Register allocation and optimization techniques in compiler construction

Edward R. Jorgensen
University of Nevada, Las Vegas

Follow this and additional works at: https://digitalscholarship.unlv.edu/rtds

Repository Citation
https://digitalscholarship.unlv.edu/rtds/133

This Thesis is brought to you for free and open access by Digital Scholarship@UNLV. It has been accepted for inclusion in UNLV Retrospective Theses & Dissertations by an authorized administrator of Digital Scholarship@UNLV. For more information, please contact digitalscholarship@unlv.edu.
INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.
Register allocation and optimization techniques in compiler construction

Jorgensen, Edward R., II, M.S.
University of Nevada, Las Vegas, 1991

Copyright ©1991 by Jorgensen, Edward R., II. All rights reserved.
REGISTER ALLOCATION AND
OPTIMIZATION TECHNIQUES
IN COMPILER CONSTRUCTION

by

Edward R. Jorgensen

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

Master of Science
in
Computer Science

Department of Computer Science and Electrical Engineering
University of Nevada, Las Vegas
April, 1991
Abstract

The purpose of this thesis was to investigate the implementation of register allocation and optimization techniques used in the process of compiler construction. The implementation issues were investigated by choosing an architecture and examining various register allocation and optimization techniques. In choosing the techniques to be implemented, only the most promising possibilities were explored for the specific architecture chosen. The goal was to categorize the register allocation and optimization schemes for the architecture.
The thesis of Edward R. Jorgensen II for the degree of Master of Science in Computer Science is approved:

Evangelos Yfantis
Chairperson, Evangelos Yfantis, Ph.D.

Examining Committee Member, Tom Nartker, Ph.D.

Examining Committee Member, John Minor, Ph.D.

Graduate Faculty Representative, Tom Schaffter, Ph.D.

Graduate Dean, Ronald W. Smith, Ph.D.

University of Nevada, Las Vegas
April, 1991
Acknowledgements

I would like to thank my advisor Dr. Evangelos Yfantis for advice on this thesis and for encouragement throughout my undergraduate and graduate years at UNLV.

Additionally, I would like to thank Dr. Tom Nartker for his continuous support. Other members of the faculty to whom I would like to extend this appreciation to include Dr. John Minor and Dr. Tom Schaffter.

Thanks also go to Ron Young for providing the compiler that was used for implementation.

I would like to thank my employer and co-workers Roger Hardwick, Tom Bozarth, Lorita Scanlan, and Tim Kirkpatrick. Without their support, this effort would not have been possible.

And finally, I would like to thank Bob Maichle and Dolly Turner for editing, general support and encouragement.
Table of Contents

Abstract ............................................................................................................................. i
Acknowledgements ........................................................................................................... iii
Table of Contents .............................................................................................................. iv
List of Figures ...................................................................................................................... vii

1 Introduction ................................................................................................ 1
  1.1 Purpose and Goals ................................................................................. 2
  1.2 Compiler Used ........................................................................................ 3
  1.3 Target Machine ..................................................................................... 3

2 General Issues in Register Allocation ....................................................... 4
  2.1 Potential Execution Time Savings ....................................................... 4
  2.2 Register Allocation Partitioning ............................................................ 5
  2.3 Register Allocation Across Procedure Calls ........................................ 6

3 Previous Work ......................................................................................... 7
  3.1 Usage Counts ...................................................................................... 7
  3.2 Directed Acyclic Graph (DAG) ............................................................ 8
  3.3 Register Descriptors .............................................................................. 9
  3.4 Lifetime Analysis ................................................................................. 10
  3.5 Graph Coloring Strategies ................................................................. 10
    3.5.1 Graph Coloring .............................................................................. 11
    3.5.2 Priority-based Coloring ................................................................. 12

4 Register Allocation and Optimization Techniques ................................ 14
  4.1 Single-Use Register Allocation ........................................................... 14
  4.2 Usage Counts ................................................................................... 15
  4.3 Graph Coloring .......................................................................... 16
  4.4 Register Descriptors ........................................................................... 17
    4.4.1 Algorithm ....................................................................................... 17
    4.4.2 Register Allocation Across Procedure Calls ................................. 18
    4.4.3 Results ............................................................................................ 18
  4.5 Directed Acyclic Graph (DAG) ......................................................... 19
    4.5.1 Example .......................................................................................... 21
    4.5.2 Results ............................................................................................ 23
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.6</td>
<td>Reaching Definitions .................................</td>
<td>25</td>
</tr>
<tr>
<td>4.6.1</td>
<td>Data Flow Analysis ...................................</td>
<td>26</td>
</tr>
<tr>
<td>4.6.2</td>
<td>Computing the gen() and kill() functions ..........</td>
<td>28</td>
</tr>
<tr>
<td>4.6.3</td>
<td>Computing the out() function ....................</td>
<td>28</td>
</tr>
<tr>
<td>4.6.4</td>
<td>Set Representation ...................................</td>
<td>29</td>
</tr>
<tr>
<td>4.6.5</td>
<td>Results ...............................................</td>
<td>30</td>
</tr>
<tr>
<td>4.7</td>
<td>Live-Variable Analysis ...............................</td>
<td>32</td>
</tr>
<tr>
<td>4.7.1</td>
<td>Data Flow Analysis ...................................</td>
<td>33</td>
</tr>
<tr>
<td>4.7.2</td>
<td>Computing the in() function .......................</td>
<td>34</td>
</tr>
<tr>
<td>4.7.3</td>
<td>Results ...............................................</td>
<td>35</td>
</tr>
<tr>
<td>4.8</td>
<td>Variable-Use Analysis ................................</td>
<td>36</td>
</tr>
<tr>
<td>4.8.1</td>
<td>Results ...............................................</td>
<td>36</td>
</tr>
<tr>
<td>4.9</td>
<td>Peephole Optimization ................................</td>
<td>38</td>
</tr>
<tr>
<td>4.9.1</td>
<td>Optimizations .........................................</td>
<td>39</td>
</tr>
<tr>
<td>4.9.2</td>
<td>Results ...............................................</td>
<td>39</td>
</tr>
<tr>
<td>4.10</td>
<td>Other Optimizations ..................................</td>
<td>42</td>
</tr>
<tr>
<td>5</td>
<td>Conclusion ............................................</td>
<td>43</td>
</tr>
</tbody>
</table>

Bibliography .......................................................... 44

Appendix A, Intel 80x86 Architecture Overview .................... 46

Appendix B, Single-Use Register Allocation .......................... 49
B.1 Examples ....................................................... 50
B.2 Source Code .................................................... 58
B.2.1 Source Code, Single-Use Register Allocation .......... 58

Appendix C, Register Descriptors ..................................... 78
C.1 Examples ......................................................... 79
C.2 Source Code ..................................................... 105
C.2.1 Source Code, Register Descriptors ..................... 105
C.2.2 Source Code, Optimize Function ......................... 116

Appendix D, Directed Acyclic Graph ................................. 124
D.1 Examples ......................................................... 125
D.2 Source Code ..................................................... 144
D.2.1 Source Code, Optimize Function ......................... 144
D.2.2 Source Code, DAG Routines .............................. 146

Appendix E, Reaching Definitions ...................................... 154
E.1 Examples ........................................................ 155
E.2 Source Code ..................................................... 163
E.2.1 Source Code, Optimize Function ......................... 163
E.2.2 Source Code, Reaching Definitions ...................... 164
E.2.3 Source Code, Store Registers ............................ 175
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1</td>
<td>A Compiler</td>
<td>1</td>
</tr>
<tr>
<td>Figure 2</td>
<td>Compiler Phases</td>
<td>2</td>
</tr>
<tr>
<td>Figure 3</td>
<td>Directed Acyclic Graph</td>
<td>8</td>
</tr>
<tr>
<td>Figure 4</td>
<td>DAG Algorithm</td>
<td>21</td>
</tr>
<tr>
<td>Figure 5</td>
<td>DAG Example</td>
<td>22</td>
</tr>
<tr>
<td>Figure 6</td>
<td>Predecessor Flow-Graph</td>
<td>27</td>
</tr>
<tr>
<td>Figure 7</td>
<td>Predecessor Flow-Graph Algorithm</td>
<td>27</td>
</tr>
<tr>
<td>Figure 8</td>
<td>Algorithm to Calculate Reaching Definitions</td>
<td>29</td>
</tr>
<tr>
<td>Figure 9</td>
<td>Successor Flow-Graph</td>
<td>33</td>
</tr>
<tr>
<td>Figure 10</td>
<td>Successor Flow-Graph Algorithm</td>
<td>34</td>
</tr>
<tr>
<td>Figure 11</td>
<td>Algorithm to Perform Live-Variable Analysis</td>
<td>35</td>
</tr>
<tr>
<td>Figure 12</td>
<td>Final Code Generation and Register Allocation Strategy</td>
<td>43</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

A compiler is a translator between a source language and a target language (refer to Figure 1). The source language is typically written by a programmer and the target language is acted on by a machine.

![Figure 1, A Compiler](image)

Compilation involves several steps including reading the input program, recognizing constructs of the programming language, choosing an implementation of those constructs from the available instructions of the target machine, preparing an executable image of the operations. Compilers are typically organized in phases [1], each of which performs one part of the translation. The early phases of compilation which include lexical analysis, syntactic analysis, and semantic analysis are language specific and machine independent. That is, these phases incorporate knowledge of the source language, but do not refer to specific features of the target machine. The early phases are referred to as the front-end. The later phases of compilation which include resource allocation, instruction selection, and code generation are machine specific but language independent. That is, these phases incorporate specific knowledge about the target machine, but do not refer to features of the source language. The later phases are referred to as the back-end of a compiler.

