general that it is inefficient for special cases.
There is one problem associated with this concept, however. If the emulation assumes external processors, then how can the translator know when to synthesize execution-time expensive instructions and when to pass the job to another processor? The answer is simple. Make the translator portable much the same way the 'C' compiler was made portable. Allow a user definition of the back end of the translator so that it can be customized to several versions of the baseline architecture. This allows flexibility but does add somewhat of a burden to the translator writer. This is not however, an unreasonable requirement since there is a widespread belief that the full capabilities of GaAs technology will never be achieved without the appropriate advances in compiler (or in this case translator) technology.

2.2 A Solution to the Efficiency Problem

The previous section would indicate that an emulation of a CISC with any sort of efficiency is virtually impossible. That is not the case. The implication of the previous section is that a translation based emulation with any sort of efficiency is impossible. The obvious conclusion is that translation of assembly language instructions is not desirable unless it is absolutely necessary. As already indicated, the most efficient method of emulation consists of debugging compiled HLL source code on a CISC and then recompiling the code with a RISC compiler that understands the nature of the target architecture and can fully exploit the resources available to it.

Although recompiling HLL source code is not generally considered as an emulation, it is the most efficient approach. As long as it can be shown that an emulation is possible from a translation, the method employed to achieve an increased efficiency is unimportant. If a compiler based emulation is used to emulate the MC68020 with the SU-MIPS, then execution efficiencies greater than unity can be achieved as illustrated in Figure 1 and Figure 2. The eight pascal benchmarks used for this comparison are the following:

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Puzzle</td>
<td>Solves a three dimensional cube packing problem.</td>
</tr>
<tr>
<td>Queen</td>
<td>Solves eight queens chess problem.</td>
</tr>
<tr>
<td>Perm</td>
<td>Computes all permutations of one through seven.</td>
</tr>
<tr>
<td>Towers</td>
<td>Solves the Towers of Janoi problem for 14 discs.</td>
</tr>
<tr>
<td>Intmm</td>
<td>Multiplies two 40x40 integer matrices.</td>
</tr>
<tr>
<td>Bubble</td>
<td>Bubble sorts 5000 integers</td>
</tr>
<tr>
<td>Quick</td>
<td>Quicksorts 5000 integers</td>
</tr>
<tr>
<td>Tree</td>
<td>Tree insertion sort of 5000 integers.</td>
</tr>
</tbody>
</table>

When a translation based emulation is used for the same two processors, the execution efficiency is reduced and storage space in increased. The extent of these effects can be determined for the Quicksort benchmark as shown in Figure 3; the actual code translation can be found in Appendix D. This translation was
done for MC68020 code written directly in assembly language and is compared with SU-MIPS code also written in assembly language so that compiler technology does not influence the performance evaluation. From these figures it is clear that whether GaAs or silicon is the material of choice for the RISC, compiler based emulation is a more attractive alternative by far.

Unfortunately, there are situations when a direct translation is the only alternative for emulation. A translation is required, for example, when the only source code is machine code that can be disassembled into assembly code and then translated into RISC assembly code. This does not include system monitors or I/O drivers for the most part. Although these routines are written in assembly code because of their speed critical nature, they are generally architecturally dependent and a translation is meaningless. For example, a RISC in general will not have the same hardware controls such as the interrupt mechanisms, nor will it have the same memory mapping of its I/O ports. Since the translator cannot anticipate these situations, it does not make sense to perform a translation. Perhaps more importantly though, the key words for consideration are "speed critical nature." It has already been shown that an efficient translation cannot be achieved so it is impossible to make the translated routines fast.

THE TRANSLATOR'S SOPHISTICATION

3.1 Condition Code Optimization

In subsection 2.1.2, the steps in condition code emulation were briefly discussed. This section will further elaborate on these problems. The reason so much attention is being given to this problem is because this has the potential of being a major execution-time bottleneck. Fortunately, with a sophisticated translator some of the speed degradation which this problem imposes can be eliminated. The ensuing subsections will outline the solution to the partial elimination of this problem.

3.1.1 Locating Conditional Instructions

The dominant execution-time inefficiency when attempting to synthesize condition codes results from trying to emulate them for every instruction in the CISC instruction set which affects the status register. The first step in minimizing the unnecessary overhead is to reduce the number of instances where the synthesis occurs. The way this is accomplished is by finding a conditional action and then retracing the sequential execution until the last instruction (or instructions) that affect the condition codes used in the conditional action are isolated. For most instructions, this will not be difficult. For instructions which have labels in close proximity to the conditional action, this may not be the case. Whenever a program segment does not have clear cut direction which it is destined to follow, all of its condition codes must be synthesized regardless of whether they are known to be used or not. This can be very execution-time expensive.

Consider translating code written at the assembly level. At one point or another, most assembly language programmers have written a program where
branches occur to a program segment based on an entry in a lookup table which may or may not be known at assembly time. In this event, there is no way for the translator to anticipate the direction of program flow and consequently the last instruction or instructions prior to a branch must set the condition codes for all bits that are affected by the job stream. As mentioned earlier, the complete emulation of condition codes is disastrous from an efficiency standpoint. The conclusion from this is that the existence of an efficient translation for code written at the assembly level is often dubious. As already mentioned, the best method of assuring efficiency is using well structured HLL programs which are compiled into well structured assembly language statements using modern compiler methods.

3.1.2 Substituting RISC Supported Conditional Constructs

Although the substitution of MIPS supported conditional actions is not addressed in Appendix B, its discussion is appropriate. What this process entails is locating instructions where a conditional action such as "Branch on Equal" or "Set on Positive" can be performed equivalently by a MIPS primitive and the two operands can be identified. When this is the case, the condition code synthesis can be completely ignored and an efficient translation can be supported by replacing CISC instructions with RISC primitives.

How often does the situation occur where a RISC's primitives can replace a CISC's conditional action and the macros which set the condition codes? In practice, this situation occurs quite frequently as shown in Appendix D. The condition codes are usually set in the instruction directly preceding the conditional action. When this occurs, the two instructions are prime candidates for single instruction replacement. One of the few examples of a conditional action that can never be replaced with a MIPS conditional action is a ROTATE or a SHIFT instruction. Fortunately, the decomposition of these instructions include primitives which set the appropriate bits in the condition code register every time they are synthesized. This generally allows the macros which are listed in Appendix B.3 to handle these cases without the aid of the condition code macros for the other affected bits.

The conclusion of this subsection is that it is often possible to salvage some efficiency in a majority of the cases of conditional actions. The practical implementation of this type of synthesis depends heavily on the structure of the assembly language program, but compilers today generate relatively well structured code. Although the best method of emulation results from compiling HLL constructs directly into RISC machine instructions, it is encouraging to realize that in cases where that is impossible a translation can still be performed with some amount of efficiency in condition code synthesis.

3.1.3 A Default Solution to Condition Code Synthesis

The conditional actions of a CISC cannot always be replaced by a RISC's primitives. The macros for the individual bits of the condition code cannot always be separated either. What can be done for the sake of efficiency about the situation where for one reason or another it cannot be determined what should be done about the condition codes? Unfortunately, the only solution is to synthesize all bits affected by the sequence of instructions in question.
Consider the case of a subroutine that has a conditional branch a few instructions into the routine. Naturally, this subroutine has a label associated with its entry point. This means that any of a number of other branches or calls can reference this entry point as a destination. Since it is sometimes indeterminant at compile time which calls or branches will vector execution to this subroutine, all routines which have its entry point as a possible destination may be forced to set the condition codes for the bits used in the destination subroutine's conditional branch. Although this is not an attractive situation, it is unavoidable.

3.2 The Impact of the Data Format on Efficiency

Another concern related to improving execution-time efficiency of emulation, is the data format. More precisely, most CISCs support formats other than binary arithmetic. For example, the MC68020 supports Binary Coded Decimal (BCD) arithmetic for many types of operations. It would be advantageous from an efficiency standpoint to support only those types of formats which the architecture supports; in the case of a RISC, this is only binary arithmetic.

Consider the task of adding three BCD numbers located in registers and storing the result back into another register. The translation sequence for this would be:

```
ABCD RS1,RD <=> UNPK RS1,RS1
    UNPK RD,RD
    Add RS1,RD
    PACK RS1,RS1
    PACK RD,RD
```

```
ABCD RS2,RD <=> UNPK RS2,RS2
    UNPK RD,RD
    Add RS2,RD
    PACK RS2,RS2
    PACK RD,RD
```

In this example, the capitalized instructions are MC68020 instructions used in the translation and the lower case instructions are MIPS primitives. This example illustrates the generalized method employed in translation but this is clearly an inefficient method. Obviously the first step in optimizing this code is to remove the instructions "PACK RD" and "UNPK RD" in the middle of the sequence.

The next level of improvement is by totally removing all of the PACK and UNPK operations. Although in a CISC the operands may be resident in main memory instead of registers, this is not true in LOAD/STORE architectures. Because of this, the most practical method of improvement is to realize that the operand is in a BCD format when it is fetched. If it is immediately translated into a binary number and left in the register file in that format, no translation has to be performed until it is written back into main memory. This will improve performance for frequently used data items which are used in repeated iterations of arithmetic operations.
3.3 The Effect of Increasing the Register File Size

One notable inconsistency in the addressing mode emulation can be found in the register allocation scheme. In Appendix A.1, 24 registers are listed for use by the translator. The 24 registers specified are all required if there is to be no sacrifice in the number of registers available to the MC68020 compiler. Since the original SU-MIPS has only 16 general purpose registers, this means one of two things: either a lot of register/memory swapping must be done for the least frequently used registers to accommodate the translation between architectures, or the register file of the SU-MIPS must be enlarged to a minimum of 24 registers. Increasing the register file size will impact the instruction format by enlarging the register field 1-bit per field. Both of these will be discussed in more detail below.

3.3.1 Decreasing the LOAD/STORE Latency

If swapping between the register file and memory is the required alternative, this will change the somewhat optimistic emulation efficiency because of the additional wait states introduced from spill over LOAD/STORE operations. In the silicon environment the addition of eight 32-bit registers would be inconsequential; in the GaAs environment this would not be the case. This assertion is based on the design parametrics of the original SU-MIPS. The original register file size was approximately 21% of the total number of transistors and about 8.3% of the VLSI area [PrGrH84]. If 50% more registers and the control circuitry to support them were added, the transistor count would be over 8.0K. If 100% more registers were added to support 32, the device count would exceed 11K transistors. More importantly, with today's GaAs technology, the data path area might be prohibitive unless one the the register data busses were excluded.

However, within the next decade or so, GaAs technology is expected to support 60K transistors. Because of the off-chip communication bottleneck, it is believed that the additional transistor count should be used to exploit a larger register file or a register windowing scheme. An example of the benefits of register windowing can be found in the examination of the UCB-RISC where dramatic execution-time decreases were obtained from an eight window register scheme [Katev83]. If a translator were designed to minimize the off-chip memory references by keeping more of its operands in the enlarged register file, this could overshadow other available alternatives for increasing execution speed. Examples of alternatives which improve performance include additional hardware resources such as multiple Arithmetic Logic Units (ALUs) or serial on-chip multipliers [6].

6. In digital signal processing there is an advantage to these types of resources because of the nature of signal processing. This may outweigh the advantages of the additional registers [Tseng84].
3.3.2 The Impact on the Instruction Format

The instruction format would require minor modifications to allow the addition of 16 registers in the silicon environment [7]. The major difference would be the elimination of ALU3 instructions [8]. Fortunately, an examination of the Appendices reveals that the number of ALU3 instructions is relatively small. If an ALU3 instruction were required, it could be synthesized by moving one of the source operands into the destination register and then executing an ALU2 instruction. Alternately, it could simply be included in the instruction set in its current form with the provision that it cannot be packed with an ALU2 instruction.

A more important consideration than the elimination of the ALU3 instruction is the inability for the modified instruction format to support packing LOAD instructions with ALU2 instructions. One proposed method to help reduce the impact of this modification, is the introduction of two new instructions known as LD2 or ST2 which could be packed with an ALU2 instruction. Any of the following methods could be used:

1. A LD2 (ST2) instruction could have two registers where one register is the address of the source (destination) operand and the other register is the destination (source). No offset would be allowed for the address register.

2. A LD2 instruction could have two registers or a register and an immediate. The two sources would be added to form the effective address of the operand with the result written to the second source.

3. A LD2 (ST2) instruction could have two registers or a register and an immediate with an implicit destination (source) register that is always used.

All existing three operand formats could still be supported if that were desired but they could not be packed with an ALU2 instruction. One additional benefit which the enlarged register field allows is the expansion of the immediate constant to any number in the range of -16 to +15. This can be very beneficial for the emulation of condition codes where the limitations of a four bit immediate field resulted in excessive ALU3 MOV instructions.

7. In the GaAs environment, packing will not be allowed so these modifications to the instruction format will not be required.

8. An ALU3 instruction is so named because it has three operand fields: (1) source 1, (2) source 2, and (3) destination.
4.1 Why Pipelining is an Important Consideration

The pipeline depth can have a dramatic impact on the emulation efficiency when translating CISC instructions. For example, a delayed branch scheme is very attractive in silicon because the probability of finding an instruction to replace the NO-OP directly after the branch has been estimated to be as high as 90% [Gross83b]. Unfortunately, delayed branch schemes lose some of their impact in the GaAs environment with a long pipe. When the branch delay is extended to six machine cycles due to the instruction fetch time, the replacement probability for all five NO-OPs is less than 5%. The probability of NO-OP replacement for pipe depths between two and five approximately follows a logarithmic curve between the two. A similar analogy exists for an operand fetch which has the same memory latency.

What this implies is that it is very difficult to support elaborate addressing modes or excessive branching. The only hope of overcoming these problems is the possibility of an off-chip, on-package cache split between instruction storage and data storage. If these can be made large enough to ensure a reasonable hit probability, then it is possible to bring the average memory fetch latency down to two machine cycles and obtain a closer match to the silicon SU-MIPS architecture. This will be discussed in more detail in section 6.1.

4.2 Pipelining in the Silicon Environment

The SU-MIPS pipeline scheme and instruction cycle time was optimized to match the silicon operand fetch time [Gross83b]. This includes the use of a five stage pipe with the following sequence:

(IF) Instruction Fetch - Send out the Program Counter (PC) and increment it.
(ID) Instruction Decode - Decode the requested instruction.
(OD) Operand Decode - Compute the effective address and send to memory if a LOAD or a STORE; alternately, use the ALU for a register to register ALU operation.
(SX) Operand Store - Send out the operand if a STORE operation; alternately, use the ALU for a register to register ALU operation.
(OF) Operand Fetch - Receive the operand if a LOAD and write it to a register.

This sequence lends itself well to the silicon environment because the operand or instruction fetch time corresponds to only twice the instruction execution-time. Consequently, the pipeline overlap was designed to fetch the
next instruction every two machine cycles. The result is a good balance between the datapath time, ALU time, and memory access time. With efficient packing, delayed branches, and a large register file to reduce off-chip operand references, an efficient use of the hardware resources is possible. Even when the emulation requires support of elaborate addressing modes, it is not prohibitive in the silicon environment because it will only take two clock cycles to fetch a memory operand.

4.3 Pipelining in the GaAs Environment

The GaAs environment is radically different than the silicon environment for several reasons. The most troublesome, as mentioned before, is the tremendous bottleneck of off-package communications. This makes many of the advantageous mechanisms in silicon difficult to implement in GaAs. One of these is the pipeline mechanism because the instruction execution-time is an order of magnitude faster than the memory fetch in GaAs time as compared to only twice as fast in silicon. This changes the pipeline implementation although it does not force much of a difference in the particular sequence of events which occurs. One possible pipeline scheme in the GaAs environment is shown below:

(IF) Instruction Fetch - Send out the PC, increment it, and decode the current instruction.

(REG1) Register 1 Read - Read the first operand from a register.

(REG2) Register 2 Read - Read the second operand from a register.

(ALU) Arithmetic or logic operations on the operands to produce an arithmetic result, calculate an effective address, or perform a comparison.

(MEM) Memory Address - Send out an effective memory address.

(WB) Write Back - Send out the results of an ALU operation for a memory or register write, or fetch the operand for a memory read.

This pipeline scheme is designed to consist of five stages where REG1 and REG2 were separated to emphasize that they are fetched sequentially. The sequential fetch makes sense for two reasons. First, an attempt is being made to get an optimum match between the datapath time, memory fetch time, and the instruction fetch time. Secondly, two read busses are expensive in VLSI real estate. This is not a problem in silicon but is a problem in GaAs. Since the extra bus could not be fully utilized because of the limited memory bandwidth, it makes more sense to use the VLSI area saved from using only one bus for an enlarged register file. This has all of the benefits listed in section 3.3.
5.1 Intrainstruction Packing

In the SU-MIPS architecture, there is no pipeline interlocking scheme [9]. Instead of hardware controls which prevent pipeline hazards and conflicts, the Stanford architects decided to use software interlocking [HeJoP83]. The advantage of this scheme is that the hardware can generally achieve a higher throughput because an intelligent compiler can frequently put an unrelated instruction in the job stream which can execute while the hazard is being avoided. Of course, this is not always the case. One example of this is discovered when attempting to pack an addressing mode decomposition with an instruction decomposition.

Because of software interlocks, it would appear that there would be an advantage in combining each of the eighteen MC68020 addressing modes with every instruction decomposition to determine the overall execution efficiency. Upon closer examination, this proves not to be the case. Whenever the next primitive is dependent on the data being fetched, a situation known as a data dependency exists and intrainstruction packing is usually prevented. This implies that the NO-OPs following the final LOAD instruction in an addressing mode decomposition cannot be optimized out.

5.2 Interinstruction Packing

Now consider packing on an interinstruction basis. Although it was shown that data dependencies often prevent packing on an intrainstruction basis, packing sequential addressing mode primitives with previous instruction primitives can be done for a large majority of cases [10].

Unfortunately, there is one problem associated with this type of packing. If a particular addressing mode decomposition has a label associated with it, packing is not always possible on an interinstruction basis with the instruction primitives preceding it. The reason is that if primitives from a previous instruction are packed with the primitives of the next addressing mode, then any routine that jumps or calls that instruction will execute code that could have adverse affects of the execution of the program. For example, if the last primitive in the decomposition was a register ADD and it was packed into the next sequential addressing mode decomposition which happened to have a label, then the destination register of the ADD would always be modified whenever a branch was taken to that label. This cannot always be allowed.

---------

9. MIPS was derived from "Microprocessor without Interlocked Pipeline Stages."

10. An interesting consequence of this fact is that the efficiency of emulation cannot accurately be determined without an actual translation and execution since the interinstruction packing probability is program dependent.
6.1 On-Package Cache Support

The significance of an on-package, off-chip cache on the execution-time performance cannot be overemphasized. Since the cache affects the pipeline depth it is extremely important; it can significantly decrease the average pipe fetch latency. Cache basically comes in two storage types, Instruction (I) cache and Data (D) cache. Both types have tremendous advantages in the GaAs environment.

Significant execution-time improvements can be attained from a cache with as little as 4K bits of storage [HwaBr84]. For example, a hit ratio of over 50% can be expected from an I cache with 64 32-bit words [SmiGo85]. Since memories of this size have already been fabricated in GaAs [HilnM84], this is not unreasonable to consider. The I cache should be large enough to hold a typical size program loop; in the MC68020, this was chosen so that 64 instructions can be kept on-package. Given the 4K limitation, a modest D cache could also be implemented in GaAs with 64 32-bit words. These can be used to keep many of the most frequently used instructions or operands of a program only two machine cycles away.

If a cache is available, then support for a four level memory hierarchy must also exist. The four levels are:

1. Register operands available with virtually no fetch latency.
2. Cache operands available within two machine cycles on a cache hit.
3. Memory operands available within five machine cycles.
4. Mass storage that has an access time which is device dependent.

Assuming a reasonable cache hit ratio, many of the memory references should statistically reside in the first two levels of memory hierarchy as shown in Figure 4. This will enhance the efficiency of emulation because when attempting to translate CISC instructions it is advantageous (from an architectural point of view) to have RISC hardware with characteristics as close as possible.

One of the possible implementations for a cache would be to use software interlocks for operands which are fetched from the cache and hardware interlocks for operands which are fetched on a cache miss. If this scheme is used, then a GaAs MIPS would be able to use a two stage pipeline identical to the SU-MIPS. The conclusion that can be drawn from an implementation of cache in GaAs is that the closer the operands are to the ALU, the better the efficiency will be.
6.2 Coprocessor Support

One aspect of the instruction set synthesis which was not addressed in Appendix C is the extensive coprocessor instruction set which the MC68020 supports. The primary reason for this omission is that most RISCs don't have coprocessor support and it is tedious at best to emulate floating point instructions. Another reason is that with an instruction cache on-package, it is very difficult for the coprocessor to monitor the instruction bus to ascertain which instruction the microprocessor is executing at any given point in time [11]. Finally, some types of instructions, such as a conditional action based on a coprocessor result, are almost prohibitive in the GaAs environment because of the difference in the instruction cycle time and the off-chip signal propagation. In this section, we will attempt to address these questions.

6.2.1 Monitoring the Instruction Bus

The problem with trying to monitor the instruction bus is that when an on-package cache is present the internal instruction bus is usually hidden from external peripherals. In general, only the Cache Direct Memory Access (CDMA) controller has access to both the internal and external busses. Because of this, the CDMA is the only device capable of granting the coprocessor access to the currently executing instruction.

One proposal for handling coprocessors is based on the CDMA sending a requested instruction to both the microprocessor and the external instruction bus in the event of a cache hit. In the event of a cache miss, the CDMA must request the instruction from external memory anyway, so it is available to the coprocessor immediately. One drawback of this proposal is that it does prohibit CDMA prefetching of the next sequential instruction to be executed from main memory [12] and it does increase bus contention. In most applications that would require a coprocessor however, these drawbacks are offset by the advantages of coprocessor support.

6.2.2 Conditional Actions Based on Coprocessor Conditions

For an existing RISC there are two predominant methods for supporting a conditional action based on a coprocessor condition. One employs a memory location used as a status register to support a software interlock which is set to TRUE when a memory mapped coprocessor register becomes valid. The other method uses a memory location as a data register but is interrupt driven and causes an interrupt when the data becomes valid by asserting the interrupt pin. If the interrupt method is used to indicate that the condition has been tested, the validity of the test result is found as soon as the interrupt vector is placed on the instruction bus; this is the most efficient method.

11. This is the method most frequently used in coprocessors today.

12. This applies most directly to CDMA's with a remote PC to indicate the location of the next instruction and/or a "likely bit" to indicate the probability of a branch [PaGaH83].
Fortunately, on new RISC designs there are many coprocessor support options available for implementation with regard to flags being tested and set. In addition to the methods described above, a separate set of pins could be included to indicate the validity of the result, and optionally, the result of the test. The constraining factor is the number of pins which are available due to packaging considerations.

6.2.3 An Operand Transfer Scheme

The number of internal registers available to the coprocessor is best left transparent to the RISC unless a particular coprocessor is being designed to be integrated with the overall architecture or it is decided that only a particular coprocessor will be supported. Therefore, it makes no difference whether the coprocessor is a stack machine or a register machine; rather, what is important is that the operands are readily available to the RISC.

The most frequent solution to the operand transfer question is based on a memory mapped register scheme. This scheme requires that the values that are contained in the registers used by the coprocessor are reflected in the contents of main memory. Memory should be updated to reflect any changes in the internal registers in a "write through" fashion similar to a data cache by stealing processor memory cycles. If, for example, the instruction on the instruction bus is a coprocessor instruction, then any of the mechanisms described in the previous subsection can be used to indicate the validity of the result as long as the coprocessor has written its result to a memory location prior to the microprocessor requesting it. If all memory references are handled through a memory controller, then the exponent can be stripped or the mantissa normalized during the operand fetch. This will be discussed in more detail in the next section.

6.3 Bit and Byte Field Support

As indicated in other sections, because of the limitations in VLSI area which a GaAs design can exploit, any migration of CISC operations into peripheral chips has the potential to improve execution efficiency. This is only true for formatting preprocessors if the peripheral processor doesn't impact the memory fetch latency. Even though bit and byte field operations may be infrequent, if they are expensive enough relative to the overall instruction translation it could prove beneficial to add a peripheral chip with formatting capabilities.

For instance, if a data format chip increased the operand fetch latency for every operand fetched by two machine cycles, then it would not make sense for this chip to be placed in the memory path if most memory requests consisted of 32-bit word aligned operands. However, if a chip were designed which acted much like a DMA controller and only increased the memory latency when a request for a formatted memory operand or coprocessor operand was made, it could be a valuable addition to the architecture. Appendix C illustrates the expense of trying to do bit and byte alignment in software for a CISC/RISC pair.
INSTRUCTION TRANSLATION

7.1 Translation From MC68020 Code to MIPS Code

Just as it is assumed that a new hardware technology will soon be available for commercial use, so too is it assumed that a new software technology will be available. This assumption is in fact almost necessitated in the hope that some amount of emulation efficiency is salvageable. The belief in a new software technology is twofold in justification. First, it is substantiated by the large development effort in the areas of intelligent compilers and in other types of artificially intelligent software. Secondly, it is substantiated by the general interest in RISC architectures as a whole - an architecture which emphasizes the role of an intelligent compiler as an integral part of the architecture.

So far, the individual pieces of the translation have been analyzed without dealing with the actual task at hand. Now, elaboration on the actual translation scheme employed in getting from MC68020 machine code to MIPS machine code is needed. The emulation has been described as consisting of modular, disjoint tasks which could be integrated into a single program that is capable of the translation. The goal now is to combine all the pieces together and comment on their various interactions. This process begins with raw MC68020 machine code and ends in optimized MIPS machine code.

7.2 Disassembly and Decomposition of the MC68020 Instructions

The first step of translation is to disassemble the MC68020 machine code into assembly level code so that it can be dealt with in a symbolic manner. Disassemblers have already been written for this processor so little elaboration about this point is required. The thrust of the effort here is in assigning labels to various parts of the program and in decoding operand lengths.

The next step in emulation consists of decomposing the disassembled instructions into their MIPS primitives. This will include the addressing mode decomposition outlined in Appendix A, followed with the operation decomposition outlined in Appendix C. The MC68020 machine code address will be associated with the first primitive of the addressing mode decomposition. The condition code synthesis will then be inserted into the job stream without regard to whether or not the status register bits are used; this decomposition is outlined in Appendix B. The MIPS supported constructs will however, replace their MC68020 counterparts where appropriate.

Very little optimization will be performed at this point in the translation with the possible exception of register reassignment for global optimization and any intratruction packing that is possible. If a register file larger than 24 registers were available, then the register reassignment would attempt to assign variables in such a manner as to reduce the amount of LOAD/STORE instructions required. The instruction boundaries will be preserved until after the condition codes have been analyzed and removed where possible. This will prevent optimizations for code that is destined to be removed.
7.3 Condition Code Analysis and Interinstruction Optimization

Condition code analysis and optimization will occur next in the sequence as per specifications outlined in section 3.1. This is one of the most lengthy optimization steps in the sequence. One thing which was not previously discussed is the concept of preserving the condition code boundaries to ensure that the synthesis is not prevented from altering the required bits in the status register at the proper point during execution.

The next module of translation handles interinstruction packing of MIPS primitives. This task was outlined in more detail in section 5.2. Care must be taken to preserve instruction, subroutine, and condition code boundaries where required. For example, most RISC primitives cannot be packed with a preceding decomposition's primitives if it is a subroutine boundary in question. This is because it is generally not desirable to execute primitives from a preceding instruction decomposition every time a branch is taken to the subroutine.

The preservation of the subroutine boundary is frequently not required in the case of condition code synthesis, however. The condition code synthesis only modifies registers R19 and R20. Since it does not matter if these registers are modified throughout the subroutine, it is inconsequential if the condition code synthesis crosses the routine's boundary as long as the synthesis is completed before the affected bits are reset by another instruction or a conditional action is reached.

7.4 Standard MIPS Optimization Mechanisms

The remainder of the translation mechanism will rely on the standard MIPS optimizer and assembler with some minor differences. The key difference is that the optimizer will only be allowed to operate on register reassignment and delayed branching NO-OP replacement. Subroutine, procedure, and condition code boundaries must be preserved since the optimizer and reorganizer are oblivious to the translator's overall purpose. The following is the order which this optimization will occur.

1. Reorganize of the MIPS machine code to avoid pipeline and branch hazards.

2. Pack the MIPS machine code into the most compact form allowed under the imposed boundary constraints.

3. Assembly of the resulting instruction translation into MIPS machine code which is "hidden" from the end user.

Although certain portions of the standard MIPS optimization scheme could be combined with other modules of the translation, it does not make sense to rewrite any more of the existing software than is absolutely necessary. Following that philosophy, an attempt has been made to reuse as much existing software as possible. It is generally easier to modify a piece of software than it is to rewrite a new piece and be faced with a new set of problems
altogether. In addition, following the conclusions of this paper, it is assumed that a software translation will be avoided when possible so starting from scratch is not justified because of the development effort that would require.

CONCLUSION

This paper analyzed the architectural considerations concerning the emulation of a CISC with a RISC. It was shown that it is possible to emulate the constructs supported by a CISC except for some software constructs which are used to control the hardware. The MC68020 and the SU-MIPS were selected as the comparison architectures to help exemplify the important concepts of a translation; their selection was based on the current interest in both of these microprocessors. It was further shown that a poor emulation efficiency is expected from a direct translation of the MC68020 instructions into MIPS primitives because of the nature of the MC68020 instructions.

In addition, several comments were made on the translation from CISC to RISC code with regard to the design of GaAs fabricated architectures. The most important differences between GaAs and silicon architectures is due to device physics; specifically, the speed limitations in GaAs are due to off-chip communications bottlenecks. GaAs enjoys an order of magnitude decrease over silicon in its machine cycle time but there is virtually no speed advantage when going off-package. As a result, many of the accepted constructs of CISC architectures are not transferrable into GaAs if a RISC architecture is involved. When CISC concepts are implemented on a RISC design, they violate the philosophy that gives a RISC its speed advantage.

It is therefore no surprise that virtually no execution-time improvement can be realized in a translation because the CISC's compiled code uses constructs and resources unavailable or inconsistent with the RISC's. The resulting inefficiency may be tolerable because of the radiation hard advantages in a GaAs RISC, but it was concluded that the most efficient method of emulation could be achieved by first debugging HLL source code on a CISC using all of the development hardware available and then recompiling the source code with a compiler that is aware of the resources available to the RISC. This allows the exploitation of the advantages of both types of architectures; namely, the flexibility of a CISC and the speed of a RISC. Furthermore, using a RISC to execute compiled code instead of translated code allows emulation efficiencies much greater than unity.

ACKNOWLEDGEMENTS

The authors are thankful to their colleagues from the Purdue University GaAs project for their useful comments and suggestions and to RCA's Advanced Technologies Center for their guidance in this research.
REFERENCES


[Honey85] Lee, T.C., EE694 Graduate Engineering Seminar, Purdue University, Spring 1985.


<table>
<thead>
<tr>
<th></th>
<th>GaAs</th>
<th>Silicon</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>COMPLEXITY</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Transistor Count/Chip</td>
<td>20-30K</td>
<td>300-400K</td>
</tr>
<tr>
<td>Chip Area</td>
<td>yield and power dependent</td>
<td>yield and power dependent</td>
</tr>
<tr>
<td><strong>SPEED</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Gate Delay</td>
<td>50-150 ps</td>
<td>1-3 ns</td>
</tr>
<tr>
<td>On-chip Memory Access</td>
<td>0.5-2.0 ns</td>
<td>10-20 ns</td>
</tr>
<tr>
<td>Off-chip/On-package Access</td>
<td>4-10 ns</td>
<td>40-80 ns</td>
</tr>
<tr>
<td>Off-chip/Off-package Access</td>
<td>20-80 ns</td>
<td>100-200 ns</td>
</tr>
<tr>
<td><strong>IC DESIGN</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Transistors/Gate</td>
<td>1 + fanin</td>
<td>1 + fanin</td>
</tr>
<tr>
<td>Transistors/Memory Cell</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Static</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>Dynamic</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Fanin (typical transistor)</td>
<td>3 - 4</td>
<td>5</td>
</tr>
<tr>
<td>Fanout (typical transistor)</td>
<td>3 - 4</td>
<td>5</td>
</tr>
<tr>
<td>Fanout Gate Delay (ea. gate)</td>
<td>25 - 40 %</td>
<td>25 - 40 %</td>
</tr>
</tbody>
</table>
Figure 1. SU-MIPS (4 MHz.) EXECUTION EFFICIENCY FOR DIFFERENT COMPILED BENCHMARKS [PrGrH84]
Figure 2. MC68020 (10 MHz.) EXECUTION EFFICIENCY FOR DIFFERENT COMPILED BENCHMARKS

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUZZLE</td>
<td>4.0</td>
</tr>
<tr>
<td>QUEEN</td>
<td>1.0</td>
</tr>
<tr>
<td>PERM</td>
<td></td>
</tr>
<tr>
<td>TOWERS</td>
<td></td>
</tr>
<tr>
<td>INTMM</td>
<td></td>
</tr>
<tr>
<td>BUBBLE</td>
<td></td>
</tr>
<tr>
<td>QUICK</td>
<td></td>
</tr>
<tr>
<td>TREE</td>
<td>7.0</td>
</tr>
</tbody>
</table>
Figure 3. COMPARISON OF EXECUTION EFFICIENCIES AND CODE SIZE FOR DIFFERENT METHODS OF CODE GENERATION
Figure 4. CACHE HIT RATIOS VERSUS CACHE SIZE

HIT RATIO (%)