The front-end compiler design issues rely on formal language theory. There are various well established models for compiler front-end design [2]. These include top-down (LL) and bottom-up (LALR). Compiler front-end phases include lexical analysis, syntactical analysis, and semantic analysis. Additionally, table driven tools such as yacc and lex are available to perform the front-end parsing and lexical analysis.
Back-end related research has yet to produce a widely accepted design model for overall back-end construction. Due to the diverse number of architectures and recent advances in new architecture designs, a significant amount of work has been performed in this area. Between the diverse architectures and lack of a widely accepted back-end model, much of the work in this area has been ad-hoc routines developed for very specific cases. The following diagram shows the general front-end, back-end relationship.

**Figure 2, Compiler Phases**

One of the back-end issues that is encompassed by the code optimization and code generation is register allocation.

### 1.1 Purpose and Goals

This thesis examines various register allocation and related optimization techniques and further examines the related implementation issues for back-end code generation. This included choosing a specific architecture to implement and investigate various register allocation techniques. Only the most promising possibilities were explored for the specific architecture chosen. It is expected that for different architectures an entirely different set of results would be generated.

The goal was to categorize the register allocation and optimization techniques for a specific architecture.
1.2 Compiler Used

The compiler used for implementation is a compiler for a locally defined language named UNLV2-Version 2 [3]. The UNLV2-Version 2 language is an enhancement of the UNLV2-Version 1 compiler [4]. The UNLV2-Version 2 compiler was written by Ron Young another graduate student at the University of Nevada, Las Vegas. The compiler consists of a basic front-end and back-end that produces assembly language code for the Intel 80x86 architecture. The produced code can then be assembled by a standard PC assembler. The run-time support routines for basic input and output were also included with the compiler.

The UNLV2-Version 2 language is a straight-forward PASCAL-like, structured language. Language constructs include WHILE loops, IF statements, IF...THEN...ELSE statements, and procedure/function calls. Procedures and functions can have parameters and local variables whose storage is dynamically allocated and can call themselves recursively. The issues of variable scoping must be adequately handled throughout the front- and back-end of the compiler. Data types include integers, real's, characters, arrays, records, and pointers. All Pascal features are included except for the set features.

The reasons this compiler was used include the source code availability and the fact that the intermediate code generated is quadtuples. Quadtuples were chosen as a representative type of intermediate code for general application.

1.3 Target Machine

The Intel 80x86 architecture [5][6] was chosen due to the wide-spread availability of such processors. Due to the limited number of registers available and the restrictive instruction set, an efficient and effective register allocation mechanism is particularly important.

An overview of the Intel 80x86 architecture, including the machine registers, is located in Appendix A. A basic knowledge of the architecture is required to understand the comments and descriptions presented in the remainder of this document.

The processor specific extensions are not addressed by the compiler. The processor specific extensions refer to the architecture enhancements for alternate memory addressing mechanisms (i.e., virtual memory). One reason for the processor extensions not being addressed by the compiler is that the 80286 extensions are not compatible with the 80386 extensions [7]. The primary reason the processor specific extensions are not used is that the 80286 and 80386 processor extensions are not being used by the operating system (i.e., DOS). As a result of the operating system not using the processor specific extensions, code written using the extensions is unable to call operating system functions to perform system interaction (i.e., file or terminal input/output).
Chapter 2

General Issues in Register Allocation

Register allocation refers to the process of allocating or assigning registers to program variables. This section presents an overview of the general issues involved in register allocation. These issues include a description of the advantages and disadvantages of performing register allocation.

The general process of always performing an optimal selection of program variables to assign to registers and when to assign them is NP-complete.

2.1 Potential Execution Time Savings

The general idea behind register allocation is that machine instructions which operate on values in registers are typically executed faster than values in memory. Depending on the instruction, this difference or speed-up can be significant. Due to the nature of the Intel architecture instruction set, one operand must be in a register. The second operand can be either in memory or in a register. There is a difference in the amount of time required to execute an instruction with both operands in registers as compared to only one operand being in a register.

For example, a very common instruction on the Intel 80286 processor like the ADD instruction with a register operand (REG) and a memory operand (MEM):

ADD REG, MEM

would be take 16 machine cycles plus the effective address calculation time¹ which is between 6 and 12 additional machine cycles [5][6]. Whereas, the ADD instruction with two register operands (REG, REG):

ADD REG, REG

would be take 3 machine cycles. The difference or minimum speed-up of 19 machine cycles.

¹ The effective address calculation time is the amount of time required to calculate the offset. This might include a base address plus an offset for indexed or additional displacement memory addressing. The time required to compute the physical memory addresses from the segment address and offset is included in the instruction time. Refer to Appendix A for an overview of the segmentation scheme used by the Intel 80x86 architecture.
cycles would be multiplied by the number of uses. The difference can accumulate over the course of a large program to be quite significant.

However, the difference or potential speed-up must be offset by the amount of machine cycles required to move the operand from memory-to-register. If the value was altered, the additional amount of machine cycles required to move the operand from the register back to memory must also be considered.

Since one operand must be moved from memory into a register, the time for that memory-to-register operation is required. It is then an option to either move the second operand into a register or leave it in memory. The final decision will be affected by many factors. One of which might be the amount of time required to perform the memory-to-register and potential register-to-memory operations. For a single operation, using two registers would increase the overall execution speed.

For example, assuming a register is used the value must be put into a register with the MOV instruction:

\[ \text{MOV REG, MEM} \]

which takes 9 machine cycles plus the effective address calculation time which is between 6 and 12 additional machine cycles. If the value in the register is altered, at some point it must be saved back into memory:

\[ \text{MOV MEM, REG} \]

which takes the same number of machine cycles as the previous example. The minimum 15 machine cycles for the load, plus the potential of a minimum additional 15 machine cycles clearly shows that for a single instance, using a register would increase execution time.

In order to fully realize the potential benefits of the additional speed-up due to register usage, a series of instructions would need to be performed with the value in a register.

2.2 Register Allocation Partitioning

Register allocation can be performed locally or globally. Local register allocation refers to assigning registers within either a set of basic blocks or within a program procedure. Global register allocation refers to assigning or using registers across multiple basic blocks or procedures. A subset of registers may be reserved for local assignment and the remaining registers may be assigned to global assignment.

Additionally, some registers are reserved for the compiler back-end to use for bookkeeping purposes. Depending on the architecture, these might include such registers as the stack pointer, base pointer, frame pointer, and/or instruction pointer. In general these registers, defined by the hardware, are dedicated to their respective function. For
the Intel 80x86 architecture, these dedicated registers include the segment, stack, base pointer, and instruction pointer registers.

In general, the registers are divided into two fixed partitions. One for the dedicated registers and the other for the non-dedicated or general purpose registers. This document will address register allocation for the partition of general purpose registers.

2.3 Register Allocation Across Procedure Calls

One major issue in register allocation is how to handle allocated registers in the context of procedure calls. There are three major alternatives. The first is caller-save. Here the register save code must be generated to save all allocated registers before the procedure is called, and then to restore the registers after the procedure returns. The second alternative, callee-save, is to save the registers at the entry point of the procedure. In this manner, the procedure need only save the registers it can potentially modify. The last alternative is to, at run-time, compute dynamically which registers to save. As part of the procedure linkage, the set of registers to save is computed by taking the intersection of the set of live registers in the calling routine and the set of registers required by the procedure.

None of these alternatives is optimal in all cases. Saving registers prior to the call is optimal if the called procedure uses more registers than the caller routine, since fewer registers would need to be saved at the call-site than would need to be saved at the entry point. Conversely, saving registers at the entry point is optimal if the calling routine uses more registers than the called routine, since fewer registers would need to be saved at the entry point than at the procedure call.

While dynamic linkage appears to save the minimum number of registers in all cases, the scheme can quickly degrade into an expensive form of callee-save. The higher expense arises from the computation of the set of registers that must be saved which is performed at run-time for each procedure call [8].

One study has shown there to be very little difference between caller-save and callee-save [8]. This study was conducted by recompiling a large number of programs (all UNIX utilities) using both methods and comparing the execution times. Of all the programs, over 80% showed no change. The remaining programs were split between executing slightly faster and executing slightly slower. Overall, the differences were small and primarily a function of the style in which the programs were written.

The traditional approach is to use callee-save for register allocation across procedure calls.
Chapter 3

Previous Work

There has been a great deal of work in the area of register allocation. This chapter presents an overview of previous work in this area. For the purposes of presenting this information, the previous work has been categorized into the following general register allocation strategies:

- Usage Counts
- Directed Acyclic Graph (DAG)
- Register Descriptors
- Lifetime Analysis
- Graph Coloring

These general strategies and the related previous works are presented in the following sections.

3.1 Usage Counts

One of the earliest methods of performing register allocation involved the calculation of program variable "usage counts" [9]. Machine registers are assigned to program variables with the highest usage during code generation. Usage counts are used primarily for local register allocation.

A data structure is created which contains a record of the variables that are currently assigned to each register and a status flag for each register. Additionally, a record of each variable and its usage count is also maintained. Then, as code generation occurs, each time an instruction is generated that uses a variable, the associated usage count is decremented. When the usage count drops to zero, the register is released for reuse. When there is an excess demand for registers, the register containing the lowest usage count is selected for possible re-loading. The previous value must be stored only if it has been changed as indicated by the status flag. This approach is generally applied either to basic blocks or procedures.

One of the primary advantages is the relatively low overhead associated with this approach. This is true even when there is a relatively low number of registers available on the target machine.

Another of the stated advantages is that this approach performs near optimal register allocation for linear code regions. However, this is not true for all architectures.
Unfortunately, one of the disadvantages is that locally optimal solutions to the problem of register allocation do not necessarily add up to the globally optimal solution. Specifically, register assignment across non-linear regions is not optimal. An optimal solution would depend on the frequency of execution of each flow path through the basic block. Assumptions are then made regarding the frequency of execution of each flow path and the associated program variables. For example, variables in a loop can be weighted more heavily.

3.2 Directed Acyclic Graph (DAG)

The use of a Directed Acyclic Graph (DAG) was suggested by Aho, Sethi, and Ullman [1]. The DAG is used to assign registers to temporaries during expression evaluation. Interior nodes of the DAG represent the generation or modification of temporaries created along the various paths of the DAG. Registers can be assigned to these interior nodes, thus improving the time required for overall evaluation of the expression.

For example, given the expression:

\[ x = a / b + (c + d) * e \]

the DAG generated would be as follows:

![Directed Acyclic Graph](image)

*Figure 3, Directed Acyclic Graph*

Registers would be assigned to the \( t_0 \), \( t_1 \), and \( t_2 \) nodes. The advantage of this approach is in identifying which node to assign registers to when there is not enough registers for all nodes, as might be the case with larger expressions. If not enough registers are available for all nodes, registers are assigned only to the longest paths in the DAG.
One potential problem with this approach is how embedded subroutine or functions calls are handled. If registers are used as temporaries, then they have to be saved before calling the function and restored afterwards. Additional effort is required to recognize the function calls, and decide how to handle such cases. Registers might be assigned to temporaries only after the function call or might be passed as part of the function call.

3.3 Register Descriptors

The basic idea of using register descriptors and address descriptors to keep track of register contents and variable addresses during code-generation is outlined by Aho, Sethi, and Ullman [1] and is the combination of several strategies [10][11][12].

The strategies are combined into a single algorithm, referred to as getreg(), that performs register assignment during code-generation using the register and address descriptors. The register descriptors keep track of what is in each register, and is consulted when a new register is required. The address descriptors keep track of the location of the most recent value of the variable and the assigned memory location. At various points during code-generation, the current location might be a register or memory address.

As code-generation progresses and variables are accessed or updated, the function getreg() is used to assign registers to variables when possible. The function uses the register descriptors and, when required, may move a variable from a register back to memory. The function is invoked for every tuple that accesses program variables. The basic version of getreg(), as outlined in the literature, is used to perform local register allocation.

The getreg() function takes a standard quadtuple (i.e., \((x = y \text{ op } z)\)) as input and returns a location, \(L\), where the results of the quadtuples should be stored. The location is typically a register.

The basic algorithm for getreg is as follows:

\[
L = \text{getreg} \quad \text{(quadtuple } x = y \text{ op } z) \quad \)

1) If the name \(y\) is in a register that holds the value of no other names, and \(y\) is not live and has no next use after execution of \(x := y \text{ op } z\), then return the register of \(y\) for \(L\). Update the address descriptor of \(y\) to indicate that \(y\) is no longer in \(L\).

2) Failing (1), return an empty register for \(L\) if there is one.

3) Failing (2), if:

   1) \(x\) has a next use in the block.
   2) \(\text{op}\) requires a register.

   Find an occupied register \(R\). Store the value in \(R\) into memory location (via a MOV instruction) if it is not already in the proper memory location...
M, update the address descriptor for M, and return R. If R holds the value of several variables, a MOV instruction must be generated for each variable that needs to be stored. A suitable occupied register might be one whose datum is referenced furthest in the future, or one whose value is also in memory.

4) If x is not used in the block, or no suitable occupied register can be found, select the memory location of x as L.

The architecture specific implementation and extensions to the basic getreg() algorithm are described in Chapter 4, Section 4.4.

3.4 Lifetime Analysis

In order perform register allocation across basic blocks, information about the variables used in the next basic block or blocks is required. Generally, for local register allocation, the values in registers must be written to memory at the end of each basic block. This is required since control may reach the block from multiple other blocks, and it cannot be directly assumed that a variable used by the successor block will always appear in the same register. In some cases this may result in performing a register-to-memory operation and then in a successive block, performing a corresponding memory-to-register operation on the same variable. Additionally, since it spans a basic block a peep-hole optimization will be unable to correct the problem. In fact, depending on the specific data-flow, the additional memory-to-register operation may be required (i.e., a loop situation).

It might be possible to retain a value or values in a register across multiple basic blocks, by obtaining information about the variables used in the successor block or blocks. This would allow a more global solution to register allocation issues.

To obtain such information, a data-flow analysis must be performed in order to generate the data-flow information. The data-flow information consists of the in() (i.e., variables that are live going into the basic block) and/or the out() (i.e., variables still live going out of the basic block).

There are several methods of computing the in()’s and out()’s of basic blocks. Two of those methods are investigated in Chapter 4, Sections 4.6 and 4.7.

3.5 Graph Coloring Strategies

The application of graph coloring to the problem of register allocation is outlined by Chaitin, et al. [13] and later refined by Chaitin [14]. There have been a great deal of variations on the graph coloring strategy [15][16][17]. For register allocation, each node in a coloring graph represents a program variable or quantity that is a candidate for residing in a machine register. The number of colors used for coloring the graph is the
number of registers available for use in register allocation. The goal is to find the best way to assign program variables to registers so the number of variables in registers is maximized.

Coloring a graph is an assignment of a color to each graph node in such a manner that if two nodes are adjacent (i.e., connected by an edge of the graph), then they must have a different color assigned. A coloring is said to be an \( n \)-coloring if it does not use more than \( n \) colors. The chromatic number of a graph is defined to be the minimal number of colors for which there is an \( n \)-coloring of the graph. The basic problem of determining whether a given graph is \( n \)-colorable is NP-complete. This suggests that in some cases an impractical amount of computation time might be required to determine the coloring. In some cases, the amount of time could be an exponential function of the size of the graph. Any algorithms that use the graph coloring strategy must take some steps to overcome this potentially significant obstacle.

This section presents an overview of the basic graph coloring algorithm and one of the graph coloring variations.

3.5.1 Graph Coloring

The basic graph coloring algorithm outlined by Chaitin, et. al. involves creating a register interference graph for each procedure in the source program. Two computations which reside in machine registers are said to interfere with each other if they are live simultaneously at any point in the program. The resulting graph is potentially very large, and a must be stored in a data structure that takes as little space as possible. For an \( N \)-node graph, an \( N \times N \) bit matrix can be constructed to represent the graph.

A series of operations and optimizations are performed on the graph after it is built. The resulting graph is then used to perform register assignment. One of the basic assumptions of graph coloring is that no previous register allocation has been performed, and that all values have memory locations assigned. This is required in case a register does not get assigned to the variable or should spill code be required. Spilling or spill code implies that the values are kept in a memory location as opposed to a register.

After the interference graph is built, the next step is to coalesce the nodes for the purpose of assuring that separate nodes in the graph get the same color. This is done by taking two nodes that do not interfere and combining them in a single node which interferes with any node which either of them interfered with before.

Additionally, before the interference graphs for program procedures are colored, the graph is reduced. This reduction, referred to as subsumption, is performed by eliminating nodes in the graph that have fewer edges than there are colors available. That is, if there are \( n \) colors available, and the graph has fewer than \( n \) nodes, the graph is colorable no matter how the colors are assigned. As such, the node can be eliminated from the graph. This is a very powerful reduction, which can in some cases completely eliminate the interference graph. If the graph is reduced to the empty set, colors can be assigned sequentially to the nodes in the order that they were removed.
The final step is to perform the graph coloring on the processed interference graph (if it is non-empty). This step can be potentially very time consuming, and would only be performed if a high degree of optimization is required.

If the graph is not colorable, spill code must be introduced. As spill code is introduced, the interference graph is modified accordingly. As the interference graph is updated, the new graph may be colorable. This process is continued until a complete coloring can be obtained.

Due to the overhead required and the nature of the coloring algorithm, this approach works best for a target machine with a large number of free registers.

3.5.2 Priority-based Coloring

One of the variations of the standard coloring algorithm involves assigning priorities in node coloring. This assignment of priorities is based on estimates of the benefits that can be derived from allocating individual quantities in registers. Since the costs involved in register allocation are taken into account, the algorithm will not over-allocate and will execute in linear time.

This approach to graph coloring still assumes that all values have memory locations assigned. This is required in case a register does not get assigned to the variable or should spill code be required.

Standard graph coloring does not take into account the cost and saving involved in allocating variables to registers. The cost refers to the required instructions that perform the register-to-memory and memory-to-register operations that put variables in registers or make the registers available for other uses. The savings refers to the amount of time gained in the execution of the program. Variables occur with different frequencies and with varying degrees of clustering, so that the relative benefits of assigning registers to variables differ. The standard coloring algorithm always tries to allocate as many items in registers as possible and does not recognize the fact that this is sometimes not beneficial due to the register load/store costs. Also, the standard coloring algorithm does not take into account the loop structure of the program. In practice, variables occurring in frequently executed regions should be given greater preference for residing in registers.