DATA CACHE

INSTRUCTION CACHE

CACHE SIZE IN 32-BIT WORDS
### A.1 Register Allocation

<table>
<thead>
<tr>
<th>MIPS</th>
<th>FUNCTION</th>
<th>MC68020</th>
<th>BITS</th>
</tr>
</thead>
<tbody>
<tr>
<td>R0-R7</td>
<td>(Data registers)</td>
<td>D0-D7</td>
<td>32</td>
</tr>
<tr>
<td>R8-R14</td>
<td>(Address registers)</td>
<td>A0-A6</td>
<td>32</td>
</tr>
<tr>
<td>R15</td>
<td>(User stack pointer)</td>
<td>A7</td>
<td>32</td>
</tr>
<tr>
<td>PC</td>
<td>(Program counter)</td>
<td>PC</td>
<td>32</td>
</tr>
<tr>
<td>R16</td>
<td>(Status &amp; cond code)</td>
<td>SR</td>
<td>16</td>
</tr>
<tr>
<td>R17</td>
<td>(Interrupt stack pointer)</td>
<td>A7'</td>
<td>32</td>
</tr>
<tr>
<td>R18</td>
<td>(Master stack pointer)</td>
<td>A7&quot;</td>
<td>32</td>
</tr>
<tr>
<td>[13]</td>
<td>(Vector base register)</td>
<td>VBR</td>
<td>32</td>
</tr>
<tr>
<td>[14]</td>
<td>(Alternate function)</td>
<td>SFC</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>&quot;</td>
<td>DFC</td>
<td>3</td>
</tr>
<tr>
<td>[15]</td>
<td>(Cache control register)</td>
<td>CACR</td>
<td>32</td>
</tr>
<tr>
<td></td>
<td>(Cache address register)</td>
<td>CAAR</td>
<td>32</td>
</tr>
<tr>
<td>R19-R23</td>
<td>(Scratchpad registers)</td>
<td>[16]</td>
<td>32</td>
</tr>
</tbody>
</table>

13. The default address of for exception handling in MIPS is always zero, therefore VBR is not required.

14. These are not supported by MIPS.

15. Since MIPS has built in (non-optional) cache support, these registers are not required.

16. MIPS will require additional scratchpad registers for address calculations that the MC68020 doesn't require.
A.2 Addressing modes supported by the MC68020

1. Data Register Direct [17]
2. Address Register Direct
3. Address Register Indirect
4. Address Register Indirect with Postincrement
5. Address Register Indirect with Predecrement
6. Address Register Indirect with Displacement
7. Address Register Indirect with Index (8 Bit Displacement)
8. Address Register Indirect with Index (Base Displacement)
9. Memory Indirect Post-Indexed
10. Memory Indirect Pre-Indexed
11. PC Indirect with Displacement [18]
12. PC Indirect with Index (8 Bit Displacement)
13. PC Indirect with Index (Base Displacement)
14. PC Memory Indirect Post-Indexed
15. PC Memory Indirect Pre-Indexed
16. Absolute Short Address
17. Absolute Long Address
18. Immediate

A.3 Register Definitions used in Translation

Alo
An
Ahi
Bd
DISP
Od
SCALE
SIZE
Xn

Low order bits of a physical address (24-bits)
Any address register
High order bits of a physical address (8-bits)
Base Displacement (sign extended to 32-bits)
Any 8-bit displacement
Outer Displacement (sign extended to 32-bits)
Constant equal to 0,2,4, or 8
Constant equal to 1,2, or 4
Any register used as an index register (sign extended to 32-bits)

17. Modes 1,2,16, and 18 are directly supported by MIPS.

18. PC relative modes will need some modifications because the translation will change the relative location of some of the operands.
A.4 Addressing Mode Translation

**MODE 1:** 
Directly supported by MIPS.

**MODE 2:** 
Directly supported by MIPS

**MODE 3:**
```
Ld 0[An],R23 ;;Mov An,R21
No-op
```

**MODE 4:**
```
Ld 0[An],R23 ;;Mov An,R21
Add SIZE,An
```

**MODE 5:**
```
Ld -SIZE[An],R23 ;;Sub SIZE,An
Mov An,R21
```

**MODE 6:**
```
Ld Bd,R21
Ld [An+R21],R23 ;;Add An,R21
No-op
```

**MODE 7:**
```
Mov DISP,R22 ;;Mov Xn,R21
Sll SCALE,R21 ;;Add R22,R21
Ld [An+R21],R23 ;;Add An,R21
No-op
```

**MODE 8:**
```
Ld Bd,R22
Mov Xn,R21 ;;Sll SCALE,R21
Add R22,R21 ;;Add An,R21
Ld 0[R21],R23 ;;No-op
No-op
```

**MODE 9:**
```
Ld Bd,R21
Ld [An+R21],R22 ;;Mov Xn,R21
Sll SCALE,R21 ;;Add R22,R21
Ld Od,R22
Ld [R21+R22],R23 ;;Add R22,R21
No-op
```

**MODE 10:**
```
Ld Bd,R22
Mov Xn,R21 ;;Sll SCALE,R21
Add An,R21 ;;No-op
Ld [R21+R22],R21 ;;No-op
Ld Od,R22
Ld [R21+R22],R23 ;;Add R22,R21
No-op
```

- 30 -
MODE 11:
Ld  Bd-2,R22
Mov  PC,R21
Ld  0[R21],R23
No-op

;;Add  R22,R21
;;No-op

MODE 12:
Mov  DISP-1,R22
Add  R22,R21
Sll  SCALE,R22
Ld  0[R21],R23
No-op

;;Mov  PC,R21
;;Mov  Xn,R22
;;Add  R22,R21
;;No-op

MODE 13:
Ld  Bd-2,R22
Mov  PC,R21
Mov  Xn,R22
Ld  [R21+R22],R23
No-op

;;Add  R22,R21
;;Sll  SCALE,R22
;;Add  R22,R21

MODE 14:
Ld  Bd-2,R22
Mov  PC,R21
Ld  [R21+R22],R21
Ld  Od,R22
Add  R22,R21
Ld  0[R21],R23
No-op

;;Mov  Xn,R23
;;Sll  SCALE,R23
;;Add  R23,R21
;;No-op

MODE 15:
Ld  Bd-2,R22
Mov  PC,R21
Mov  Xn,R22
Ld  [R21+R22],R21
Ld  Od,R22
Ld  [R21+R22],R23
No-op

;;Add  R22,R21
;;Sll  SCALE,R22
;;No-op

MODE 16: Directly supported by MIPS

MODE 17:
Ld  Alo,R22
Mov  Ah,R21
Ld  [R21+R22],R23
No-op

;;Sll  #24,R21
;;Add  R22,R21

MODE 18: Directly supported by MIPS
Appendix B

CONDITION CODE SYNTHESIS

B.1 Condition Code Bit Map

The status register consists of a 32 bit dedicated register (R16) where there are only a few bits defined by the condition code emulator. The defined bit map is as follows:

Bit 0: Carry (C)
Bit 1: Overflow (V)
Bit 2: Zero (Z)
Bit 3: Negative (N)
Bit 4: Extend (X)
Bit 5: 0
Bit 6: 0
Bit 7: 0

The upper 24 bits of the condition code register will be reserved for the system monitor. The lower byte of the status register is comparable to the user byte in the MC68020.

B.2 Basic Conditions Tested

There are 29 instruction groupings listed in appendix A-3 of the Motorola user's manual which affect the condition codes in some way. Since the overflow and carry bits are handled with the macros listed below, the overflow exception handler must be disabled; it will be used only for the multiply and divide instructions and will be specifically tested using the conditional trap instruction. Notice that the condition code synthesis is not completely optimized since only the macros required will be inserted. The optimization will be performed after the specific macros required are inserted. The MIPS instruction format was obtained from [GiGrH83].
B.3 Macros Preceding a Conditional Instruction

CC => carry clear
Or #-2,R16,R19 ;;Not R19

CS => carry set
And #1,R16,R19 ;;

EQ => equal
And #4,R16,R19 ;;

F => always false
Mov #0,R19 ;;

GE => greater or equal
Mov #10,R19 ;;And R16,R19
Seq #0,R19
Seq #10,R19
And #1,R19 ;;

GT => greater than
Mov #10,R19 ;;And R16,R19
Seq #0,R19
Seq #10,R19
And #1,R19 ;;No-op
And #4,R16,R20 ;;Sra #2,R20
And R20,R19 ;;

HI => higher
And #5,R16,R19 ;;No-op
Seq #0,R19
And #2,R19 ;;
LE  => less or equal

Mov  #10,R19  ;;And R16,R19
Seq  #2,R19
Seq  #8,R19
And  #4,R16,R20  ;;And #1,R19
Sra  #2,R20  ;;Or R20,R19

LS  => lower or same

And  #5,R16,R19

LT  => less than

Mov  #10,R19  ;;And R16,R19
Seq  #2,R19
Seq  #8,R19
And  #1,R19

MI  => minus

Mov  #8,R19  ;;And R16,R19

NE  => not equal

Or  #-5,R16,R19  ;;Not R19

PL  => plus

Mov  #-9,R19  ;;Or R16,R19
Not R19

T  => always true

Mov  #-1,R19

VC  => overflow clear

Or  #-3,R16,R19  ;;Not R19

VS  => overflow set

And  #2,R16,R19

---
B.4 Condition Code Translation

Definitions:
- **Dn**: Destination register of the instruction
- **Lb**: Lower bound for a comparison
- **Offset**: The offset of the bit field within the effective operand
- **Rn**: Result of the operation
- **Sn**: Source register of the instruction
- **Ub**: Upper bound for a comparison
- **Width**: Width of the bit field in the effective address

**OPERATION**

<table>
<thead>
<tr>
<th>OPERATION</th>
<th>SYNTHESIS</th>
<th>BIT AFFECTED</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABCD</td>
<td>Mov #199,R19 ;;No-op</td>
<td>/* set carry and extend</td>
</tr>
<tr>
<td></td>
<td>Sgt Rn,R19</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Mov #17,R20 ;;And R20,R19</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Not R20 ;;And R20,R16</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Or R19,R16 ;;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Mov Rn,R19 ;;No-op</td>
<td>/* set zero</td>
</tr>
<tr>
<td></td>
<td>Seq #0,R19 ;;Or R19,R16</td>
<td></td>
</tr>
<tr>
<td>ADD</td>
<td>Mov #128,R20 ;;Sll #24,R20</td>
<td>/* set overlap</td>
</tr>
<tr>
<td>ADDI</td>
<td>Xor Sn,Rn,R19 ;;And R19,R20</td>
<td></td>
</tr>
<tr>
<td>ADDQ</td>
<td>Xor Dn,Rn,R19 ;;And R19,R20</td>
<td></td>
</tr>
<tr>
<td>ADDA</td>
<td>Rol #2,R20 ;;And #13,R16</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Or R20,R16 ;;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Mov #128,R20 ;;Sll #24,R20</td>
<td>/* set carry</td>
</tr>
<tr>
<td></td>
<td>And Sn,Dn,R19 ;;And R20,R19</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Rol #1,R19 ;;And #14,R16</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Or R19,R16 ;;No-op</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Or Sn,Dn,R19 ;;And R20,R19</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Xor #1,Rn,R20 ;;And R19,R20</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Rol #1,R20 ;;Or R20,R16</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Mov Rn,R19 ;;No-op</td>
<td>/* set zero</td>
</tr>
<tr>
<td></td>
<td>Seq #0,R19</td>
<td></td>
</tr>
<tr>
<td></td>
<td>And #4,R19 ;;And #11,R16</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Or R19,R16 ;;</td>
<td></td>
</tr>
<tr>
<td>Operation</td>
<td>Synthesis</td>
<td>Bit Affected</td>
</tr>
<tr>
<td>-----------</td>
<td>-----------</td>
<td>--------------</td>
</tr>
<tr>
<td>Mov #9,R19</td>
<td>;;And R19,R16</td>
<td>/* set negative</td>
</tr>
<tr>
<td>Mov #8,R20</td>
<td>;;Mov Rn,R19</td>
<td></td>
</tr>
<tr>
<td>Sgt #0,R19</td>
<td>;;Or R19,R16</td>
<td></td>
</tr>
<tr>
<td>And R20,R19</td>
<td></td>
<td></td>
</tr>
<tr>
<td>And #1,R16,R19</td>
<td>;;Rol #4,R19</td>
<td>/* set extend</td>
</tr>
<tr>
<td>Mov #17,R20</td>
<td>;;And R20,R16</td>
<td></td>
</tr>
<tr>
<td>Or R19,R16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADDX</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mov #128,R20</td>
<td>;;Sll #24,R20</td>
<td>/* set overflow</td>
</tr>
<tr>
<td>Xor Sn,Rn,R19</td>
<td>;;And R19,R20</td>
<td></td>
</tr>
<tr>
<td>Xor Dn,Rn,R19</td>
<td>;;And R19,R20</td>
<td></td>
</tr>
<tr>
<td>Rol #2,R20</td>
<td>;;And #13,R16</td>
<td></td>
</tr>
<tr>
<td>Or R20,R16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mov #128,R20</td>
<td>;;Sll #24,R20</td>
<td>/* set carry</td>
</tr>
<tr>
<td>And Sn,Dn,R19</td>
<td>;;And R20,R19</td>
<td></td>
</tr>
<tr>
<td>Rol #1,R19</td>
<td>;;And #14,R16</td>
<td></td>
</tr>
<tr>
<td>Or R19,R16</td>
<td>;;No-op</td>
<td></td>
</tr>
<tr>
<td>Or Sn,Dn,R19</td>
<td>;;And R20,R19</td>
<td></td>
</tr>
<tr>
<td>Xor #1,Rn,R20</td>
<td>;;And R19,R20</td>
<td></td>
</tr>
<tr>
<td>Rol #1,R20</td>
<td>;;Or R20,R16</td>
<td></td>
</tr>
<tr>
<td>Mov Rn,R19</td>
<td>;;No-op</td>
<td>/* set zero</td>
</tr>
<tr>
<td>Seq #0,R19</td>
<td></td>
<td></td>
</tr>
<tr>
<td>And R16,R19</td>
<td>;;Or R19,R16</td>
<td></td>
</tr>
<tr>
<td>Mov #9,R19</td>
<td>;;And R19,R16</td>
<td>/* set negative</td>
</tr>
<tr>
<td>Mov #8,R20</td>
<td>;;No-op</td>
<td></td>
</tr>
<tr>
<td>Sgt #0,R19</td>
<td></td>
<td></td>
</tr>
<tr>
<td>And R20,R19</td>
<td>;;Or R19,R16</td>
<td></td>
</tr>
<tr>
<td>And #1,R16,R19</td>
<td>;;Rol #4,R19</td>
<td>/* set extend</td>
</tr>
<tr>
<td>Mov #17,R20</td>
<td>;;And R20,R16</td>
<td></td>
</tr>
<tr>
<td>Or R19,R16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AND</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ANDI</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EOR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EORI</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MOVEQ</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ORI</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CLR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NOT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TAS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TST</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OPERATION</td>
<td>SYNTHESIS</td>
<td>BIT AFFECTED</td>
</tr>
<tr>
<td>-----------</td>
<td>-----------</td>
<td>--------------</td>
</tr>
<tr>
<td>CHK</td>
<td>Mov #9,R19 ;And R19,R16 /* set negative</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Mov #8,R20 ;Mov Rn,R19</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Sgt #0,R19 ;Or R19,R16</td>
<td></td>
</tr>
<tr>
<td></td>
<td>And R20,R19</td>
<td></td>
</tr>
<tr>
<td>CHK2</td>
<td>Mov Ub,R19 ;No-op /* set zero</td>
<td></td>
</tr>
<tr>
<td>CMP2</td>
<td>Seq Rn,R19 ;No-op</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Mov Lb,R20 ;No-op</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Seq Rn,R20 ;And R19,R20</td>
<td></td>
</tr>
<tr>
<td></td>
<td>And #4,R20 ;And #11,R16</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Or R20,R16 ;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Mov #8,R20 ;No-op /* set carry</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Slt Lb,R19 ;No-op</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Mov Rn,R20 ;No-op</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Sgt Ub,R20 ;Mov Lb,R20</td>
<td></td>
</tr>
<tr>
<td></td>
<td>And R20,R19 ;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Sgt Ub,R20 ;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Xor R19,R20 ;And #1,R20</td>
<td></td>
</tr>
<tr>
<td></td>
<td>And #14,R16 ;Or R20,R16</td>
<td></td>
</tr>
<tr>
<td>SUB</td>
<td>Mov #128,R20 ;Sll #24,R20 /* set overflow</td>
<td></td>
</tr>
<tr>
<td>SUBI</td>
<td>Xor Sn,Dn,R19 ;And R19,R20</td>
<td></td>
</tr>
<tr>
<td>SUBQ</td>
<td>Xor Dn,Rn,R19 ;And R19,R20</td>
<td></td>
</tr>
<tr>
<td>SUBA</td>
<td>Rol #2,R20 ;And #13,R16</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Or R20,R16 ;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Mov #128,R20 ;Sll #24,R20 /* set carry</td>
<td></td>
</tr>
<tr>
<td></td>
<td>And Sn,Rn,R19 ;And R20,R19</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Rol #1,R19 ;And #14,R16</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Or R19,R16 ;No-op</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Or Sn,Rn,R19 ;And R20,R19</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Xor #1,Dn,R20 ;And R19,R20</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Rol #1,R20 ;Or R20,R16</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Mov Rn,R19 ;No-op /* set zero</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Seq #0,R19 ;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>And #4,R19 ;And #11,R16</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Or R19,R16 ;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Mov #9,R19 ;Mov Rn,R19 /* set negative</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Mov #8,R20 ;And R19,R16</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Sgt #0,R19 ;Or R19,R16</td>
<td></td>
</tr>
<tr>
<td></td>
<td>And R20,R19 ;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>And #1,R16,R19 ;Rol #4,R19 /* set extend</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Mov #17,R20 ;And R20,R16</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Or R19,R16 ;</td>
<td></td>
</tr>
<tr>
<td>OPERATION</td>
<td>SYNTHESIS</td>
<td>BIT AFFECTED</td>
</tr>
<tr>
<td>-----------</td>
<td>-----------</td>
<td>--------------</td>
</tr>
<tr>
<td>SUBX</td>
<td>Mov #128,R20 ;;Sll #24,R20</td>
<td>/* set overflow</td>
</tr>
<tr>
<td></td>
<td>Xor Sn,Dn,R19 ;;And R19,R20</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Xor Dn,Rn,R19 ;;And R19,R20</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Rol #2,R20 ;;And #13,R16</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Or R20,R16</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Mov #128,R20 ;;Sll #24,R20</td>
<td>/* set carry</td>
</tr>
<tr>
<td></td>
<td>And Sn,Rn,R19 ;;And R20,R19</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Rol #1,R19 ;;And #14,R16</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Or R19,R16 ;;No-op</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Or Sn,Rn,R19 ;;And R20,R19</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Xor #-1,Dn,R20 ;;And R19,R20</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Rol #1,R20 ;;Or R20,R16</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Mov Rn,R19 ;;No-op</td>
<td>/* set zero</td>
</tr>
<tr>
<td></td>
<td>Seq #0,R19</td>
<td></td>
</tr>
<tr>
<td></td>
<td>And R16,R19 ;;Or R19,R16</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Mov #-9,R19 ;;And R19,R16</td>
<td>/* set negative</td>
</tr>
<tr>
<td></td>
<td>Mov #8,R20 ;;Mov Rn,R19</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Sgt #0,R19</td>
<td></td>
</tr>
<tr>
<td></td>
<td>And R20,R19 ;;Or R19,R16</td>
<td></td>
</tr>
<tr>
<td></td>
<td>And #1,R16,R19 ;;Rol #4,R19</td>
<td>/* set extend</td>
</tr>
<tr>
<td></td>
<td>Or R19,R16</td>
<td></td>
</tr>
<tr>
<td>CAS</td>
<td>Mov #128,R20 ;;Sll #24,R20</td>
<td>/* set overflow</td>
</tr>
<tr>
<td>CASZ</td>
<td>Xor Sn,Dn,R19 ;;And R19,R20</td>
<td></td>
</tr>
<tr>
<td>CMP</td>
<td>Xor Dn,Rn,R19 ;;And R19,R20</td>
<td></td>
</tr>
<tr>
<td>CMPI</td>
<td>Rol #2,R20 ;;And #13,R16</td>
<td></td>
</tr>
<tr>
<td>CMPM</td>
<td>Or R20,R16 ;;No-op</td>
<td></td>
</tr>
<tr>
<td>CMPA</td>
<td>Mov #128,R20 ;;Sll #24,R20</td>
<td>/* set carry</td>
</tr>
<tr>
<td></td>
<td>And Sn,Rn,R19 ;;And R20,R19</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Rol #1,R19,R19 ;;And #14,R16</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Or R19,R16 ;;No-op</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Or Sn,Rn,R19 ;;And R20,R19</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Xor #-1,Dn,R20 ;;And R19,R20</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Rol #1,R20 ;;Or R20,R16</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Mov Rn,R19 ;;No-op</td>
<td>/* set zero</td>
</tr>
<tr>
<td></td>
<td>Seq #0,R19</td>
<td></td>
</tr>
<tr>
<td></td>
<td>And #4,R19 ;;Or R19,R16</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Mov #-9,R19 ;;And R19,R16</td>
<td>/* set negative</td>
</tr>
<tr>
<td></td>
<td>Mov #8,R20 ;;Mov Rn,R19</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Sgt #0,R19</td>
<td></td>
</tr>
<tr>
<td></td>
<td>And R20,R19 ;;Or R19,R16</td>
<td></td>
</tr>
</tbody>
</table>