In order to overcome these problems, cost and saving estimates are generated for each program variable. These estimates include the cost of moving variables from memory into registers and saving the register values back to memory. The following parameters are used:

\[
\text{MOV\text{COST}} \quad \text{The cost of a memory-to-register or register-to-memory operation, which in practice is the execution time of the move instruction of the target machine.}
\]
LODSAVE The amount of execution time saved for each reference of a variable residing in a register compared with the corresponding memory reference that is replaced.

STRSAVE The amount of execution time saved for each definition of a variable residing in a register compared with the corresponding store to memory being replaced.

Thus, for each variable in the basic block, the potential savings of having the variable in a register can be estimated. However, if the current basic block is considered together with the preceding and subsequent blocks, the actual saving may increase. This can happen if preceding or subsequent blocks use the same variable, and that variable is assigned to the same machine register. This might potentially eliminate a series of memory-to-register or register-to-memory operations, which would have an impact of the savings estimates. To deal with this possibility, the following two separate estimates are tracked:

\[
\text{MAXSAVE} = \text{LODSAVE} \times u + \text{STRSAVE} \times d
\]

\[
\text{MINSAVE} = \text{LODSAVE} \times u + \text{STRSAVE} \times d - \text{MOVcost} \times n
\]

Where: 
- \( u \) is the number of uses of the variable,
- \( d \) is the number of definitions, and
- \( n \) is either 0, 1, or 2.

The \( n \) depends on whether a load is required at the beginning and/or end of the basic block. If the variable is loaded into a register at the beginning of a block and saved back to memory at the end of the block \( n \) would be 2. If the variable is not changed and does not need to be moved back to memory, \( n \) would be 1. If the variable is already in a register and is not altered, \( n \) would be 0. The variable might already be in a register from a previous basic block.

After the estimates are generated, they are weighted by the loop-nesting depths of the program. In this manner, variables in loops, particularly inner loops, are given a higher priority for register allocation.

Then successive iterations of the coloring algorithm are performed. Each iteration assigns one live range to a register by choosing the most promising live range according to the cost and saving estimates computed over that live range. By assigning the live ranges with the highest priority first, registers are allocated where they will have the most impact on the overall execution time of the program. The algorithm terminates when either all live ranges have been allocated, or all registers have been assigned over all basic blocks. By using the priority based register assignment and potentially terminating before a complete coloring is obtained, the computation time does not deteriorate when an a complete coloring cannot be achieved.
Chapter 4

Register Allocation and Optimization Techniques

This section outlines the register allocation and optimization techniques that were investigated for the Intel 80x86 architecture. The general concepts for existing register allocation and optimization techniques are described in the literature and summarized in Chapter 3. A detailed description of how each general concept was specifically applied is included in this section. A copy of the code and any additional specific information related to each method is located in a respective appendix.

Not all register allocation techniques were evaluated through direct implementation. Two methods, usage counts and graph coloring were investigated and not implemented. The results of those evaluations are also outlined in this chapter.

4.1 Single-Use Register Allocation

The most basic and simplest form of allocating registers during code generation is single-use register allocation. Single-use register allocation means that the value or values are placed in registers as required for the instruction and saved back to memory after the instruction. No attempt to retain values in registers across multiple instructions or track register contents is made.

The back-end, using single-use register allocation would convert the expression:

\[ a = b + c \]

into the following code:

```assembly
mov ax, word ptr ss:[bp-4]
add ax, word ptr ss:[bp-6]
mov word ptr ss:[bp-2], ax
```

The advantage of this approach is that there is very little compile-time overhead. Also, the construction of the back-end becomes much simpler and correspondingly smaller. The primary disadvantage is that the code generated is very inefficient. A large number of needless and possibly redundant memory-to-register and register-to-memory operations are performed.
For example, after the preceding code has been generated, the value of a is in the AX register. If any succeeding statements used the same variable, a would be reloaded from memory. If the succeeding statement that used the same variable immediately followed, it would be possible to perform peep-hole optimization to remove the redundant load/store.

This approach is most commonly used when the compiler is not being used to produce high quality code. Several books for C-Language programming and compiler construction, particularly ones that focus on front-end issues, favor this approach for its simplicity [18].

The purpose of implementing this approach was to establish a baseline for comparison with the other register allocation schemes. This also helped to establish the back-end model.

Refer to Appendix B for detailed examples and a copy of the source code.

4.2 Usage Counts

Register allocation by usage counts involves the calculation of program variable frequency counts. As code is generated, machine registers are assigned to the program variables with the highest usage. In general, usage counts are used primarily for local register allocation. During this process, a register descriptor must be maintained to track the variables that are assigned to registers. When applying the usage counts register allocation technique to the Intel 80x86 architecture, some architecture specific issues must be addressed.

It would be possible, and with relatively low overhead, to generate the program variable usage counts. This information would then be used to assign registers to the variables with the highest use counts.

This method alone would be inadequate due to the mandatory register use requirements of the instruction set. For example, given the quadtuple:

\[ a = b + c \]

The instruction set requires that the operand b be assigned to a register, and that the result, a, would then reside in the same register. It would be possible for the variables a or b to have a relatively low usage in the basic block. If that were the case, and the variable did not get assigned to a register in the correct order, the program would not work.

The variable b must be moved into a register prior to the instruction regardless of its usage count. If all registers are being used, another register must be made available by the generation of spill code. Since the variable has a low usage, it might then either needlessly use a register or require more spill code to be generated to be able to re-use the register.
Additionally, it might be possible for the variable \( c \) in the quadtuple above to be used a large number of times without being redefined. If the instances of use are in close proximity, register assignment would be appropriate. If the instances of use are infrequent or more widely distributed, register assignment may not be appropriate depending on the number of free registers available.

The generation of spill code to free registers for quadruples with mandatory register use requirements would make the direct application of usage counts tend to generate poor code when there are more variables than available registers. This would be true even for linear code segments, where usage counts normally generate very high-quality code.

While the direct application of usage counts would be inappropriate for the Intel 80x86 architecture, it might be possible to combine this approach with another mechanism.

4.3 Graph Coloring

The basic graph coloring algorithm involves creating a register interference graph for each procedure in the source program. Interference implies that two variables are live simultaneously at some in the program. A series of optimizations is performed after the interference graph is built to reduce the graph so that multiple nodes that do not interfere are coalesced into a single node. The resulting graph is then colored in order to assign different registers to the interfering variables.

If a coloring cannot be obtained, spill code must be introduced. The interference graph is updated as the spill code is generated to see if the resulting graph is colorable. This process is continued until a coloring can be obtained. With a small number of registers available, it is very unlikely that for non-trivial programs a coloring will be obtained without the generation of spill code.

This approach does not take into consideration the mandatory register use requirements of the instruction set. Variables there are spilled may be required in a register due to the instruction set. If that is the case, additional code must be generated to free the required register, load the variable into the register, and potentially save the result back to memory. The end result will be an excess of potentially conflicting spill code generation.

Another problem with graph coloring is that, due to the overhead required and the nature of the coloring algorithm, this approach tends to work best for a target machine with a large number of free registers. The more free registers, the better the coloring algorithm will be able to assign a maximum of variables to registers. For the Intel architecture, there are six registers available for allocation to variables. However, the number of registers actually available for allocation is dependant upon the operation or series of operations being performed. For example, the word multiply instruction uses a minimum of two registers and possibly three registers. Because of this, the number of available registers does not remain static, further eroding the potential effectiveness of the coloring algorithm.
Therefore, the coloring algorithm was determined to be non-applicable to the Intel architecture, and was therefore not implemented as part of this effort.

4.4 Register Descriptors

The general algorithm for using register descriptors and address descriptors for register allocation and code generation is embodied in the getreg() algorithm as presented by Aho, Sethi, and Ullman [1]. This section presents information about the specific implementation of the getreg() function for the Intel 80x86 architecture.

The function getreg() is used during code-generation to assign registers to variables when possible. The dynamic nature of the register assignment makes this approach flexible for architecture specific modifications. As such, instructions that require a specific register will be assigned those registers as part of the modified getreg() function. A series of support routines are used to perform various memory store and register free operations. The modified algorithm and the associated support routines are described in the following section.

4.4.1 Algorithm

The modified getreg() function, performs the register assignment in a architecture specific, instruction specific context. To do this, the standard quadtuple \((x = y \text{ op } z)\) along with a flag to indicate which register must be assigned. This might be "any_register" or a specific register depending on the operation. The modified algorithm for the Intel architecture is as follows:

\[
L = \text{getreg} ( \text{quadtuple} (x = y \text{ op } z), \text{which_register} )
\]

1) If the variable \(y\) is in a register that holds the value of no other names, and \(y\) is not live and has no next use after execution of the quadtuple, then return the register of \(y\) for \(L\). Update the address descriptor of \(y\) to indicate that \(y\) is no longer in \(L\).

2) Failing (1), return an empty register for \(L\) if there is one. Search the registers available for free registers. The search must be performed based on register required and the variable type since real, characters, and integers use different registers.

3) Failing (2), if:
   1) \(x\) has a next use in the block.
   2) The operand requires a register.