- 38 -
<table>
<thead>
<tr>
<th>OPERATION</th>
<th>SYNTHESIS</th>
<th>BIT AFFECTED</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIVS</td>
<td>Overflow bit is set during the instruction.</td>
<td>/* clear carry</td>
</tr>
<tr>
<td>DIVU</td>
<td>And #14, R16</td>
<td>/* set negative</td>
</tr>
<tr>
<td>MULS</td>
<td>Mov #-9, R19 And R19, R16</td>
<td>/* set negative</td>
</tr>
<tr>
<td>MULU</td>
<td>Mov #8, R20 Mov Rn, R19</td>
<td>/* clear carry</td>
</tr>
<tr>
<td></td>
<td>Sgt #0, R19 ;; Or R19, R16</td>
<td>/* set negative</td>
</tr>
<tr>
<td></td>
<td>And R20, R19 ;; Or R19, R16</td>
<td>/* clear carry</td>
</tr>
<tr>
<td></td>
<td>Mov Rn, R19 ;; No-op</td>
<td>/* set zero</td>
</tr>
<tr>
<td></td>
<td>Seq #0, R19</td>
<td>/* set zero</td>
</tr>
<tr>
<td></td>
<td>And #4, R19 ;; And #11, R16</td>
<td>/* set carry</td>
</tr>
<tr>
<td></td>
<td>Or R16, R19</td>
<td>/* set carry and extend</td>
</tr>
<tr>
<td>SBCD</td>
<td>Mov Rn, R19 ;; No-op</td>
<td>/* set carry</td>
</tr>
<tr>
<td>NBCD</td>
<td>Sgt #0, R19 ;; Or R19, R16</td>
<td>/* set carry</td>
</tr>
<tr>
<td></td>
<td>Mov #17, R20 ;; And R20, R19</td>
<td>/* set carry and extend</td>
</tr>
<tr>
<td></td>
<td>Not R20 ;; And R20, R16</td>
<td>/* set carry</td>
</tr>
<tr>
<td></td>
<td>Or R19, R16 ;; And R20, R16</td>
<td>/* set carry and extend</td>
</tr>
<tr>
<td></td>
<td>Mov Rn, R19 ;; No-op</td>
<td>/* set zero</td>
</tr>
<tr>
<td></td>
<td>Seq #0, R19</td>
<td>/* set zero</td>
</tr>
<tr>
<td></td>
<td>And #4, R19 ;; And #11, R16</td>
<td>/* set carry</td>
</tr>
<tr>
<td></td>
<td>Or R19, R16 ;; And R20, R16</td>
<td>/* set carry and extend</td>
</tr>
<tr>
<td>NEG</td>
<td>Mov #128, R20 ;; Sll #24, R20</td>
<td>/* set carry</td>
</tr>
<tr>
<td></td>
<td>Mov Dn, R19 ;; Or Rn, R19</td>
<td>/* clear carry</td>
</tr>
<tr>
<td></td>
<td>And R20, R19 ;; Rol #1, R19</td>
<td>/* set carry</td>
</tr>
<tr>
<td></td>
<td>And #14, R16 ;; Or R19, R16</td>
<td>/* set carry</td>
</tr>
<tr>
<td></td>
<td>Mov #128, R20 ;; Sll #24, R20</td>
<td>/* set carry</td>
</tr>
<tr>
<td></td>
<td>Mov Dn, R19 ;; And Rn, R19</td>
<td>/* set carry</td>
</tr>
<tr>
<td></td>
<td>And R20, R19 ;; Rol #2, R19</td>
<td>/* set carry</td>
</tr>
<tr>
<td></td>
<td>And #13, R16 ;; Or R19, R16</td>
<td>/* set carry</td>
</tr>
<tr>
<td></td>
<td>Mov Rn, R19 ;; No-op</td>
<td>/* set zero</td>
</tr>
<tr>
<td></td>
<td>Seq #0, R19</td>
<td>/* set zero</td>
</tr>
<tr>
<td></td>
<td>And #4, R19 ;; And #11, R16</td>
<td>/* set carry</td>
</tr>
<tr>
<td></td>
<td>Or R19, R16 ;; And R20, R16</td>
<td>/* set carry and extend</td>
</tr>
<tr>
<td></td>
<td>Mov #-9, R19 ;; And R19, R16</td>
<td>/* set negative</td>
</tr>
<tr>
<td></td>
<td>Mov #8, R20 ;; Mov Rn, R19</td>
<td>/* clear carry</td>
</tr>
<tr>
<td></td>
<td>Sgt #0, R19 ;; Or R19, R16</td>
<td>/* clear carry</td>
</tr>
<tr>
<td></td>
<td>And R20, R19 ;; Or R19, R16</td>
<td>/* clear carry</td>
</tr>
<tr>
<td></td>
<td>And #1, R16, R19 ;; Rol #4, R19</td>
<td>/* set extend</td>
</tr>
<tr>
<td></td>
<td>Mov #-17, R20 ;; And R20, R16</td>
<td>/* set extend</td>
</tr>
<tr>
<td></td>
<td>Or R19, R16 ;; And R20, R16</td>
<td>/* set extend</td>
</tr>
<tr>
<td>OPERATION</td>
<td>SYNTHESIS</td>
<td></td>
</tr>
<tr>
<td>-----------</td>
<td>-----------</td>
<td></td>
</tr>
<tr>
<td><strong>NEGX</strong></td>
<td>Mov #128,R20 ;;Sll #24,R20 /* set carry*/</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Mov Dn,R19 ;;Or Rn,R19</td>
<td></td>
</tr>
<tr>
<td></td>
<td>And R20,R19 ;;Rol #1,R19</td>
<td></td>
</tr>
<tr>
<td></td>
<td>And #14,R16 ;;Or R19,R16</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Mov #128,R20 ;;Sll #24,R20 /* set overflow*/</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Mov Dn,R19 ;;And Rn,R19</td>
<td></td>
</tr>
<tr>
<td></td>
<td>And R20,R19 ;;Rol #2,R19</td>
<td></td>
</tr>
<tr>
<td></td>
<td>And #13,R16 ;;Or R19,R16</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Mov Rn,R19 ;;No-op /* set zero*/</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Seq #0,R19</td>
<td></td>
</tr>
<tr>
<td></td>
<td>And #4,R19 ;;And #11,R16</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Or R19,R16 ;;</td>
<td></td>
</tr>
<tr>
<td><strong>BTST</strong></td>
<td>Mov #1,R19 ;;Mov Dn,Lo /* set zero*/</td>
<td></td>
</tr>
<tr>
<td><strong>BCHG</strong></td>
<td>Rol Lo,R19 ;;And R23,R19</td>
<td></td>
</tr>
<tr>
<td><strong>BSET</strong></td>
<td>Seq #0,R20</td>
<td></td>
</tr>
<tr>
<td><strong>BCLR</strong></td>
<td>And #4,R20 ;;And #11,R16</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Or R20,R16 ;;</td>
<td></td>
</tr>
<tr>
<td><strong>BFTST</strong></td>
<td>And #12,R16 ;; /* clear carry and overflow*/</td>
<td></td>
</tr>
<tr>
<td><strong>BFCHG</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>BFSET</strong></td>
<td>Mov Width,Lo ;;Mov #0,R19 /* set negative*/</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Mov #1,R20 ;;Rlc R20,R19</td>
<td></td>
</tr>
<tr>
<td><strong>BFCLR</strong></td>
<td>Sll Offset,R19 ;;And Dn,R19</td>
<td></td>
</tr>
<tr>
<td><strong>BFEXTS</strong></td>
<td>SF Or=R+Wis-1,R19 ;;Sll #3,R19</td>
<td></td>
</tr>
<tr>
<td><strong>BFEXTU</strong></td>
<td>Mov #9,R16 ;;And R20,R16</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Or R19,R16 ;;</td>
<td></td>
</tr>
<tr>
<td><strong>BFFFO</strong></td>
<td>Mov Width,Lo ;;Mov #0,R19 /* set zero*/</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Mov #1,R20 ;;Rlc R20,R19</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Sll Offset,R19 ;;And Dn,R19</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Seq #0,R19</td>
<td></td>
</tr>
<tr>
<td></td>
<td>And #4,R19 ;;And #11,R16</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Or R19,R16 ;;</td>
<td></td>
</tr>
<tr>
<td><strong>BFINS</strong></td>
<td>And #12,R16 ;; /* clear carry and overflow*/</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Mov #128,R20 ;;Sll #24,R20 /* set negative*/</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Mov Sn,R19 ;;And R20,R19</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Mov #9,R20 ;;Rol #4,R19</td>
<td></td>
</tr>
<tr>
<td></td>
<td>And R20,R16 ;;Or R19,R16</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Mov Sn,R19 ;;No-op /* set zero*/</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Seq #0,R19</td>
<td></td>
</tr>
<tr>
<td></td>
<td>And #4,R19 ;;And #11,R16</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Or R19,R16 ;;</td>
<td></td>
</tr>
</tbody>
</table>
OPERATION SYNTHESIS BIT AFFECTED

ASL

Carry/overflow set during instruction execution

<table>
<thead>
<tr>
<th>Operation</th>
<th>Syntax</th>
<th>Description</th>
<th>Bit Affected</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mov</td>
<td>Rn,R19</td>
<td>No-op</td>
<td>/* set zero</td>
</tr>
<tr>
<td>Seq</td>
<td>#0,R19</td>
<td></td>
<td></td>
</tr>
<tr>
<td>And</td>
<td>#4,R19</td>
<td>;;And #11,R16</td>
<td></td>
</tr>
<tr>
<td>Or</td>
<td>R19,R16</td>
<td>;;</td>
<td></td>
</tr>
<tr>
<td>Mov</td>
<td>#9,R19</td>
<td>;;And R19,R16</td>
<td>/* set negative</td>
</tr>
<tr>
<td>Mov</td>
<td>#8,R20</td>
<td>;;Mov Rn,R19</td>
<td></td>
</tr>
<tr>
<td>Sgt</td>
<td>#0,R19</td>
<td>;;Or R19,R16</td>
<td></td>
</tr>
<tr>
<td>And</td>
<td>#1,R16,R19</td>
<td>;;Rol #4,R19</td>
<td>/* set extend</td>
</tr>
<tr>
<td>Mov</td>
<td>#-17,R20</td>
<td>;;And R20,R16</td>
<td></td>
</tr>
<tr>
<td>Or</td>
<td>R19,R16</td>
<td>;;</td>
<td></td>
</tr>
</tbody>
</table>

ASL (r=0)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Syntax</th>
<th>Description</th>
<th>Bit Affected</th>
</tr>
</thead>
<tbody>
<tr>
<td>And</td>
<td>#12,R16</td>
<td>;;</td>
<td>/* clear carry and overflow</td>
</tr>
<tr>
<td>Mov</td>
<td>Rn,R19</td>
<td>No-op</td>
<td>/* set zero</td>
</tr>
<tr>
<td>Seq</td>
<td>#0,R19</td>
<td></td>
<td></td>
</tr>
<tr>
<td>And</td>
<td>#4,R19</td>
<td>;;And #11,R16</td>
<td></td>
</tr>
<tr>
<td>Or</td>
<td>R19,R16</td>
<td>;;</td>
<td></td>
</tr>
<tr>
<td>Mov</td>
<td>#9,R19</td>
<td>;;And R19,R16</td>
<td>/* set negative</td>
</tr>
<tr>
<td>Mov</td>
<td>#8,R20</td>
<td>;;Mov Rn,R19</td>
<td></td>
</tr>
<tr>
<td>Sgt</td>
<td>#0,R19</td>
<td>;;Or R19,R16</td>
<td></td>
</tr>
</tbody>
</table>