Free a register (either the register required or one with the least number of entities currently assigned). If any register, search occupied registers for the specific variable type. Generate the required instructions to store the selected register to memory. This implies that multiple instructions might be generated if the register has multiple variables assigned.
In addition to the `getreg()` function, a number of support routines are required. These routines are used to perform the functions of manipulating the register descriptors, freeing registers as required, and generating the memory-to-register/register-to-memory instructions.

Additionally, the next-use information must be generated. The next-use information consists of a simple live-variable analysis for the basic block. Performing the live-variable analysis through a single block without the potential of multiple control-flows is very straightforward. These routines are used exclusively by the back-end. Appendix C presents a copy of the source code for the `getreg()` function for reference.

4.4.2 Register Allocation Across Procedure Calls

The `getreg()` function performs caller-save register allocation for procedure calls. This is handled by default since a procedure call would be placed into a separate basic block. Since the `getreg()` function will write to memory, and therefore free, all registers at the end of a basic block, no additional effort is required at the start of a procedure. All registers are assumed to be available.

It would be possible to change this, and leave values in the registers across a procedure call. If the procedure required a register, the register descriptors would indicate the prior use. If all registers are used, the appropriate register-to-memory instructions could be generated for the register required. Such a scheme would require additional compile-time.

Previous studies have established [8] that the difference between caller-save and callee-save do not justify the additional effort.

4.4.3 Results

The use of the `getreg()` function produced acceptable results for local register allocation. The results were very good when the program had a small number of variables. The results were acceptable for programs with a larger number of variables. Specific registers must be used for certain machine instructions due to the mandatory register use requirements of the architecture.

Un-necessary register-to-memory and memory-to-register operations were generated in some situations. A certain amount of register swapping is unavoidable with the small number of registers available and the required register use for the instruction set.

Refer to Appendix C for detailed examples and a copy of the source code.
4.5 Directed Acyclic Graph (DAG)

Directed Acyclic Graphs (DAGs) are widely used for common subexpression elimination within basic blocks and performing transformations on basic blocks. The basic approach to building and applying a DAG is presented by Aho, Sethi, and Ullman [1]. The use of a DAG for register allocation is also mentioned by Holub [19] for register assignment during expression evaluation. This DAG approach, developed locally [20], primarily follows Aho, Sethi, and Ullman approach.

The DAG has been applied for the purpose of improving register allocation mechanisms in the context of the Intel architecture. A complete DAG is created for the quadtuples in a basic block prior to instruction generation. This DAG is then used to heuristically reorder the quadtuples. Code is then generated based upon the reordered quadtuples. The nature of the heuristic reordering tends to coalesce the quadtuples that share common variables. This is directly applicable to register allocation since a variable is more likely to be put in a register, and allowed to stay in a register, when the statements that use the variable/register are in close proximity. This is especially true for the Intel 80x86 architecture, due to the mandatory register use requirements for most instructions. The proximity of quadtuples or instructions that use the same variables can significantly improve overall register allocation and code generation efficiency where there is a high contention for available registers.

The general processes of building, accessing, and numbering a DAG is outlined in the literature. The classic approach to using a DAG includes building the DAG and subsequently numbering the DAG nodes in a very specific manner. This numbering is performed starting with a root node, traversing downward, left child first, sequentially numbering each node. A node is only numbered if all the parents have been previously numbered. If one or more of the parents is as yet un-numbered, the downward traversal for that branch stops. This process continues until all the nodes are numbered. This numbering, in reverse, is used to re-order the nodes. The new ordering for the nodes is more optimal since the algorithm, when possible, tends to coalesce the nodes that share common variables. This would allow a more efficient register assignment and code generation for the Intel architecture.

The classic approach to creating and accessing a DAG typically includes a parent field, where the parent field is used to refer to a linked list of parent nodes. This linked list of parents is used to verify the parent numbering status during DAG node numbering process. Each parent is checked, via the linked list, for its node numbering status. When an un-numbered parent is found, node traversal down that branch of the DAG is halted.

The DAG node description typically includes the operand. In practical terms, the operand is already stored in the symbol table, and as such the structure need only contain a pointer to the appropriate location in the symbol table.

---

2 This assumes some type of dynamic register allocation mechanism (such as the getreg() function previously described).
For this application of a DAG, the basic DAG build and all DAG accesses can be performed without using a linked list of parents. The structure contains a use_count instead of a list of parent pointers. The use count is a simple integer counter that is used to indicate the number of parents. The count is very easily set or incremented during the DAG build. This counter is used instead of a linked list of parents. The algorithm is more efficient and easier to implement without the additional complexity of a linked list of parent nodes.

A node might be defined as follows:

```c
struct typical_node {
    struct symtab *operand;
    int operator;
    struct typical_node *left;
    struct typical_node *right;
    int node_order;
    int use_count;
};
```

A linked list of root nodes is maintained as part of the node building. This is required for this application since a basic block can have multiple independent sets of quadtuples. The list of root nodes is used to track the multiple DAGs that result from the multiple independent quadtuples. Refer to Section 4.5.1 for an example.

Only the non-leaf nodes are numbered after the DAG is built for this application. The non-leaf nodes represent the result of a quadtuple and leaf nodes represent quadtuple operands for register allocation purposes. A non-leaf node can be an operand to another quadtuple.

The numbering is performed starting with a root node, traversing downward, left child first, sequentially numbering each node. A node is numbered only if all the parents have been previously numbered. If one or more of the parents is as yet un-numbered, the downward traversal for that branch stops. The status of the parent node or nodes is checked with the use_count counter which is part of the node structure. If the use_count is 1, then there is only one parent un-numbered or unaccounted for, which would be the parent performing the test. While traversing the tree, when a use_count of greater than 1 is encountered, downward traversal stops, and the use_count is decremented indicating that this parent, one of the potentially multiple parents, has been numbered. This method is used instead of checking parents via a linked list.
The following is the basic algorithm developed for building, numbering, re-ordering, deleting the DAG, and generating code:

```c
roots_list = NULL;
while ( block != NULL )
    for ( block_start to block_end )
        makenode ( tuple, roots_list );
        number_tuples ( roots_list );
        reorder_tuples ( block, roots_list );
        delete_dag ( roots_list );
        generate_code ( block );
end_while;
```

Figure 4, DAG Algorithm

This algorithm is specific for the purpose of re-ordering quadtuples within the basic block prior to code generation.

4.5.1 Example

This example is provided in order to illustrate the potential advantages of using a DAG for reordering quadtuples prior to register allocation. This example demonstrates the `use_count` and node numbering as described. Assuming the following set of quadtuples:

- \( e = x + y \)
- \( c = a + b \)
- \( j = e + l \)
- \( n = o + p \)
- \( d = b + e \)
- \( i = j + k \)
- \( h = d + e \)
- \( f = c + d \)
The DAG that would be generated is as follows:

```
1 0
2 1
3 2
4 3
5 4
6 5
7 6
8 7
```

```
f = o + p
i = x + y
j = e + 1
l = j + k
d = b + e
h = d + e
c = a + b
f = c + d
```

Figure 5, DAG Example

When traversing the DAG to determine the ordering, the *use_counts* (the number inside the node) provide the information regarding the number of parents. For example, in Figure 5 as the traversing progresses from the *(d, +)* node, the 2 would indicate an un-numbered parent. The 2 would be decremented to 1, and traversal down that branch would be discontinued. As traversal downward from the *(h, +)* node progresses, the *use_count* for the *(d, +)* would have already been changed to 1, and the node would be appropriately numbered.

The node numbering that is generated is indicated by the numbers to the left on the outside of the node. According to this numbering, the re-ordered quadtuples would be as follows:

```
n = o + p
e = x + y
j = e + 1
i = j + k
d = b + e
h = d + e
c = a + b
f = c + d
```

This new ordering would tend to generate more efficient code due to the improved register allocation possibilities. This is especially true when there are a limited number of registers available.
4.5.2 Results

The DAG is not used directly for register allocation. Instead, the DAG is applied prior to code generation to re-order to quadtuples. The re-ordered quadtuples should then lend themselves to better register allocation due to the proximity of quadtuples that have common variables.

The DAG tended to have little or no impact on programs with numerous small basic blocks. Small basic blocks tend to be generated for a class of control-oriented programs. Such programs might include I/O processing (where little or no data processing is performed). This is because small basic blocks tend to have less variables, and the getreg() function was able to assign variables to registers regardless of the order. For very small blocks (three to five quadtuples), there are not enough quadtuples to allow significant re-ordering.

A parameter, MAX_DAG_SIZE, was implemented to address this issue. If the basic block had MAX_DAG_SIZE or less quadtuples, the DAG build was not performed, and the root pointer was set to NULL.

The DAG tended to improve the register allocation only for larger blocks that contained a series of quadtuples. This might occur in programs that evaluated a formula or performed a series calculations within a single basic block. The following program is used to demonstrate the results of the DAG approach.

```plaintext
UNLV Language Compiler

1:   { Example Program for DAG Re-ordering }
2:   program one
3:   var a, b, c, d, e, f, h, i: integer;
4:   var j, k, l, n, o, p, x, y: integer;
5:   begin
6:     writeln ("tst1 -- test DAG ");
7:     e := x + y;
8:     c := a + b;
9:     j := e + l;
10:    n := o + p;
11:    d := b + e;
12:    i := j + k;
13:    h := d + e;
14:    f := c + d;
15:    end.
```

This program is used as an example and, due to the un-initialized variables, will not have meaningful results. The intermediate quadtuples that are generated for the primary basic
block are listed below. For space considerations, only the quadtuples for the basic block of interest are shown. The \( \text{Ord} \) represents the new order as assigned by the DAG.

unoptimized tuples (main), 32 tuples.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Source</th>
<th>Destination</th>
<th>Ord</th>
</tr>
</thead>
<tbody>
<tr>
<td>IM_ADD</td>
<td>x(1/0)</td>
<td>y(1/0)</td>
<td>14</td>
</tr>
<tr>
<td>IM_STORE</td>
<td>t001(0/0)</td>
<td>NULL</td>
<td>e(1/0)</td>
</tr>
<tr>
<td>IM_ADD</td>
<td>a(1/0)</td>
<td>b(1/0)</td>
<td>4</td>
</tr>
<tr>
<td>IM_STORE</td>
<td>t002(0/0)</td>
<td>NULL</td>
<td>c(1/0)</td>
</tr>
<tr>
<td>IM_ADD</td>
<td>e(1/0)</td>
<td>l(1/0)</td>
<td>12</td>
</tr>
<tr>
<td>IM_STORE</td>
<td>t003(0/0)</td>
<td>NULL</td>
<td>j(1/0)</td>
</tr>
<tr>
<td>IM_ADD</td>
<td>o(1/0)</td>
<td>p(1/0)</td>
<td>16</td>
</tr>
<tr>
<td>IM_STORE</td>
<td>t004(0/0)</td>
<td>NULL</td>
<td>n(1/0)</td>
</tr>
<tr>
<td>IM_ADD</td>
<td>b(1/0)</td>
<td>e(1/0)</td>
<td>8</td>
</tr>
<tr>
<td>IM_STORE</td>
<td>t005(0/0)</td>
<td>NULL</td>
<td>d(1/0)</td>
</tr>
<tr>
<td>IM_ADD</td>
<td>j(1/0)</td>
<td>k(1/0)</td>
<td>10</td>
</tr>
<tr>
<td>IM_STORE</td>
<td>t006(0/0)</td>
<td>NULL</td>
<td>i(1/0)</td>
</tr>
<tr>
<td>IM_ADD</td>
<td>d(1/0)</td>
<td>e(1/0)</td>
<td>6</td>
</tr>
<tr>
<td>IM_STORE</td>
<td>t007(0/0)</td>
<td>NULL</td>
<td>h(1/0)</td>
</tr>
<tr>
<td>IM_ADD</td>
<td>c(1/0)</td>
<td>d(1/0)</td>
<td>2</td>
</tr>
<tr>
<td>IM_STORE</td>
<td>t008(0/0)</td>
<td>NULL</td>
<td>f(1/0)</td>
</tr>
</tbody>
</table>

The following set of quadtuples represents the intermediate code after reordering.

optimized tuples (main), 32 tuples.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Source</th>
<th>Destination</th>
<th>Ord</th>
</tr>
</thead>
<tbody>
<tr>
<td>IM_ADD</td>
<td>o(1/0)</td>
<td>p(1/0)</td>
<td>16</td>
</tr>
<tr>
<td>IM_STORE</td>
<td>t004(0/0)</td>
<td>NULL</td>
<td>n(1/0)</td>
</tr>
<tr>
<td>IM_ADD</td>
<td>x(1/0)</td>
<td>y(1/0)</td>
<td>14</td>
</tr>
<tr>
<td>IM_STORE</td>
<td>t001(1/8)</td>
<td>NULL</td>
<td>e(1/13)</td>
</tr>
<tr>
<td>IM_ADD</td>
<td>e(1/19)</td>
<td>l(1/0)</td>
<td>12</td>
</tr>
<tr>
<td>IM_STORE</td>
<td>t003(1/14)</td>
<td>NULL</td>
<td>j(1/22)</td>
</tr>
<tr>
<td>IM_ADD</td>
<td>j(1/0)</td>
<td>k(1/0)</td>
<td>10</td>
</tr>
<tr>
<td>IM_STORE</td>
<td>t006(1/23)</td>
<td>NULL</td>
<td>i(1/0)</td>
</tr>
<tr>
<td>IM_ADD</td>
<td>b(1/0)</td>
<td>e(1/25)</td>
<td>8</td>
</tr>
<tr>
<td>IM_STORE</td>
<td>t005(1/20)</td>
<td>NULL</td>
<td>d(1/25)</td>
</tr>
<tr>
<td>IM_ADD</td>
<td>d(1/28)</td>
<td>e(1/25)</td>
<td>6</td>
</tr>
<tr>
<td>IM_STORE</td>
<td>t007(1/26)</td>
<td>NULL</td>
<td>h(1/0)</td>
</tr>
<tr>
<td>IM_ADD</td>
<td>a(1/0)</td>
<td>b(1/19)</td>
<td>4</td>
</tr>
<tr>
<td>IM_STORE</td>
<td>t002(1/11)</td>
<td>NULL</td>
<td>c(1/28)</td>
</tr>
<tr>
<td>IM_ADD</td>
<td>c(1/0)</td>
<td>d(1/0)</td>
<td>2</td>
</tr>
<tr>
<td>IM_STORE</td>
<td>t008(1/29)</td>
<td>NULL</td>
<td>f(1/0)</td>
</tr>
</tbody>
</table>

In order to fully demonstrate the DAG approach, the code generated was examined.
A fragment of the code generated by the preceding program is presented with and without the DAG optimization. Both use the standard `getreg()` function as described in Section 4.4.

<table>
<thead>
<tr>
<th>Standard Code</th>
<th>DAG Re-Ordered Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1  mov ax,word ptr ss:[bp-4]</td>
<td>mov ax,word ptr ss:[bp-8]</td>
</tr>
<tr>
<td>2  add ax,word ptr ss:[bp-2]</td>
<td>add ax,word ptr ss:[bp-6]</td>
</tr>
<tr>
<td>3  mov bx,word ptr ss:[bp-32]</td>
<td>mov bx,word ptr ss:[bp-4]</td>
</tr>
<tr>
<td>4  add bx,word ptr ss:[bp-30]</td>
<td>add bx,word ptr ss:[bp-2]</td>
</tr>
<tr>
<td>5  mov cx,ax</td>
<td>mov cx,bx</td>
</tr>
<tr>
<td>6  add cx,word ptr ss:[bp-12]</td>
<td>add cx,word ptr ss:[bp-12]</td>
</tr>
<tr>
<td>7  mov dx,word ptr ss:[bp-8]</td>
<td>mov dx,cx</td>
</tr>
<tr>
<td>8  add dx,word ptr ss:[bp-6]</td>
<td>add dx,word ptr ss:[bp-14]</td>
</tr>
<tr>
<td>9  mov di,word ptr ss:[bp-30]</td>
<td>mov di,word ptr ss:[bp-30]</td>
</tr>
<tr>
<td>10 add di,ax</td>
<td>add di,bx</td>
</tr>
<tr>
<td>11 mov si,cx</td>
<td>mov si,di</td>
</tr>
<tr>
<td>12 add si,word ptr ss:[bp-14]</td>
<td>add si,bx</td>
</tr>
<tr>
<td>13 mov word ptr ss:[bp-24],ax</td>
<td>mov word ptr ss:[bp-10],ax</td>
</tr>
<tr>
<td>14 mov ax,di</td>
<td>mov ax,word ptr ss:[bp-32]</td>
</tr>
<tr>
<td>15 add ax,word ptr ss:[bp-24]</td>
<td>add ax,word ptr ss:[bp-30]</td>
</tr>
<tr>
<td>16 mov word ptr ss:[bp-20],ax</td>
<td>mov word ptr ss:[bp-28],ax</td>
</tr>
<tr>
<td>17 mov ax,bx</td>
<td>add ax,di</td>
</tr>
<tr>
<td>18 add ax,di</td>
<td></td>
</tr>
</tbody>
</table>

The differences between the two programs are bolded. The DAG approach was able to save two memory accesses (lines 7 and 12), and one instruction (line 17). However, by re-ordering the nodes an additional memory access was required (line 14). As such the total amount of improvement for this small example was 31 machine cycles or approximately 10%. More complex blocks could potentially yield even better results.

Overall the DAG approach was able to increase the overall efficiency of larger, more complex basic blocks. In doing so, no negative impacts were produced.

Refer to Appendix D for detailed examples and a copy of the source code.

### 4.6 Reaching Definitions

One type of variable lifetime analysis is *reaching definitions*. The use of reaching definitions involves the tracking of where a specific variable was last defined before reaching a given block. The use of reaching definitions for code optimizations is presented by Aho, Sethi, and Ullman [1]. The primary use is for performing transformations such as constant folding, dead-code elimination, loop invariant detection, code motion, induction variable detection, strength reduction, and induction variable elimination.
The use of variable lifetime information can also be directly applied to register allocation techniques. Specifically, lifetime information about variables, within the context of the program control flow, can be used to determine which variables would be best left in register across multiple basic blocks. This allows a more global approach to register allocation. This approach would be easily applied to the various local register allocation techniques without significant alteration in the local allocation scheme.

During local register allocation using the `getreg()` algorithm, the values in registers are written to memory at the end of each basic block. This leads to the possibility of redundant register-to-memory and memory-to-register operations for some variables. However, depending on the data-flow, the second memory-to-register operation may be required due to a loop or jump.