LSL ROXL

Carry set during instruction execution

<table>
<thead>
<tr>
<th>Operation</th>
<th>Syntax</th>
<th>Description</th>
<th>Bit Affected</th>
</tr>
</thead>
<tbody>
<tr>
<td>And</td>
<td>#13,R16</td>
<td>;;</td>
<td>/* clear overflow</td>
</tr>
<tr>
<td>Mov</td>
<td>Rn,R19</td>
<td>No-op</td>
<td>/* set zero</td>
</tr>
<tr>
<td>Seq</td>
<td>#0,R19</td>
<td></td>
<td></td>
</tr>
<tr>
<td>And</td>
<td>#4,R19</td>
<td>;;And #11,R16</td>
<td></td>
</tr>
<tr>
<td>Or</td>
<td>R19,R16</td>
<td>;;</td>
<td></td>
</tr>
<tr>
<td>Mov</td>
<td>#9,R19</td>
<td>;;And R19,R16</td>
<td>/* set negative</td>
</tr>
<tr>
<td>Mov</td>
<td>#8,R20</td>
<td>;;Mov Rn,R19</td>
<td></td>
</tr>
<tr>
<td>Sgt</td>
<td>#0,R19</td>
<td>;;Or R19,R16</td>
<td></td>
</tr>
<tr>
<td>And</td>
<td>#1,R16,R19</td>
<td>;;Rol #4,R19</td>
<td>/* set extend</td>
</tr>
<tr>
<td>Mov</td>
<td>#-17,R20</td>
<td>;;And R20,R16</td>
<td></td>
</tr>
<tr>
<td>Or</td>
<td>R19,R16</td>
<td>;;</td>
<td></td>
</tr>
<tr>
<td>OPERATION</td>
<td>SYNTHESIS</td>
<td>BIT AFFECTED</td>
<td></td>
</tr>
<tr>
<td>-----------</td>
<td>------------</td>
<td>--------------</td>
<td></td>
</tr>
<tr>
<td><strong>LSR</strong> (r=0)</td>
<td>And #12,R16</td>
<td>*/* clear carry and overflow</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Mov Rn,R19</td>
<td>*/* set zero</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Seq #0,R19</td>
<td>*And #11,R16</td>
<td></td>
</tr>
<tr>
<td></td>
<td>And #4,R19</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Or R19,R19</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Mov #−9,R19</td>
<td>*And R19,R16</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Mov #8,R20</td>
<td>*Mov Rn,R19</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Sgt #0,R19</td>
<td>*Or R19,R16</td>
<td></td>
</tr>
<tr>
<td></td>
<td>And R20,R19</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td><strong>ROXL</strong> (r=0)</td>
<td>Mov #16,R20</td>
<td>*set carry</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Srl #4,R20</td>
<td>*And #14,R16</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Or R20,R19</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td></td>
<td>And #13,R16</td>
<td>*clear overflow</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Mov Rn,R19</td>
<td>*set zero</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Seq #0,R19</td>
<td>*And #11,R16</td>
<td></td>
</tr>
<tr>
<td></td>
<td>And #4,R19</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Or R19,R19</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Mov #−9,R19</td>
<td>*And R19,R16</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Mov #8,R20</td>
<td>*Mov Rn,R19</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Sgt #0,R19</td>
<td>*Or R19,R16</td>
<td></td>
</tr>
<tr>
<td></td>
<td>And R20,R19</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td><strong>ROL</strong></td>
<td>Carry set during instruction execution</td>
<td></td>
<td></td>
</tr>
<tr>
<td>And #13,R16</td>
<td>*clear overflow</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mov Rn,R19</td>
<td>*set zero</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Seq #0,R19</td>
<td>*And #11,R16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>And #4,R19</td>
<td>*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Or R19,R19</td>
<td>*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mov #−9,R19</td>
<td>*And R19,R16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mov #8,R20</td>
<td>*Mov Rn,R19</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sgt #0,R19</td>
<td>*Or R19,R16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>And R20,R19</td>
<td>*</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>ROL</strong> (r=0)</td>
<td>And #12,R16</td>
<td>*clear carry and overflow</td>
<td></td>
</tr>
<tr>
<td>Mov Rn,R19</td>
<td>*set zero</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Seq #0,R19</td>
<td>*And #11,R16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>And #4,R19</td>
<td>*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Or R19,R19</td>
<td>*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OPERATION</td>
<td>SYNTHESIS</td>
<td>BIT AFFECTED</td>
<td></td>
</tr>
<tr>
<td>-----------</td>
<td>-----------</td>
<td>--------------</td>
<td></td>
</tr>
<tr>
<td>Mov #9,R19</td>
<td>;;And R19,R16</td>
<td>/* set negative</td>
<td></td>
</tr>
<tr>
<td>Mov #8,R20</td>
<td>;;Mov Rn,R19</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sgt #0,R19</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>And R20,R19</td>
<td>;;Or R19,R16</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**ASR**

**LSR**

**ROXR**

<table>
<thead>
<tr>
<th>Carry set during instruction execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>And #13,R16</td>
</tr>
<tr>
<td>Mov Rn,R19</td>
</tr>
<tr>
<td>Seq #0,R19</td>
</tr>
<tr>
<td>And #4,R19</td>
</tr>
<tr>
<td>Or R19,R16</td>
</tr>
<tr>
<td>Mov #9,R19</td>
</tr>
<tr>
<td>Mov #8,R20</td>
</tr>
<tr>
<td>Sgt #0,R19</td>
</tr>
<tr>
<td>And R20,R19</td>
</tr>
<tr>
<td>And #1,R16,R19</td>
</tr>
<tr>
<td>Mov #17,R20</td>
</tr>
<tr>
<td>Or R19,R16</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ASR</th>
<th>LSR</th>
<th>ROXR</th>
</tr>
</thead>
<tbody>
<tr>
<td>(r=0)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>And #12,R16</td>
<td>;;</td>
<td></td>
</tr>
<tr>
<td>Mov Rn,R19</td>
<td>;;No-op</td>
<td></td>
</tr>
<tr>
<td>Seq #0,R19</td>
<td></td>
<td></td>
</tr>
<tr>
<td>And #4,R19</td>
<td>;;And #11,R16</td>
<td></td>
</tr>
<tr>
<td>Or R19,R16</td>
<td>;;</td>
<td></td>
</tr>
<tr>
<td>Mov #9,R19</td>
<td>;;And R19,R16</td>
<td></td>
</tr>
<tr>
<td>Mov #8,R20</td>
<td>;;Mov Rn,R19</td>
<td></td>
</tr>
<tr>
<td>Sgt #0,R19</td>
<td></td>
<td></td>
</tr>
<tr>
<td>And R20,R19</td>
<td>;;Or R19,R16</td>
<td></td>
</tr>
<tr>
<td>Mov #16,R20</td>
<td>;;And R16,R20</td>
<td></td>
</tr>
<tr>
<td>Srl #4,R20</td>
<td>;;And #14,R16</td>
<td></td>
</tr>
<tr>
<td>Or R20,R16</td>
<td>;;</td>
<td></td>
</tr>
<tr>
<td>And #13,R16</td>
<td>;;</td>
<td></td>
</tr>
<tr>
<td>Mov Rn,R19</td>
<td>;;No-op</td>
<td></td>
</tr>
<tr>
<td>Seq #0,R19</td>
<td></td>
<td></td>
</tr>
<tr>
<td>And #4,R19</td>
<td>;;And #11,R16</td>
<td></td>
</tr>
<tr>
<td>Or R19,R16</td>
<td>;;</td>
<td></td>
</tr>
<tr>
<td>Mov #9,R19</td>
<td>;;And R19,R16</td>
<td></td>
</tr>
<tr>
<td>Mov #8,R20</td>
<td>;;Mov Rn,R19</td>
<td></td>
</tr>
<tr>
<td>Sgt #0,R19</td>
<td></td>
<td></td>
</tr>
<tr>
<td>And R20,R19</td>
<td>;;Or R19,R16</td>
<td></td>
</tr>
<tr>
<td>OPERATION</td>
<td>SYNTHESIS</td>
<td>BIT AFFECTED</td>
</tr>
<tr>
<td>-----------</td>
<td>-----------</td>
<td>--------------</td>
</tr>
<tr>
<td>ROR</td>
<td>Carry set during instruction execution</td>
<td></td>
</tr>
<tr>
<td>And</td>
<td>#13, R16</td>
<td></td>
</tr>
<tr>
<td>Mov</td>
<td>Rn, R19</td>
<td>;; No-op</td>
</tr>
<tr>
<td>Seq</td>
<td>#0, R19</td>
<td></td>
</tr>
<tr>
<td>And</td>
<td>#4, R19</td>
<td>;; And #11, R16</td>
</tr>
<tr>
<td>Or</td>
<td>R19, R16</td>
<td>;;</td>
</tr>
<tr>
<td>Mov</td>
<td>#9, R19</td>
<td>;; And R19, R16</td>
</tr>
<tr>
<td>Mov</td>
<td>#8, R20</td>
<td>;; Mov Rn, R19</td>
</tr>
<tr>
<td>Sgt</td>
<td>#0, R19</td>
<td></td>
</tr>
<tr>
<td>And</td>
<td>R20, R19</td>
<td>;; Or R19, R16</td>
</tr>
<tr>
<td>ROR</td>
<td></td>
<td>/* clear carry and overflow</td>
</tr>
<tr>
<td>(r=0)</td>
<td>And #12, R16</td>
<td>;;</td>
</tr>
<tr>
<td>Mov</td>
<td>Rn, R19</td>
<td>;; No-op</td>
</tr>
<tr>
<td>Seq</td>
<td>#0, R19</td>
<td></td>
</tr>
<tr>
<td>And</td>
<td>#4, R19</td>
<td>;; And #11, R16</td>
</tr>
<tr>
<td>Or</td>
<td>R19, R16</td>
<td>;;</td>
</tr>
<tr>
<td>Mov</td>
<td>#9, R19</td>
<td>;; And R19, R16</td>
</tr>
<tr>
<td>Mov</td>
<td>#8, R20</td>
<td>;; Mov Rn, R19</td>
</tr>
<tr>
<td>Sgt</td>
<td>#0, R19</td>
<td></td>
</tr>
<tr>
<td>And</td>
<td>R20, R19</td>
<td>;; Or R19, R16</td>
</tr>
</tbody>
</table>
Appendix C

INSTRUCTION SET SYNTHESIS

C.1 Register Definitions used in Translation

An Any address register
Cond Any valid condition
Dn Any data register
Dc Compare operand register
Du Update operand register
<ea> The effective address register of the operand (R21)
<eop> The prefetched effective operand register (R23)
Hi Hi register
Label Any valid label
LI Any constant in the range of 0 and 2^24
Lo L register
OFFSET A bit field offset (0-31 MOD 32)
Rx,Ry Any data register pair
SI Any constant between 0 and 32 (0 to 8 in most cases)
WIDTH A bit field width (0-31 MOD 32)

NOTES:

1. In register direct mode, <eop> is replaced by an arbitrary destination register and the store instruction is omitted.
2. R19, R20 and R22 are used as scratch registers but must not affect condition code evaluation. If two effective operands are used, the first operand fetched will be placed in R20 and the second operand will reside in R23.
3. WIDTH + OFFSET must not exceed 32 bits. A translation error message should be generated if they are.
4. Not all forms of the instructions will be translated. For example:

    ADD <ea>,Dn  <=>  Add R23,Dn  ;;
    ADD Dn,<ea>  <=>  Add Dn,R23  ;;No-op
                      St  R23,0[R21]  ;;

will be translated using the second form of the instruction because the first form is less complex. The primary concern in the translation is to determine the worst case efficiency in these examples.
C.2 Instruction Set Translation

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABCD</td>
<td></td>
</tr>
<tr>
<td>UNPK &lt;eop1&gt;,&lt;eop1&gt;</td>
<td>[ MC68020 instruction ]</td>
</tr>
<tr>
<td>UNPK &lt;eop2&gt;,&lt;eop2&gt;</td>
<td>[ MC68020 instruction ]</td>
</tr>
<tr>
<td>Mov #1,R19</td>
<td>;;Sll #4,R19</td>
</tr>
<tr>
<td>And R16,R19</td>
<td>;;Sra #4,R19</td>
</tr>
<tr>
<td>Add &lt;eop1&gt;,&lt;eop2&gt;</td>
<td>;;Add R19,&lt;eop2&gt;</td>
</tr>
<tr>
<td>PACK &lt;eop2&gt;,&lt;eop2&gt;</td>
<td>[ MC68020 instruction ]</td>
</tr>
<tr>
<td>St &lt;eop2&gt;,0[&lt;ea&gt;]</td>
<td>;;</td>
</tr>
<tr>
<td>ADD</td>
<td>Add &lt;Dn,&lt;eop&gt;</td>
</tr>
<tr>
<td></td>
<td>St &lt;eop&gt;,0[&lt;ea&gt;]</td>
</tr>
<tr>
<td>ADDA</td>
<td>Add &lt;eop&gt;,An</td>
</tr>
<tr>
<td>ADDI</td>
<td>Ld LI,R19</td>
</tr>
<tr>
<td></td>
<td>Add &lt;eop&gt;,R19</td>
</tr>
<tr>
<td></td>
<td>St R19,&lt;ea&gt;</td>
</tr>
<tr>
<td>ADDQ</td>
<td>Add SI,&lt;eop&gt;</td>
</tr>
<tr>
<td></td>
<td>St &lt;eop&gt;,0[&lt;ea&gt;]</td>
</tr>
<tr>
<td>ADDX</td>
<td>Mov #1,R19</td>
</tr>
<tr>
<td></td>
<td>And R16,R19</td>
</tr>
<tr>
<td></td>
<td>Add &lt;eop1&gt;,&lt;eop2&gt;</td>
</tr>
<tr>
<td></td>
<td>St &lt;eop2&gt;,0[&lt;ea&gt;]</td>
</tr>
<tr>
<td>AND</td>
<td>And Dn,&lt;eop&gt;</td>
</tr>
<tr>
<td></td>
<td>St &lt;eop&gt;,0[&lt;ea&gt;]</td>
</tr>
<tr>
<td>ANDI</td>
<td>Ld LI,R19</td>
</tr>
<tr>
<td></td>
<td>And R19,&lt;eop&gt;</td>
</tr>
<tr>
<td></td>
<td>St &lt;eop&gt;,0[&lt;ea&gt;]</td>
</tr>
<tr>
<td>ASL</td>
<td>Mov SI,Lo</td>
</tr>
<tr>
<td></td>
<td>Rlc &lt;eop&gt;,R19</td>
</tr>
<tr>
<td></td>
<td>Beq #0,R19,Out</td>
</tr>
<tr>
<td></td>
<td>And #1,R19,R20</td>
</tr>
<tr>
<td></td>
<td>Or R20,R16</td>
</tr>
<tr>
<td></td>
<td>Sra #32-SI,R19</td>
</tr>
<tr>
<td></td>
<td>Sra #0,R19</td>
</tr>
<tr>
<td></td>
<td>And #2,R19</td>
</tr>
<tr>
<td></td>
<td>Out St &lt;eop&gt;,0[&lt;ea&gt;]</td>
</tr>
</tbody>
</table>
ASR
Mov SI-#1,Lo
And #1,<eop>,R19
Sra #1,<eop>
St <eop>,0[<ea>]

BCC
Bne #0,R19,Label
No-op

BCHG
Mov #1,R19
Xor R19,<eop>
St <eop>,0[<ea>]

BCLR
Mov #1,R19
Not R19
St <eop>,0[<ea>]

BFCHG
Mov WIDTH,Lo
Mov #1,R20
Sll OFFSET,R19
St <eop>,0[<ea>]

BFCLR
Mov WIDTH,Lo
Mov #1,R20
Sll OFFSET,R19
And R19,<eop>
St <eop>,0[<ea>]

BFEXTS
Mov WIDTH,Lo
Mov #1,R20
Sll OFFSET,R19
Sll #32-WIDTH-OFFSET,R19
Mov R19,Dn

BFEXTU
Mov WIDTH,Lo
Mov #1,R20
Sll OFFSET,R19
Srl OFFSET,R19
Mov OFFSET,R19

BFFFFO
Mov WIDTH+OFFSET,R19
Mov OFFSET,R22
Next And Dn,<eop>,R20
Bne #0,R20,Out
Sub #1,R19
Bne R19,R22,Next
No-op
Mov WIDTH+OFFSET,R19
No-op
Out Add #1,R19,Dn

BFINS
Mov WIDTH,Lo
Mov #1,R20
And Dn,R19,R20

- 47 -
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sll</td>
<td>OFFSET,R19</td>
<td>;;And R19,&lt;eop&gt;</td>
</tr>
<tr>
<td>Or</td>
<td>R20,&lt;eop&gt;</td>
<td>;;No-op</td>
</tr>
<tr>
<td>St</td>
<td>&lt;eop&gt;,0[&lt;ea&gt;]</td>
<td></td>
</tr>
<tr>
<td>BFSET</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mov</td>
<td>WIDTH,Lo</td>
<td>;;Mov #0,R19</td>
</tr>
<tr>
<td>Mov</td>
<td>#1,R20</td>
<td>;;Rlc R20,R19</td>
</tr>
<tr>
<td>Sll</td>
<td>OFFSET,R19</td>
<td>;;Or R19,&lt;eop&gt;</td>
</tr>
<tr>
<td>St</td>
<td>&lt;eop&gt;,0[&lt;ea&gt;]</td>
<td></td>
</tr>
<tr>
<td>BFTST</td>
<td>Affects condition codes only.</td>
<td></td>
</tr>
<tr>
<td>BSET</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mov</td>
<td>#1,R19</td>
<td>;;Sll OFFSET,R19</td>
</tr>
<tr>
<td>Or</td>
<td>R19,&lt;eop&gt;</td>
<td>;;No-op</td>
</tr>
<tr>
<td>St</td>
<td>&lt;eop&gt;,0[&lt;ea&gt;]</td>
<td></td>
</tr>
<tr>
<td>BSR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ld</td>
<td>LI-1,R19</td>
<td>;;Sub #1,R15</td>
</tr>
<tr>
<td>Mov</td>
<td>PC,R23</td>
<td>;;Mov R19,PC</td>
</tr>
<tr>
<td>Add</td>
<td>R23,R19</td>
<td></td>
</tr>
<tr>
<td>St</td>
<td>R23,0[R15]</td>
<td></td>
</tr>
<tr>
<td>BTST</td>
<td>Affects condition codes only.</td>
<td></td>
</tr>
<tr>
<td>CAS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Beq</td>
<td>&lt;eop&gt;,Dc,Equal</td>
<td>;;No-op</td>
</tr>
<tr>
<td>Mov</td>
<td>&lt;eop&gt;,Dc</td>
<td></td>
</tr>
<tr>
<td>Bra</td>
<td>Out</td>
<td></td>
</tr>
<tr>
<td>No-op</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Equal</td>
<td>St</td>
<td>Du,0[&lt;ea&gt;]</td>
</tr>
<tr>
<td>Out</td>
<td>No-op</td>
<td></td>
</tr>
<tr>
<td>CAS2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bne</td>
<td>Rn1,Dc1,Nequal1</td>
<td>;;No-op</td>
</tr>
<tr>
<td>No-op</td>
<td>Bra</td>
<td>Out1</td>
</tr>
<tr>
<td></td>
<td>Mov</td>
<td>Du1,Rn1</td>
</tr>
<tr>
<td></td>
<td>Nequal1</td>
<td>Rn1,Dc</td>
</tr>
<tr>
<td></td>
<td>Out1</td>
<td>Bne Rn2,DC2,Nequal2</td>
</tr>
<tr>
<td></td>
<td>No-op</td>
<td>Bra Out2</td>
</tr>
<tr>
<td></td>
<td>Mov</td>
<td>Du2,Rn2</td>
</tr>
<tr>
<td></td>
<td>Nequal2</td>
<td>Mov Rn2,DC2</td>
</tr>
<tr>
<td></td>
<td>Out2</td>
<td>No-op</td>
</tr>
<tr>
<td>CHK</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tlt</td>
<td>Dn,#0,Chk</td>
<td></td>
</tr>
<tr>
<td>Tgt</td>
<td>Dn,&lt;eop&gt;,Chk</td>
<td></td>
</tr>
<tr>
<td>CHK2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tlt</td>
<td>Dn,&lt;eop&gt;,Chk2</td>
<td>;;No-op</td>
</tr>
<tr>
<td>Ld</td>
<td>1[&lt;ea&gt;],&lt;eop&gt;</td>
<td>;;No-op</td>
</tr>
<tr>
<td>No-op</td>
<td>Tgt</td>
<td>Dn,&lt;eop&gt;</td>
</tr>
</tbody>
</table>

- 48 -
CLR          St     #0,0[<ea>]

CMP          Affects condition codes only.

CMPA         Affects condition codes only.

CMPI         Affects condition codes only.

CMPM         Affects condition codes only.

CMP2         Affects condition codes only.

DBCC         Bne     #0,R19,Out
             Sub     #1,Dn
             Bne     #-1,Dn,Label
             Out     No-op

DIVSL.L      Teq     #0,<eop>,Zerodivide
             Sra     #31,Dq,R19
             Xor     R19,Dq,R20
             Sub     R19,R20
             Sra     #31,<eop>,R19
             Xor     R19,<eop>,R20
             Mov     #0,Hi
             Loop    Dstep R20
             Dstep R20
             Bgt     #7,R19,Loop
             Add     #1,R19
             Xor     <eop>,Dq,R19
             Mov     Lo,Dq
             Sub     R19,Dq
             Sra     #31,Hi
             Mov     Hi,R19
             Sra     #31,Dr,R19
             Mov     Hi,R19
             Sub     R19,R20,Dr

DIVUL.L      Teq     #0,<eop>,Zerodivide
             Sra     #31,Dq,R19
             Tne     #0,R19,Overflow
             Mov     Dq,Dr
             Sra     #31,<eop>,R19
             Tne     #0,R19,Overflow
             Mov     #0,Hi
             Loop    Dstep R20
             Dstep R20
             Bgt     #7,R19,Loop
             Add     #1,R19
             Xor     <eop>,Dq,R19
             Mov     Lo,Dq
             Sub     R19,Dq
             Sra     #31,Hi
             Xor     R19,Dq
             Mov     Hi,R19
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Examples</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sra</td>
<td>Move high</td>
<td>Hi,R19</td>
<td>;Add</td>
</tr>
<tr>
<td>Mov</td>
<td>Move</td>
<td>#31,Hi</td>
<td>;And</td>
</tr>
<tr>
<td>Sra</td>
<td>Storage</td>
<td>#31,Dr,R19</td>
<td></td>
</tr>
<tr>
<td>Sub</td>
<td>Subtract</td>
<td>R19,R20,Dr</td>
<td>;Sub</td>
</tr>
<tr>
<td>EOR</td>
<td>Xor</td>
<td>Dn,&lt;eop&gt;</td>
<td>;No-op</td>
</tr>
<tr>
<td></td>
<td>Store</td>
<td>&lt;eop&gt;,0[&lt;ea&gt;]</td>
<td></td>
</tr>
<tr>
<td>EORI</td>
<td>Load</td>
<td>LI,R19</td>
<td>;No-op</td>
</tr>
<tr>
<td></td>
<td>Xor</td>
<td>R19,&lt;eop&gt;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Store</td>
<td>&lt;eop&gt;,0[&lt;ea&gt;]</td>
<td></td>
</tr>
<tr>
<td>EXG</td>
<td>Move</td>
<td>Rx,R19</td>
<td>;Mov</td>
</tr>
<tr>
<td></td>
<td>Move</td>
<td>R19,Ry</td>
<td></td>
</tr>
<tr>
<td>EXT</td>
<td>Sll</td>
<td>#Bits,Dn</td>
<td>;Sra</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>#Bits,Dn</td>
</tr>
<tr>
<td>JMP</td>
<td>Jmp</td>
<td>&lt;ea&gt;</td>
<td>;No-op</td>
</tr>
<tr>
<td>JSR</td>
<td>Move</td>
<td>PC,R19</td>
<td>;Sub</td>
</tr>
<tr>
<td></td>
<td>Store</td>
<td>R19,0[R15]</td>
<td>#1,R15</td>
</tr>
<tr>
<td></td>
<td>Jmp</td>
<td>&lt;ea&gt;</td>
<td>;No-op</td>
</tr>
<tr>
<td>LEA</td>
<td>Move</td>
<td>&lt;eop&gt;,An</td>
<td></td>
</tr>
<tr>
<td>Link</td>
<td>Store</td>
<td>An,−1[R15]</td>
<td>;Sub</td>
</tr>
<tr>
<td></td>
<td>Move</td>
<td>R15,An</td>
<td>#1,R15</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>;Add</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>&lt;eop&gt;,R15</td>
</tr>
<tr>
<td>LSL</td>
<td>Sll</td>
<td>SI−1,#1,&lt;eop&gt;</td>
<td>;Mov</td>
</tr>
<tr>
<td></td>
<td>Sll</td>
<td>#31,R19</td>
<td>#1,R19</td>
</tr>
<tr>
<td></td>
<td>Srl</td>
<td>#31,R19</td>
<td>;And</td>
</tr>
<tr>
<td></td>
<td>And</td>
<td>#14,R16</td>
<td>&lt;eop&gt;,R19</td>
</tr>
<tr>
<td></td>
<td>Store</td>
<td>&lt;eop&gt;,0[&lt;ea&gt;]</td>
<td>;Or</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>R19,R16</td>
</tr>
<tr>
<td>LSR</td>
<td>Srl</td>
<td>SI−1,&lt;eop&gt;</td>
<td>;And</td>
</tr>
<tr>
<td></td>
<td>And</td>
<td>#1,&lt;eop&gt;,R19</td>
<td>#14,R16</td>
</tr>
<tr>
<td></td>
<td>Store</td>
<td>&lt;eop&gt;,0[&lt;ea&gt;]</td>
<td>;Srl</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>#1,&lt;eop&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>;Or</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>R19,R16</td>
</tr>
<tr>
<td>MOVE</td>
<td>Store</td>
<td>&lt;eop1&gt;,0[&lt;ea2&gt;]</td>
<td></td>
</tr>
<tr>
<td>MOVEA</td>
<td>Store</td>
<td>An,0[&lt;ea&gt;]</td>
<td></td>
</tr>
<tr>
<td>MOVECCR</td>
<td>Store</td>
<td>R16,0[&lt;ea&gt;]</td>
<td></td>
</tr>
<tr>
<td>MOVESR</td>
<td>Store</td>
<td>R16,0[&lt;ea&gt;]</td>
<td></td>
</tr>
<tr>
<td>MOVEUSP</td>
<td>Store</td>
<td>R15,0[&lt;ea&gt;]</td>
<td></td>
</tr>
<tr>
<td>MOVEM</td>
<td>Store</td>
<td>R0,0[&lt;ea&gt;]</td>
<td>;Sub</td>
</tr>
<tr>
<td></td>
<td>Store</td>
<td>R1,0[&lt;ea&gt;]</td>
<td>#1,&lt;ea&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
NOTE: The translator will remove the Store instruction for any register not selected to be saved. This only involves checking the register list.

### MOVEP

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xc #3,Dn,R19</td>
<td>;Sll #8,R19</td>
</tr>
<tr>
<td>Xc #2,Dn,R20</td>
<td>;Sll #24,R20</td>
</tr>
<tr>
<td>Xc #1,Dn,R22</td>
<td>;Or R20,R19</td>
</tr>
<tr>
<td>St R19,0[&lt;ea&gt;]</td>
<td>;Sll #8,R22</td>
</tr>
<tr>
<td>Xc #0,Dn,R19</td>
<td>;Sll #24,R19</td>
</tr>
<tr>
<td>Or R20,R19</td>
<td>;No-op</td>
</tr>
<tr>
<td>St R19,1[&lt;ea&gt;]</td>
<td>;</td>
</tr>
</tbody>
</table>

### MOVEQ

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mov LI,Dn</td>
<td>;</td>
</tr>
</tbody>
</table>

### MULS.L

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mov #0,Hi</td>
<td>;Mov Dn,Lo</td>
</tr>
<tr>
<td>Msetup &lt;eop&gt;</td>
<td>;Mstep &lt;eop&gt;</td>
</tr>
<tr>
<td>Mstep &lt;eop&gt;</td>
<td>;Mstep &lt;eop&gt;</td>
</tr>
<tr>
<td>Mstep &lt;eop&gt;</td>
<td>;Mstep &lt;eop&gt;</td>
</tr>
<tr>
<td>Mstep &lt;eop&gt;</td>
<td>;Mstep &lt;eop&gt;</td>
</tr>
<tr>
<td>Mstep &lt;eop&gt;</td>
<td>;Mstep &lt;eop&gt;</td>
</tr>
<tr>
<td>Mstep &lt;eop&gt;</td>
<td>;Mstep &lt;eop&gt;</td>
</tr>
<tr>
<td>Mstep &lt;eop&gt;</td>
<td>;Mstep &lt;eop&gt;</td>
</tr>
<tr>
<td>Mov Lo,Dn</td>
<td>;Mov Hi,R19</td>
</tr>
<tr>
<td>Sra #31,Dn</td>
<td>;Xor Dn,R19</td>
</tr>
<tr>
<td>Mov Lo,Dn</td>
<td>;No-op</td>
</tr>
<tr>
<td>The #0,R19,Overflow</td>
<td>;No-op</td>
</tr>
</tbody>
</table>

### MULU.L

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mov #0,Hi</td>
<td>;Mov Dn,Lo</td>
</tr>
<tr>
<td>Msetup &lt;eop&gt;</td>
<td>;Mstep &lt;eop&gt;</td>
</tr>
<tr>
<td>Mstep &lt;eop&gt;</td>
<td>;Mstep &lt;eop&gt;</td>
</tr>
<tr>
<td>Mstep &lt;eop&gt;</td>
<td>;Mstep &lt;eop&gt;</td>
</tr>
<tr>
<td>Mstep &lt;eop&gt;</td>
<td>;Mstep &lt;eop&gt;</td>
</tr>
<tr>
<td>Mstep &lt;eop&gt;</td>
<td>;Mstep &lt;eop&gt;</td>
</tr>
<tr>
<td>Mstep &lt;eop&gt;</td>
<td>;Mstep &lt;eop&gt;</td>
</tr>
<tr>
<td>Mstep &lt;eop&gt;</td>
<td>;Mstep &lt;eop&gt;</td>
</tr>
<tr>
<td>Umend &lt;eop&gt;</td>
<td>;Mov Lo,Dn</td>
</tr>
</tbody>
</table>
NBCD
UNPK <eop>,<eop> [ MC68020 instruction ]
Not <eop> ;;No-op
PACK <eop>,0[<ea>] [ MC68020 instruction ]

NEG
Subr #0,<eop>> ;;No-op
St <eop>,0[<ea>] ;;

NEGX
Mov #16,R19 ;;Subr #0,<eop>
And R16,R19 ;;Srl #4,R19
Sub R19,<eop> ;;No-op
St <eop>,0[<ea>] ;;

NOP
No-op ;;No-op

NOT
Not <eop> ;;No-op
St <eop>,0[<eop>] ;;

OR
Or Dn,<eop> ;;No-op
St <eop>,0[<ea>] ;;

ORI
Ld LI,R19 ;;No-op
Or R19,<eop> ;;No-op
St <eop>,0[<ea>] ;;

PACK
Ld LI,R19 ;;No-op
Xc #1,<eop1>,R20 ;;Add R20,R19
Xc #2,<eop1>,R20 ;;Sll #8,R20
Add R20,R19 ;;No-op
Mov #15,R20 ;;And R20,R19
Sll #8,R20 ;;And R20,R19
Xc #1,R19,R20 ;;Sll #4,R20
Xc #0,R19 ;;Add R20,R19
St R19,0[<ea2>] ;;No-op

PEA
St <ea>,-1[R15] ;;Sub #1,R15

RESET
Privileged instruction - not available in user mode.

ROL
Mov #1,R19 ;;Rol SL,<eop>
And <eop>,R19 ;;And #14,R16
St <eop>,0[<ea>] ;;Or R19,R16

ROR
Mov #128,R19 ;;Sll #24,R19
Rol #32-SI,<eop> ;;And <eop>,R19
Srl #31,R19 ;;And #14,R16
St <eop>,0[<ea>] ;;Or R19,R16

ROXL
Mov #1,R19 ;;Sll #31-SI,R19
And <eop>,R19 ;;Not R19,R20
And R20,<eop> ;;Srl #27-SI,R19
Mov #16,R20 ;;And R16,R20
Sll #27-SI,R20.;;Or R20,<eop>
Mov #18,R20.;;And R20,R16
Sne #0,R19
And R20,R19.;;Roi SI,<eop>
St <eop>,0[<ea>];;Or R19,R16

ROXR
Mov #1,R19.;;Sll SI-#1,R19
And <eop>,R19.;;Not R19,R20
And R20,<eop>.;;Srl SI-#4,R19
Mov #16,R20.;;And R16,R20
Sll SI-#4,R20.;;Or R20,<eop>
Mov #18,R20.;;And R20,R16
Sne #0,R19
And R20,R19.;;Roi #32-SI,<eop>
St <eop>,0[<ea>];;Or R19,R16

RTD
Ld 0[R15],R19.;;Add #1,R15
Ld LI,R20
Mov R15,PC.;;Add R20,R15

RTE
Privileged instruction - not available in user mode.

RTR
Ld 0[R15],R16.;;Add #1,R15
Ld 0[R15],R19.;;Add #1,R15
No-op.;;Mov R19,PC

RTS
Ld 0[R15],R19.;;Add #1,R15
No-op.;;Mov R19,PC

SBCD
UNPK <eop1>,R20 [ MC68020 instruction ]
UNPK <eop2>,<eop2> [ MC68020 instruction ]
Mov #1,R19.;;Sll #4,R19
And R16,R19.;;Sra #4,R19
Sub R20,<eop2>.;;Sub R19,R20
PACK R20,<ea2> [ MC68020 instruction ]

SCC
Sne #0,R19,<ea>

STOP
Privileged instruction - not available in user mode.

SUB
Sub Dn,<eop>.;;No-op
St <eop>,0[<ea>];;

SUBA
Sub <eop>,An ;;

SUBI
Ld LI,R19
Subr <eop>,R19.;;No-op
St R19,0[<ea>];;

SUBQ
Sub SI,<eop>.;;No-op
St <eop>,0[<ea>];;
**SUBX**
Mov  #1,R19  ;;Sll  #4,R19
And  R16,R19  ;;Sra  #4,R19
Sub  <eop1>,<eop2>  ;;Sub  R19,<eop2>
St   <eop2>,0[<ea2>]  ;

**SWAP**
Xc   #1,Dn,R19  ;;Sll  #24,R19
Xc   #0,Dn,R20  ;;Sll  #16,R20
Or   R20,R19  ;;Srl  #16,Dn
Or   R19,Dn  ;

**TAS**
Mov  #1,R19  ;;Sll  #7,R19
Or   R19,<eop>  ;;No-op
St   <eop>,0[<ea>]  ;

**TRAP**
Mov  #0,R19  ;;No-op
Teq  #0,R19,Trap

**NOTE:** This trap is always taken. 11 bits are available to pass the vector.

**TRAPCC**
Tne  #0,R19,Trap

**TRAPV**
Tne  #0,R19,Overflow

**TST**
Affects condition codes only.

**UNLK**
Mov  An,R15  ;;No-op
Ld   0[R15],An  ;;Add  #1,R15
No-op  ;

**UNPK**
Xc   #0,<eop>,R19  ;;Mov  #15,R22
Sll  #4,R19,R20  ;;And  R22,R19
Sll  #8,R22  ;;And  R22,R20
Ld   LI,R22  ;
Or   R20,R19  ;;Add  R22,R19
Appendix D

QUICKSORT BENCHMARK TRANSLATION

D.1 The Purpose of this Appendix

This Appendix was included for several reasons. First, the translation of MC68020 assembly language code is illustrated to aid the reader in understanding the nature of a translation. Secondly, it was incorporated to help numerically evaluate the efficiency of a typical application program for both the assembly version and the translated versions on these two microprocessors. Although the compiled versions are compared in Figures 1 and 2 for a program written in pascal to sort integer words, the actual efficiency is somewhat related to the compiler's efficiency. These routines were written directly in assembly code and are therefore a better reflection on the machine's performance than on the compiler writer's ability.