The first step is to perform a data-flow analysis to generate a flow-graph. The flow-graph is then used to generate the data-flow information. This information is represented in the form of the sets `inf()` and `outf()` that are used indicate which variables are live going into and out of a basic block. Since the `outf()` of a block is the `inf()` of the next block, only the `outf()` need be saved. The `outf()` of a basic block represents the variables at the end of the basic block that are either generated within the block or enter the block and are not killed.

In order to calculate the `outf()`, the `inf()`, `genf()`, and `killf()` sets are required. The `genf()` function represents variables that are generated in the block, the `inf()` function represents variables coming into the basic block, and the `killf()` function represents variables that are killed in the block. A variable may be left in a register across multiple blocks depending on the `outf()` set for a the successor block or blocks.

### 4.6.1 Data Flow Analysis

Program control flow information is required in order to accurately calculate the `outf()`'s. The generation of the control-flow information, in the form of a flow-graph, requires a separate pass over the quadruples. The predecessor or successor information may be represented, depending on what the flow-graph will be used for. The data-flow information necessary for the reaching definitions, as described in the previous section, requires predecessor information.
A predecessor flow-graph would look like the following:

![Figure 6, Predecessor Flow-Graph](image)

Each node in the graph represents a basic block, and each arrow represents a pointer to the predecessor block or blocks. There can potentially be any number of predecessor blocks for each node, depending on the control-flow.

The following algorithm was developed in order to efficiently calculate the predecessor information. This algorithm is more efficient than the algorithm presented in the literature.

```plaintext
for ( each basic block B ) do
  if ( label )
    labels[label_num] = block_num;

for ( each basic block B ) do
  if ( last_statement != any_jump )
    pred[B+1] = block_num;
  if ( last_statement == jump )
    pred[labels[jmp_lbl_num]] = block_num;
  if ( last_statement == cond_jump )
    pred[labels[jmp_lbl_num]] = block_num;
    pred[B+1] = block_num;
endfor;
```

![Figure 7, Predecessor Flow-Graph Algorithm](image)

The predecessor flow-graph is then used to calculate the reaching definitions.
4.6.2 Computing the gen() and kill() functions

The gen() and kill() sets for each block must be generated prior to computing the out(). The gen() function for a block represents the variables that are generated or defined within that block. The kill() function for a block represents the other definitions that are killed by a new definition (i.e., gen()) of the same variable. That is, when a variable is re-defined, the new definition is said to kill all other definitions of the variable until either the end of the program or yet another re-definition.

The gen() and kill() functions for single statements are very straight-forward to calculate. The gen() and kill() sets for each statement must be combined for a series of statements. A series of statements refers to a basic block for register allocation purposes. The gen() and kill() for a cascade of statements is computed as follows:

\[
\text{gen}\[B\] = \text{gen}\[B_1\] \cup (\text{gen}\[B_1\] - \text{kill}\[B_2\] )
\]

\[
\text{kill}\[B\] = \text{kill}\[B_1\] \cup (\text{kill}\[B_1\] - \text{gen}\[B_2\] )
\]

Each variable definition is numbered in order of occurrence. This number is then used as an identifier or index for that specific definition.

4.6.3 Computing the out() function

The in() and out() sets are generated based on the flow-graph information and the gen() and kill() information. For example:

\[
\text{in}\[B\] = \cup \text{out}[P]
\]

\[
\text{out}[B] = \text{gen}[B] \cup (\text{in}[B] - \text{kill}[B] )
\]

Where \( \cup \text{out}[P] \) represents the union of the out() sets of all the predecessor blocks. The predecessor information is contained in the flow-graph.
These sets are computed iteratively for all blocks in the program according to the following algorithm:

```plaintext
for ( each block B ) do
    out[B] = gen[B];
endfor;

change = TRUE;
while ( change )
    change = FALSE?
        for ( each block B ) do
            in[B] = \cup out[P];
            oldout = out[B];
            if ( out[B] != oldout )
                change = TRUE;
        endfor;
    endwhile;
```

**Figure 8, Algorithm to Calculate Reaching Definitions**

This algorithm will essentially propagate a variable generation or definition across as many block as it live (i.e., not killed by another definition) for all possible paths of program control-flow.

4.6.4 Set Representation

The sets for `in()`, `out()`, `gen()`, and `kill()` can be represented with a bit sequence. Each bit represents a variable definition. The variable definition numbering process produces a unique number for each definition in the program. This number is then used as an index into the bit sequence (i.e., bit 1 represents definition 1 and so forth).

Due to the potentially large number of sets required, this will help reduce the amount of storage space required to manipulate and save the information. The operations such as `union` and difference can be performed with logical operators. For example, the formula:

\[ out[B] = gen[B] \cup (in[B] - kill[B]) \]

can be implemented as follows:

\[ out[B] = gen[B] \mid (in[B] \& \neg (kill[B])) \]

Where `\neg` implies NOT, `\mid` implies OR, and `\&` implies AND. The relative speed at which the logical operations can be performed helps improve the efficiency of the overall algorithm.
4.6.5 Results

The code generated using the reaching definitions optimization, in almost all cases, tended to be worse with the optimization than without.

The reasons for this can be demonstrated with the following example program.

```
UNLV Language Compiler
1:  ( Example Program )
2:  program aho
3:  var i, j, m, n: integer;
4:  var a, ul, u2, u3: integer;
5:  begin
6:      i := 1;
7:      j := 1;
8:      ul := 1;
9:      u2 := 1;
10:     u3 := 1;
11:     m := ul - 1;
12:     n := m + 2;
13:     a := ul;
14:     begin
15:         while ( i < 4 ) do
16:             begin
17:                 j := j + 1;
18:                 i := i - 1;
19:                 if ( j > 4 ) then
20:                     a := ul2
21:                 else
22:                     i := u3;
23:                 end;
24:             end;
25:         end;
```

The `getreg()` function assigns registers as required to the variables and/or temporaries for the statements in the basic block on lines 9 through 16. All register values would be written to memory at the end of the basic block (line 16) as part of the normal `getreg()` strategy. This would free the registers to be used in the next basic block. The values required would need to be retrieved from memory as needed for the basic block beginning at line 18.
The following code fragment illustrates the normal operation of `getreg()` for the transition from the basic block ending at statement number 16 and the basic block beginning at statement 18:

```
mov word ptr ss:[bp-12],ax
mov word ptr ss:[bp-10],bx
mov word ptr ss:[bp-8],cx
mov word ptr ss:[bp-4],dx
mov word ptr ss:[bp-2],di
mov word ptr ss:[bp-16],si
L000000:
  mov ax,word ptr ss:[bp-16]
  cmp ax,4 ; if stmt
  jl short @000001
  jmp L000001
@000001:
  mov bx,word ptr ss:[bp-6]
  add bx,1
```

The register contents for all registers are written to memory at the end of the basic block (indicated by the label L000000). The registers are available for the next block (in this case the while loop). Assuming that all the values in registers are live beyond the end of the basic block, they would be retained in registers as demonstrated in the following example.

```
L000000:
  mov word ptr ss:[bp-12],ax
  mov ax,word ptr ss:[bp-6]
  cmp ax,4 ; if stmt
  jl short @000001
  jmp L000001
@000001:
  add si,1
```

However, a problem arises as registers are required in the succeeding basic block\(^3\). If the value required in a register is already in a register from the previous basic block, then a gain has been made. If the value required in a register is not currently in a register, not only does the value have to be loaded from memory, but the contents of an existing register must be saved to memory in order to free the register.

This completely eliminates the potential gain from retaining a value in a register. The register-to-memory operation resulting from a delayed store might now be located inside a loop as in the example above. In fact, for loops that use all registers, the loops will always contain additional memory-to-register and register-to-memory operations. A loop might use all the registers due to a large number of operations inside the loop or the use of register intensive instructions (i.e., multiply and divide). The reaching definitions

---
\(^3\) Registers will almost certainly be required due to the mandatory register use for most instructions.
optimization made an improvement only for cases where the loop was very small, the program had few variables, and the variable in question was in a register from the previous basic block.

This is due to the fact that a variable is considered live going into a block, even if the variable is not used in that block. If a variable is maintained in a register and not used in the block, it has very little chance of being allowed to stay in a register due to the small number of available registers and the mandatory register use of the instruction set.

This caused the code generated using the reaching definitions global register allocation optimization to be of a very poor quality. The code quality degraded to that of single-use register allocation due to the loading of loops with needless memory-to-register and register-to-memory operations.

Refer to Appendix E for detailed examples and a copy of the source code.

4.7 Live-Variable Analysis

Another type of variable lifetime analysis is live-variable analysis. Live-variable analysis is very similar to the reaching definitions in that the end-result is the computation of the sets $in()$ and $out()$. The method of generating the sets differs in that the $gen()$ and $kill()$ functions are replaced by the $use()$ and $def()$ functions respectively.

The primary difference between the two methods is that live-variable analysis produces a more refined analysis. This is because if a variable is defined, used, and then at some point never used again it can be considered dead. The live-variable analysis will recognize this. The reaching definitions consider a variable dead only when it is re-defined.

The reaching definitions tend to ignore undefined variables. The live-variable analysis, with the $use()$ function, handles undefined variables correctly. This advantage is expected to be short lived in the context of an executable program.