The benchmark chosen to test the two architectures is the Carnegie Mellon Benchmark known as Quicksort. The resulting MC68020 code is an adapted version of MC6800L10 code written by Motorola that first appeared in [GraJe81]. The SU-MIPS assembly code appears here for the first time in code written directly in assembly and in code derived from a translation of the MC68020 code. The translated version was then optimized using the translation specifications outlined in the text with global register optimization used to overcome the limitations on the register file's size.

The code was evaluated for both program size and the resulting execution time for a 4 MHz. SU-MIPS and a 10 MHz. MC68020. The test data for this benchmark consists of 102 (N=100) records, each 16 bytes long. Parameter M is set to nine. The records are as follows:

```
Record 0     ---  00  00  00  00  00  00  00  00  00  00  00
Record 1     ---  FF  00  00  00  00  00  00  00  00  00  00
Record 2     ---  FE  00  00  00  00  00  00  00  00  00  00
Record 3     ---  FD  00  00  00  00  00  00  00  00  00  00
  ...            ...  ...  ...  ...  ...  ...  ...  ...  ...  ...
Record 100   ---  9C  00  00  00  00  00  00  00  00  00  00
Record 101   ---  FF  FF  FF  FF  FF  FF  FF  FF  FF  FF
```

Notice that only key values (bytes 3 to 9 in each record) are significant. All data values are hexadecimal bytes.
D.2 Pseudocode used for the Quicksort Benchmark

procedure QUICKSORT(N,M,REC,STACK)

integer L,U,I,J
integer array STACK[0:2*F(N)-1]
character string V
L := 1; U := N

do forever
I := L; J := U+1; V := REC[L]
do forever
   do I := I+1 until REC[I] >= V end-do
   do J := J-1 until REC[J] <= V end-do
   if J > I
      then swap REC[I] with REC[J]
      else goto end-first
   end-if
end-do
end-first:
   swap REC[L] with REC[J]
   if both subfile sizes (J-L and U-J) <= M
      then
         if stack is empty
            then goto end-outer
         else pop L and U from stack
         end-if
      else
         if smaller subfile size (J-L or U-J) <= M
            then set L and U to lower and upper limit of larger subfile
         else
            push lower and upper limits of larger subfile onto stack
            set L and U to limits of smaller subfile
         end-if
      end-if
   end-do
end-outer:
   do for I from N-1 to 1 in steps of 1
   if REC[I] > REC[I+1] then
      V := REC[I]; J := J+1
   do forever
      REC[J-1] := REC[J]; J := J+1
      if REC[J] >= V then goto end-last
   end-do
end-last:
   REC[J-1] := V
end-if
end-do
D.3 MC68020 Assembly Language Code for Quicksort

*************************************************************************
*                                                               *
* MC68020 Carnegie Mellon Benchmark I                            *
*                                                               *
* QUICKSORT                                                      *
*                                                               *
* Attributes: 4 Gigabyte Address Range                          *
* Position Independent                                          *
* Reentrant                                                     *
*                                                               *
* Input: D0 - "N" Record Count                                  *
* D1 - "M" Threshold for Insertion Sort                         *
* A0 - "REC" Address of the Sort Array                          *
* A7 - "STACK" Stack Address                                    *
*                                                               *
* Output: The Sort Data Array is Sorted                         *
*                                                               *
* All Registers are Transparent over this Routine                *
*                                                               *
* Lines: 87                                                     *
* Bytes: 266                                                    *
*                                                               *
*************************************************************************

equates

LENGTH EQU 16 sort entry record length
KEY EQU 3 offset to key within record
KEYLEN EQU 7 sort key length

**************************** quicksort subroutine entry ****************************

quicksort subroutine entry

*************************************************************************
* QUICK MOVEM.L D0-D7/A0-A6,-(SP) save all registers *
* MOVEL D0,D2 copy number of records over *
* LSL.L #4,D0 calc. ptr. to last record *
* LEA -LENGTH(A0,D0.L),A1 A1 <- ptr. to last record = U *
* LSL.L #4,D1 find total size of M records *
* MOVE.L D1,A6 keep value in A6 for later *
* MOVEM.L D0/D2/A1,-(SP) save dummy, count, top of stack *
* CLR.L -(SP) mark sort stack empty *
*
quicksort phase

Register use: A0 -> first record of the subfile
A1 -> last record of the subfile
A2 & A3 -> key pointers
A4 & A5 -> work pointers
A6 -> length of the "M" records
SP -> recursive call arguments

SORT
LEA KEY(A0),A2 A2 -> KEY(I) = REC(L)
LEA LENGTH+KEY(A1),A3 A3 -> KEY(J) = REC(U+1)

LOOP1
LEA KEY(A0),A4 A4 -> V for current record
LEA LENGTH(A2),A2 I <- I+1
MOVE.L A2,A5 A5 temp for I
MOVE.L #KEYLEN-1,D0 D0 = loop counter

CMP1
CMPM.B (A5)+,(A4)+ compare V=REC(I)
DBNE D0,CMP2 loop while equal

LOOP2
LEA KEY(A0),A4 A4 -> KEY(V) of current record
LEA -LENGTH(A3),A3 J <- J-1
MOVE.L A3,A5 A5 = temp for J
MOVE.L #KEYLEN-1,D0 loop counter

CMP2
CMPM.B (A4)+,(A5)+ compare REC(J)-V
DBNE D0,CMP2 loop while equal
BHI LOOP2 if REC(I) < V continue compare

BCC END1ST branch if I >= J
MOVM.L -KEY(A2),D0-D3 swap
MOVM.L -KEY(A3),D4-D7 . . REC(J)
MOVM.L D0-D3,KEY(A3) . . . with
MOVM.L D4-D7,KEY(A2) . . . REC(I)

BRA LOOP1 continue

END1ST
SUB.L #KEY,A3 sub. key - get beg. of record
MOVM.L (A0),D0-D3 swap
MOVM.L (A3),D4-D7 . . REC(L)
MOVM.L D0-D3,(A3) . . . with
MOVM.L D4-D7,(A0) . . . REC(J)
MOVE.L A1,D1 D1 <- U
MOVE.L A3,D2 D2 <- J
SUB.L D2,D1 D1 <- U-J
**decide subfile direction**

*  

**fall into insertion sort as all subfiles below**  
*  

**note: stack space is reserved for "v" key compare record copies**  
*
* LOOP
  LEA -LENGTH(A0),A0
  LEA KEY(A0),A2
  LEA LENGTH+KEY(A0),A3
  MOVE.L #KEYLEN-1,D1
  CMPVJ CMPM.B (A3)+(A2)+
  DBNE D1,CMPVJ
  BHI
  LOOPIN
  MOVE.L (A0),D5-D7/A5
  MOVE.L D5-D7,(SP)
  LEA MOVE.L LENGTH(A0),A1
  MOVE.L A0,A4
  MOVEM.L (A1),D1-D4
  MOVE.L D1-D4,(A4)
  LEA MOVE.L LENGTH(A1),A1
  LEA KEY(SP),A2
  LEA KEY(A1),A3
  MOVE.L #KEYLEN-1,D1
  CMPVJ CMPM.B (A3)+(A2)+
  DBNE D1,CMPVJ
  BHI
  LOOPIN
  MOVEM.L D5-D7/A5,(A4)
  ENDIF DBRA
  UNLK
  RTS
  END

I <= I-1
A2 -> KEY(I)
A3 -> KEY(I+1)
loop counter for compare
compare KEY(I) & KEY(I+1)
loop while equal
branch if KEY(I) <= KEY(I+1)

V <- REC(I)
add on stack for key compare
A1 -> REC(J) = REC(I+1)
prime A4 -> REC(J-1)
temp <- REC(J)
REC(J-1) <- temp
A4 -> REC(J-1)
J = J+1
A2 -> KEY(V)
A3 -> KEY(J)
loop counter in D1
compare KEKY(V) & KEY(J)
loop while equal
if KEY(V) > KEY(J) keep looping
REC(J-1) <- V
continue linear insert
free and restore stack
restore registers
return to caller
D.4 SU-MIPS Assembly Language Code for Quicksort

*************************************************************************
*
SU-MIPS Carnegie Mellon Benchmark I
*
*
QUICKSORT - Assembly Version
*
*
Attributes: 67 Megabytes Address Range
Position Independent
Reentrant
*
*
Input: R0 - "N" Record Count
R1 - "M" Threshold for Insertion Sort
R2 - "REC" Address of the Sort Array
R3 - "STACK" Stack Address Workspace
*
*
R3 -> Return Address of the Calling Program
*
*
Register: R4 - "REC+L" Lower Limit of the Subfile
Use: R5 - "REC+U" Upper Limit of the Subfile
R6 - "REC+I" Index Pointer
R7 - "REC+J" Index Pointer
R8 - Scratch Register for Compares, etc.
R9 - Scratch Register for Compares, etc.
R10 - Scratch Register for Compares, etc.
R11 - LSW "V"
R12 - ""
R13 - MSW "V"
R14 - Swap Register
R15 - Swap Register
*
*
Output: The Sort Data Array is Sorted
*
*
Lines: 131
Bytes: 524
*
*************************************************************************
*
quicksort subroutine entry
*
*************************************************************************
*
save registers and initialize L and U
*
QUICK St R0,-1[R3] ;;Mov R0,R0
St R1,-2[R3] ;;Sll #4,R0

- 61 -
St R2,-3[R3] ;;Sll #4,R1
St R4,-4[R3] ;;Mov R2,R4
St R5,-5[R3] ;;Add #4,R4
St R6,-6[R3] ;;Sub #7,R3
St R7,0[R3] ;;Mov R2,R5
St R8,-1[R3] ;;Add R0,R5
St R9,-2[R3] ;;Mov R0,R0
St R10,-3[R3] ;;Mov R0,R0
St R11,-4[R3] ;;Mov R0,R0
St R12,-5[R3] ;;Mov R0,R0
St R13,-6[R3] ;;Sub #7,R3
St R14,0[R3] ;;Mov R0,R0
St R15,-1[R3] ;;Mov R0,R0
St #0,-2[R3] ;;Sub #3,R5

******************************************************************************
* quicksort phase
******************************************************************************

* get V=REC[L], set I=L, set J=U+4

SORT   Ld 0[R4],R13 ;;Mov R4,R6
Ld 2[R4],R11 ;;Xc #0,R13
Ld 1[R4],R12 ;;Srl #16,R11
Add #4,R5,R7 ;;Sll #16,R11

* set I=I+4, find REC[I] >= V

NEXTI  Ld 4[R6],R10 ;;Add #4,R6
Ld 1[R6],R9 ;;Xc #0,R10
Blt R10,R13,NEXTI
Ld 2[R6],R8 ;;Mov R0,R0
Blt R9,R12,NEXTI
Srl #16,R8 ;;Sll #16,R8
Blt R8,R11,NEXTI

* set J=J-4, find REC[J] <= V

NEXTJ  Ld -4[R7],R10 ;;Sub #4,R7
Ld 1[R7],R9 ;;Xc #0,R10
Bgt R10,R13,NEXTJ
Ld 2[R7],R8 ;;Mov R0,R0
Bgt R9,R12,NEXTJ
Srl #16,R8 ;;Sll #16,R8
Bgt R8,R11,NEXTJ

* if J>I then swap REC[I] with REC[J] and goto SORT, else branch out

Ld 0[R7],R10 ;;Mov R0,R0
Ble R7,R6,END1ST
Ld 0[R6],R15 ;;Mov R0,R0
Ld 1[R6],R14 ;; Mov R0,R0
St R15,0[R7] ;; Mov R0,R0
St R14,1[R7] ;; Mov R0,R0
St R10,0[R6] ;; Mov R0,R0
St R9,1[R6] ;; Mov R0,R0
Ld 2[R6],R15 ;; Mov R0,R0
Ld 3[R8],R14 ;; Mov R0,R0
Ld 2[R7],R10 ;; Mov R0,R0
Ld 3[R7],R9 ;; Mov R0,R0
St R15,2[R7] ;; Mov R0,R0
St R14,3[R7] ;; Mov R0,R0
St R10,2[R6] ;; Mov R0,R0
St R9,3[R6] ;; Mov R0,R0
Bra SORT

******************************************************************************************
* new subfile found, now determine the next stage
******************************************************************************************
* this routine means I>J so first swap V with REC[J],
* then branch to NEWLU if [(J-L) or (U-J)] > M
* END1ST
Ld 0[R4],R13 ;; Mov R0,R0
Ld 2[R4],R11 ;; Mov R0,R0
Ld 3[R4],R15 ;; Mov R0,R0
Ld 0[R7],R10 ;; Mov R0,R0
Ld 2[R7],R8 ;; Mov R0,R0
Ld 3[R7],R14 ;; Mov R0,R0
St R8,2[R4] ;; Mov R0,R0
St R9,1[R4] ;; Mov R0,R0
St R10,0[R4] ;; Mov R0,R0
St R11,2[R7] ;; Mov R0,R0
St R12,1[R7] ;; Mov R7,R8
St R13,0[R7] ;; Sub R4,R8
St R14,3[R4] ;; Mov R5,R9
Bgt R8,R1,NEWLU
St R15,3[R7] ;; Sub R7,R9
Bgt R9,R1,NEWL

* entry here means that [(J-L) and (U-J)] <= M
* Ld 1[R3],R10 ;; Mov R0,R0
Beq #0,R9,OUTER
Ld 2[R3],R11 ;; Mov R0,R0
Mov R10,R4 ;; Mov R11,R5
Bra SORT
Add #2,R3 ;; Mov R0,R0
* *********************
  *
  * decide subfile direction
  *
  * entry here means that (J-L) > M
  *
NEWLU  Bit  R9,R1,NEWU
  St   R7,-1[R3]      ;;Mov  R0,R0
  Bit  R8,R9,STACK
  St   R5,0[R3]       ;;Sub  #2,R3
  *
  * entry here means that (J-L) > M and (U-J) > M
  *
NEWU   Bra  SORT
  Sub  #4,R7,R5      ;;Mov  R0,R0
  *
  * STACK - entry here means that (J-L) > M and (J-L) < (U-J)
  * NEWL  - entry here means that (J-L) <= M and (U-J) > M
  *
STACK  St   R4,2[R3]     ;;Mov  R0,R0
NEWL   Bra  SORT
  Add  #4,R7,R4       ;;Mov  R0,R0
  *
*  ******************************************************

*  fall into insertion sort as all subfiles below
*  or equal M records - insertion sort phase
*  ******************************************************

* OUTER  Add  R0,R2,R6      ;;Add  #4,R7
LOOP   Ld    -4[R6],R13   ;;Sub  #4,R6
        Ld    4[R6],R10    ;;Xc   #0,R13
        Ld    1[R6],R12    ;;Xc   #0,R10
        Bge   R10,R13,LOOP
        Ld    5[R6],R9     ;;Mov  R0,R0
        Bge   R9,R12,LOOP
        Ld    2[R6],R11    ;;Mov  R0,R0
        Ld    6[R6],R8     ;;Srl  #16,R11
        Sll   #16,R11     ;;Srl  #16,R8
        Sll   #16,R8      ;;Sub  #4,R7
        Bge   R8,R12,LOOP
        *
*  now set J=J+4, set REC[J-4] = REC[J], set J=J+4
*  then check to see if REC[J] >= V, branch if so

NEXT   Ld    4[R7],R10    ;;Add  #4,R7
        Ld    1[R7],R9     ;;Mov  R0,R0
St R10,-4[R7] ;;Mov R0, R0
Ld 2[R7],R8 ;;Xc #0, R10
Ld 3[R7],R14 ;;Mov R0, R0
St R9,-3[R7] ;;Mov R0, R0
St R8,-2[R7] ;;Mov R0, R0
Bge R10,R13,LAST
St R14,-1[R7] ;;Srl #16, R8
Bge R9,R12,LAST
Sll #16, R8 ;;Mov R0, R0
Blt R8,R11,NEXT
Mov R0, R0

* since REC[J] >= V, move V into REC[J-1] and goto LOOP if I <= REC+4
*

LAST Ld 0[R6],R13 ;;Mov R0, R0
Ld 2[R6],R11 ;;Mov R0, R0
Ld 3[R6],R14 ;;Mov R0, R0
St R13,-4[R7] ;;Mov R0, R15
St R12,-3[R7] ;;Add R2, R15
St R11,-2[R7] ;;Add #4, R15
Ble R6,R15,LOOP
St R14,-1[R7] ;;Mov R0, R0

* now the routine is done, so exit after restoring the registers
*

Ld R15,1[R3] ;;Mov R0, R0
Ld R14,2[R3] ;;Mov R0, R0
Ld R13,3[R3] ;;Mov R0, R0
Ld R12,4[R3] ;;Mov R0, R0
Ld R11,5[R3] ;;Mov R0, R0
Ld R10,6[R3] ;;Mov R0, R0
Ld R9,7[R3] ;;Add #7, R3
Ld R8,1[R3] ;;Mov R0, R0
Ld R7,2[R3] ;;Mov R0, R0
Ld R6,3[R3] ;;Mov R0, R0
Ld R5,4[R3] ;;Mov R0, R0
Ld R4,5[R3] ;;Mov R0, R0
Ld R2,6[R3] ;;Add #7, R3
Ld R1,0[R3] ;;Add #2, R3
Ld 0[R3],R3 ;;Mov R0, R0
Ld R0,-1[R3] ;;Mov R3,PC

End
**SU-MIPS Carnegie Mellon Benchmark I**

**QUICKSORT - Translated Version**

**Attributes:** 67 Megabyte Address Range
Position Independent
Reentrant

**Input:**
D0 - "N" Record Count
D1 - "M" Threshold for Insertion Sort
A0 - "REC" Address of the Sort Array

**Output:**
The Sort Data Array is Sorted

**All Registers are Transparent over this Routine**

_lines: 165_
_bytes: 660_

---

**equates**

LENGTH EQU 16  sort entry record length
KEY EQU 3  offset to key within record
KEYLEN EQU 7  sort key length

---

**quicksort subroutine entry**

---

**MOVEM.L**
D0-D7/A0-A6,-(SP)  save all registers

**QUICK**
St R14,-1[R15]  ;;Sub #1,R15
St R13,-1[R15]  ;;Sub #1,R15
St R12,-1[R15]  ;;Sub #1,R15
St R11,-1[R15]  ;;Sub #1,R15
St R10,-1[R15]  ;;Sub #1,R15
St R9,-1[R15]  ;;Sub #1,R15
St R8,-1[R15]  ;;Sub #1,R15
St R7,-1[R15]  ;;Sub #1,R15

---

- 66 -
* St  R6,-1[R15] ;;Sub #1,R15
* St  R5,-1[R15] ;;Sub #1,R15
* St  R4,-1[R15] ;;Sub #1,R15
* St  R3,-1[R15] ;;Sub #1,R15
* St  R2,-1[R15] ;;Sub #1,R15
* St  R1,-1[R15] ;;Sub #1,R15
* St  R 0,-1[R15] ;;Sub #1,R15

* MOVE.L D0,D2 copy number of records over
* LSL.L #4,D0 calc. ptr. to last record

* Mov  R0,R2 ;;Sll #4,R0
* LEA  -LENGTH(A0,D0,L),A1 A1 <- ptr. to last record = U
* LSL.L #4,D1 find total size of M records

* Mov  -LENGTH,R9 ;;Add R8,R9
* Add  R0,R9 ;;Sll #4,R1
* MOVEM.L D0/D2/A1,-(SP) save dummy, count, top of stack
* MOVE.L D1,A6 keep value in A6 for later

* St  R9,-1[R15] ;;Sub #1,R15
* St  R2,-1[R15] ;;Sub #2,R15
* St  R0,0[R15] ;;Mov R1,R14

* CLR.L -(SP) mark sort stack empty
* St  #0,-1[R15] ;;Sub #1,R15

**********************************************************************

* quicksort phase

**********************************************************************

* Register use: A0 -> first record of the subfile
* A1 -> last record of the subfile
* A2 & A3 -> key pointers
* A4 & A5 -> work pointers
* A6 -> length of the "M" records
* SP -> recursive call arguments

* LEA  KEY(A0),A2 A2 -> KEY(I) = REC(L)
* SORT Add  KEY,R8,R10 ;;Mov R0,R0
* LEA  LENGTH+KEY(A1),A3 A3 -> KEY(J) = REC(U+1)
* Mov  LENGTH+KEY,A3 ;;Add A1,A3
* LEA  KEY(A0),A4 A4 -> V for current record


```
LOOP1
  * Add KEY,R8,R12   ;;Mov R0,R0
  * LEA LENGTH(A2),A2
  * MOV.E LENGTH(A2),A2
  * MOV.E A2,A5
  * Mov LENGTH,R5   ;;Add R10,R5
  * Mov R5,R10      ;;Mov #KEYLEN-1,R0
  * CMP.M.B (A5)+(A4)+
  * DBNE D0,CMP1
  * BHI LOOP1

CMP1
  Ld [R13>>#2],R1    ;;Not R13,R3
  Ld [R12>>#2],R2    ;;Xc R3,R1
  Not R12,R4         ;;Xc R4,R2
  Bne R2,R1,CMP1     
  Add #1,R12         ;;Add #1,R13
  Beq #0,R0,CMP1     ;;Mov R0,R0
  Sub #1,R0          ;;Mov R0,R0
  Bhi R2,R1,LOOP1    
  Mov R0,R0          

  * LEA KEY(A0),A4    
  * LOOP2
  * Add KEY,R8,R12   ;;Mov R0,R0
  * LEA -LENGTH(A3),A3
  * MOVE.L A3,A5
  * MOV.E #KEYLEN-1,D0
  * Mov LENGTH,R5   ;;Sub R5,R11
  * Mov R11,R5      ;;Mov #KEYLEN-1,R0
  * CMP.M.B (A4)+(A5)+
  * DBNE D0,CMP2
  * BHI LOOP2

CMP2
  Ld [R12>>#2],R1    ;;Not R12,R3
  Ld [R13>>#2],R2    ;;Xc R3,R1
  Not R13,R4         ;;Xc R4,R2
  Bne R2,R1,CMP2     
  Add #1,R12         ;;Add #1,R13
  Beq #0,R0,CMP2     ;;Mov R0,R0
  Sub #1,R0          ;;Mov R0,R0
  Bhi R2,R1,LOOP2    
  Mov R0,R0          

  * CMP.L A3,A2
  * BCC ENDRST
  * Bhi A2,A3,ENDIST

I <= I+1
D0 = loop counter
A5 temp for I
I = J
branch if I >= J
```
**MOVEM.L**  -KEY(A2),D0-D3

```
Ld   -3[R10],R0  ;;Add  #1,R10
Ld   0[R10],R1   ;;Add  #4,R10
Ld   0[R10],R2   ;;Mov  R0,R0
Ld   4[R10],R3   ;;Sub  #5,R10
```

**MOVEM.L**  -KEY(A3),D4-D7

```
Ld   -3[R11],R4  ;;Add  #1,R11
Ld   0[R11],R5   ;;Add  #4,R11
Ld   0[R11],R6   ;;Mov  R0,R0
Ld   4[R11],R7   ;;Sub  #5,R11
```

**MOVEM.L**  D0-D3,-KEY(A3)  with
```
St   R0,-3[R11]   ;;Add  #1,R11
St   R1,0[R11]    ;;Add  #4,R11
St   R2,0[R11]    ;;Mov  R0,R0
St   R3,4[R11]    ;;Sub  #5,R11
```

**MOVEM.L**  D4-D7,-KEY(A2)  REC(J)

```
St   R4,-3[R10]   ;;Add  #1,R10
St   R5,0[R10]    ;;Add  #4,R10
St   R6,0[R10]    ;;Mov  R0,R0
Bra  LOOP1
St   R7,4[R10]    ;;Sub  #5,R10
```

*** new subfile found, now determine the next stage ***

**SUB.L**  #KEY,A3  sub. key - get beg. of record
**MOVEM.L**  (A0),D0-D3  swap

```
END1ST Ld 0[R8],R0  ;;Add  #4,R8
Ld 0[R8],R0  ;;Add  #4,R8
Ld 0[R8],R0  ;;Add  #4,R8
Ld 0[R8],R0  ;;Sub  #KEY,R11
```

**MOVEM.L**  (A3),D4-D7  REC(L)

```
Ld 0[R11],R4  ;;Add  #4,R11
Ld 0[R11],R5  ;;Add  #4,R11
Ld 0[R11],R6  ;;Add  #4,R11
Ld 0[R11],R7  ;;Mov  R0,R0
```

**MOVEM.L**  D0-D3,(A3)  with
**MOVE.L**  A1,D1  D1 <- U
St R3,0[R11] ;;Sub #4,R11
St R2,0[R11] ;;Sub #4,R11
St R1,0[R11] ;;Sub #4,R11
St R0,0[R11] ;;Mov R9,R1

* MOVEM.L D4-D7,(A0) . . . REC(J)

* MOVE.L A3,D2

St R7,0[R8] ;;Sub #4,R8
St R6,0[R8] ;;Sub #4,R8
St R5,0[R8] ;;Sub #4,R8
St R4,0[R8] ;;Mov R11,R2

* SUB.L D2,D1 D2 <- U-J
* SUB.L A0,D2 D2 <- J-L
* CMP.L A6,D2 compare (J-L) <= MSIZE
* BHI NEWLU branch if no

Bhi R14,R2,NEWLU
Sub R2,R1 ;;Sub R8,R2

* CMP.L A6,D1 compare (U-J) <= MSIZE
* BHI NEWL branch if no

Bhi R14,R1,NEWL

* MOVEM.L (SP)+,A0/A1 pop next L and U from stack
* MOVE.L A0,D0 test if stack is empty
* BNE SORT continue if sort is not empty

Ld 0[R15],R8 ;;Mov R0,R0
Bne #0,R8,SORT
Ld 1[R15],R9 ;;Mov R8,R0

--------------------------------------------------------------------------

* decide subfile direction

--------------------------------------------------------------------------

* CMP.L A6,D1 (U-J) <= MSIZE? (U-J) smaller?
* BLE NEWU branch if so

NEWLU Ble R14,R1,NEWU
Mov R0,R0 ;;Mov R0,R0

* CMP.L D1,D2 determine smaller subfile
* BCS STACK branch if (J-L) is smaller

Blt R1,R2,STACK

* MOVE.L A1,-(SP) stack U

- 70 -
* MOVE.L A3,-(SP) stack J
* (U-J) subfile smaller, set
* L & U to larger subfile limits
* St R9,-1[R15] ;;Mov R0,R0
* St R11,-2[R15] ;;Sub #2,R15
* LEA -LENGTH(A3),A1 U <- J-1, L stays the same
* BRA SORT continue the sort
* (J-L) subfile smaller, set
* L & U to larger subfile limits
* NEWU Bra SORT
* Mov #16,R9 ;;Add R11,R9
* MOVEM.L A0/A3,-(SP) push L & J onto sort stack
* STACK St R10,-1[R15] ;;Sub #1,R15
* St R8,-1[R15] ;;Sub #1,R15
* LEA LENGTH(A3),A0 L <- J+1, U stays the same
* BRA SORT continue the sort
* NEWL Bra SORT
* Mov LENGTH,R8 ;;Add R11,R8
*
* fall into insertion sort as all subfiles bellow
* or equal M records - insertion sort phase
*
* Register use: D0 -> loop counter
* D1 -> counter and swap register
* D2 / D4 -> swap registers
* D5 / D7 -> "V" save registers
* A0 -> REC(I)
* A1 -> REC(J)
* A2 / A3 -> work registers
* A4 -> REC(J-1)
* A5 -> "V" save registers
* A6 -> frame pointer
* Note: stack space is reserved for "V" key compare record copies
*
* MOVEM.L (SP)+,D0/A0 reload rec. count & top record
* St R0,0[R15] ;;Add #1,R15
* St R8,0[R15] ;;Add #1,R15
* LINK A6,#-LENGTH allocate "V" key copy area
* SUB #2,D0 D0 ranges from N-2 through 0
MOV #LENGTH,R1 ;;Sub #4,R15
St R14,[R15] ;;Sub #2,R0
Mov R15,R14 ;;Sub R1,R15

LEA -LENGTH(A0),A0 ;;Mov R0,R0
LEA KEY(A0),A2 ;;Mov R1,R8
LEA LENGTH+KEY(A0),A3 ;;Mov #KEYLEN-1,R1
MOVE.L #KEYLEN-1,D1 ;;Mov #KEYLEN-1,R1

LEA KEY(A0),A2 ;;Mov R0,R0
LEA LENGTH+KEY(A0),A3 ;;Mov #KEYLEN-1,R1

LOOP Add #KEY,R8,R11 ;;Mov R8,R0
Mov LENGTH,Rl ;;Sub R1,R8
Add #KEY,R8,R10 ;;Mov #KEYLEN-1,R1

CMP.M.B (A3)+(A2)+ ;;Mov R8,R9
DBNE D1,CMPIII ;;Mov R8,R9
BLS ENDIF

CMPIII Ld [R11>>2],R5 ;;Not R11,R7
Ld [R10>>2],R6 ;;Xc R7,R5
Not R10,R13 ;;Xc R13,R6
Bne R5,R6,CMPIII
Add #1,R11 ;;Add #1,R10
Beq #0,R1,CMPIII
Bl s R6,R5,ENDIF

MOVE.L (A0),D5-D7/A5 ;;Mov R8,R9
LEA LENGTH(A0),A1 ;;Mov R8,R9

Ld 0[R8],R5 ;;Mov R8,R9
Ld 4[R9],R6 ;;Add #4,R9
Ld 4[R9],R7 ;;Add #4,R9
Ld 4[R9],13 ;;Sub #-8,R9

MOVE.L D5-D7,(SP) ;;Mov R8,R9
MOVE.L A0,A4 ;;Mov R8,R9

St R5,0[R15] ;;Mov R8,R12
St R6,4[R16] ;;Add #4,R15
St R7,4[R15] ;;Sub #4,R15

MOVE.L (A1),D1-D4 ;;Mov R8,R12

LOOPIN Ld 0[R9],R1 ;;Mov R0,R0
Ld 4[R9],R2 ;;Add #4,R9
Ld 4[R9],R3 ;;Add #4,R9
Ld 4[R9],R3 ;;Add #-8,R9

MOVE.L D1-D4,(A4) ;;Mov R0,R0
MOVE.L A1,A4 ;;Mov R0,R0

St R1,0[R12] ;;Add #4,R15

- 72 -
St  R2,0[R12] ;;Add  #4,R12
St  R3,0[R12] ;;Add  #4,R12
St  R4,0[R12] ;;Mov  R9,R12

* LEA  LENGTH(A1),A1  J = J+1
* LEA  KEY(A1),A3  A3 -> KEY(J)

* Mov  LENGTH,R11  ;;Add  R9,R11
* Mov  R11,R9  ;;Add  #KEY,R11

* LEA  KEY(SP),A2  A2 -> KEY(V)
* MOVE.L  #KEYLEN-1,D1  loop counter in D1

Add  #KEY,R15,R10  ;;Mov  #KEYLEN-1,R1

* CMPM.B  (A3)+(A2)+ compare KEKY(V) & KEY(J)
* DBNE  D1,CMPVJ  loop while equal
* BHI  LOOPIN if KEY(V) > KEY(J) keep looping

CMPVJ  Ld  [R11>>#2],R2  ;;Not  R11,R3
Ld  [R10>>#2],R4  ;;Xc  R3,R2
Not  R10,R3  ;;Xc  R3,R4
Bne  R2,R4,CMPVJ
Add  #1,R11
Bne  #0,R1,CMPVJ
Sub  #1,R1
Bhi  R4,R2,LOOPIN

* MOVEM.L  D5-D7/A5,(A4)  REC(J-1) <- V

St  R5,0[R4] ;;Mov  R0,R0
St  R6,4[R4] ;;Add  #4,R4
St  R7,4[R4] ;;Add  #4,R4
St  R13,4[R4] ;;Add  #-8,R4

* DBRA  D0,LOOP continue linear insert

ENDIF  Bne  #0,R0,LOOP
Sub  #1,R0  ;;Mov  R0,R0

* UNLK  A6  free and restore stack
* MOVEM.L  (SP)+,(D0-D7/A0-A6) restore registers
* RTS return to caller

Add  #4,R14,R15  ;;Add  #1,R15
Ld  -1[R15],R0  ;;Add  #1,R15
Ld  -1[R15],R1  ;;Add  #1,R15
Ld  -1[R15],R2  ;;Add  #1,R15
Ld  -1[R15],R3  ;;Add  #1,R15
Ld  -1[R15],R4  ;;Add  #1,R15
Ld  -1[R15],R5  ;;Add  #1,R15
Ld  -1[R15],R6  ;;Add  #1,R15
Ld  -1[R15],R7  ;;Add  #1,R15
Ld -1[R15], R8 ;;Add #1, R15
Ld -1[R15], R9 ;;Add #1, R15
Ld -1[R15], R10 ;;Add #1, R15
Ld -1[R15], R11 ;;Add #1, R15
Ld -1[R15], R12 ;;Add #1, R15
Ld -1[R15], R13 ;;Add R12, Hi
Ld 1[R15], R12 ;;Add #1, R15
Ld -1[R15], R14 ;;Add R12, PC
Mov R12, Hi
END
End