The lifetime information and the program control flow information, are used in a similar manner to determine which variables would be best left in registers across multiple basic blocks. The data-flow analysis is also performed in a similar manner as described in Section 4.6.1. The flow-graph for live-variable analysis requires successor information instead of predecessor information. The flow-graph is created differently to encompass the successor information.

This live-variable analysis information is also represented in the form of sets which provide information about which variables are live going into and out of a basic block. During the calculation of the $in()$ and $out()$ sets, the $use()$ and $def()$ sets are required. The $def()$ function represents variables that are defined in the block and the $use()$ function represents variables that are used in the block. The $in()$ function represents variables that are live coming into the basic block. Since the $in()$ of a block is the $out()$
of the next block, only the \( \text{in}() \) need be saved. Based on the \( \text{in}() \) sets for a successor block or blocks a variable may be left in a register at the end of the current block.

4.7.1 Data Flow Analysis

The final calculation of the \( \text{in}() \)’s requires program control flow information. The generation of the control-flow information, in the form of a flow-graph, requires a separate pass over the quadtuples. The data-flow information required for the live-variable analysis, as previously described, requires successor information. A successor flow-graph would look like the following:

![Successor Flow-Graph](image)

Figure 9, Successor Flow-Graph

Each node in the graph represents a basic block, and each arrow represents a pointer to the successor block or blocks. There will be at most two successor blocks. This fact allows for a less complex data structure to represent the successor flow-graph.
In order to generate the successor information, the following algorithm was developed.

\[\begin{align*}
&\text{for ( each basic block } B \text{ ) do} \\
&\quad \text{if ( label )} \\
&\quad \quad \text{blk}\{\text{label}\} = \text{block}\_\text{number}; \\
&\text{for ( each basic block } B \text{ ) do} \\
&\quad \text{if ( last\_statement == jump )} \\
&\quad \quad \text{succ}[B] = \text{blk}\{\text{label}\}; \\
&\quad \text{else} \\
&\quad \quad \text{succ}[B] = \text{next}\_\text{block}; \\
&\quad \quad \text{if ( last\_statement == cond\_jump )} \\
&\quad \quad \quad \text{succ}[B] = \text{blk}\{\text{label}\}; \\
&\quad \text{endif;} \\
&\text{endfor;}
\end{align*}\]

**Figure 10, Successor Flow-Graph Algorithm**

The successor flow-graph can then be used to perform the live-variable analysis.

4.7.2 Computing the \textit{in()} function

The \textit{in()} and \textit{out()} sets are generated based on the flow-graph information and the \textit{use()} and \textit{def()} information. For example:

\[\begin{align*}
\text{in}[B] &= \text{use}[B] \cup (\text{out}[B] - \text{def}[B]) \\
\text{out}[B] &= \cup \text{in}[S]
\end{align*}\]

Where \(\cup \text{in}[S]\) represents the union of all sets of successor blocks. The successor information is contained in the flow-graph.
These sets are computed iteratively for all blocks in the program according to the following algorithm:

```c
for ( each block B ) do
    in[B] = NULL;
endfor;
change = TRUE;
while ( change )
    change = FALSE;
    for ( each block B ) do
        out[B] = \cup in[S];
        oldin = in[B];
        in[B] = use[B] \cup (out[B] - def[B]);
        if ( in[B] \neq oldout )
            change = TRUE;
    endfor;
endwhile;
```

**Figure 11, Algorithm to Perform Live-Variables Analysis**

This algorithm will essentially propagate a variable definition across as many block as it live (i.e., not killed by another definition or by lack of use) for all possible paths of program control-flow.

4.7.3 Results

The results from the live-variable analysis were very similar to the results from the reaching definitions. The sets generated from the live-variable analysis were more refined than the sets generated from the reaching definitions, but it made little or no difference in the code generated.

The problem is related to the fact that variables are considered live going into a block, even if they are not used in that block. If a variable is maintained in a register and not used in the block, there is a very high chance that the register will be required for other operations prior to its eventual use in a successive block. If the register is required, the register-to-memory code is generated anyway. Not only does this eliminate the potential savings, but the loops tend to become loaded with additional, needless register-to-memory and memory-to-register operations. The code generated is significantly worse and almost degrades to single-use register allocation.
4.8 Variable-Use Analysis

Although the reaching definitions and live-variable analysis did not generate improved code, the basic assumption that some variables can be left in registers across a basic block is still valid. A number of cases are found in the code generated with the getreg() function where a variable is written to memory and then, in a successive block, the variable is read back into a register. The \textit{in()} and \textit{out()} sets, as generated, are to generalized to be useful for the specific architecture.

Another method of determining which variables can be retained in registers across blocks was required in order address the problems with the reaching definitions and live-variable analysis. It was determined by examining the generated code that the variables that potentially could be retained in registers across blocks were, in almost all cases, variables that were actually \textit{used} in the successor block. The \textit{in()} and \textit{out()} sets specified variables that were live, which does not necessarily mean that the variable is actually used.

A new approach was created, which was named \textit{variable-use analysis}, and subsequently implemented in order to evaluate this new approach. The implementation followed the basic strategy as outlined in live-variable analysis. The generation of the data-flow information is exactly the same as for the live-variable analysis which is outlined in Section 4.7.1.

The primary difference is that instead of the \textit{in()} function being used during code generation, the \textit{use()} function was used. The \textit{use()} function, already generated as part of the live-variable analysis, contains the information about variables are actually used in a given block. The flow-graph represents the information about the successor block or blocks.

At the end of a block, the \textit{use()} function for the successor block is used to determine which variables can be allowed to stay in registers. With the \textit{use()} function there is a reasonable assurance that the variable, since it is used in the successive block, won't need to be put back into memory before being accessed.

4.8.1 Results

The variable-use analysis tended to have little or no impact for programs with a number of small basic blocks. Small basic blocks tend to be generated for a class of control-oriented programs. Such program might include I/O processing (where little or no data processing is performed).

The variable-use analysis tended to improve the register allocation only for a series of basic blocks that used the same variables. This might occur in programs that evaluated a formula or performed a series calculations within a specific control structure.
This results of the variable-use analysis can be demonstrated with the following example program.

```
UNLV Language Compiler
1:  ( Example Program )
2:
3:  program v_use
4:  begin
5:    var i, j, a: integer;
6:  begin
7:    i := 1;
8:    j := 5;
9:    a := 0;
10:   begin
11:     while ( i < 4 ) do
12:       begin
13:         i := i + 1;
14:         j := j - 1;
15:       if ( j > 4 ) then
16:           a := 2
17:       else
18:           a := 3;
19:       end;
20:     end;
21:   end.
```

The following code fragment illustrates the normal operation of `getreg()` for the transition from the basic block ending at statement number 10 and the basic block beginning at statement number 12.

```
L000000:
  mov word ptr ss:[bp-6],ax
  mov word ptr ss:[bp-4],bx
  mov word ptr ss:[bp-2],cx
  mov ax,word ptr ss:[bp-6]
  cmp ax,4 ; if stmt
  jl short @000001
  jmp L000001
@0000001:
```

The contents of all registers are written to memory before the start of the next block. Then, the next block must obtain values from memory as demonstrated by this code fragment.
The variable-use method detects that a variable, currently in a register, is used in the successor block, and the value is retained in a register as demonstrated in the following code fragment.

```
        mov word ptr ss:[bp-4],bx
        mov word ptr ss:[bp-2],cx
        L000000:
        cmp ax,4 ; if stmt
        jl short @000001
        jmp L000001
@000001:
```

The value was retained in a register and used in the successive block. Both the register-to-memory and memory-to-register operation were saved. Since the second memory-to-register operation was in a loop, there is a potential for significant execution time savings.

If the loop requires significant register use, either from a large number of quadruples or register intensive instructions, the value that was retained in a register may need to be written to memory. This register-to-memory operation, and possibly another memory-to-register operation will then be performed inside a loop. The loop would then be loaded with some additional register-to-memory and memory-to-register operations.

Based on the register use of the loop, the variable-use global register allocation method can be slightly erratic.

### 4.9 Peephole Optimization

Peephole optimization is applied after code generation. An instruction set specific peephole optimizer is applied to a small window or subset of the generated code. The specialized peephole optimizer performs basic, architecture specific simplifications. This provides a last chance to improve the generated code or remove relatively obvious problems with the final assembly code.

A circular buffer was added to the code generator in order to apply the peephole optimizations. A specialized print routine was used for the code generator produced. The print routine maintained the last N generated lines in a circular buffer. The oldest instruction is written to the final output file as new instructions are added. The buffer allows the peephole optimizer to be executed on the last N instructions in the buffer. This makes it possible to perform any potential updates to the generated instructions while they are still in memory.
4.9.1 Optimizations

The peephole optimizer performs a series of architecture specific optimizations. These include the use of machine idioms, algebraic simplifications, and redundant-instruction elimination. The peephole optimizations are limited to basic blocks, but this does not effect most of the peephole optimizations being performed.

An attempt is made to use efficient instructions as part of the code generation process. This includes the use of short jumps\footnote{A short jump uses a displacement as opposed to a specific address. For the Intel architecture, the displacement jump is faster and has a smaller op-code.} where possible. Instructions that may be sequential in the final assembly code are often generated in different parts of the code generator. For example, if an instruction requires a register, the \texttt{getreg()} function will be called. Spill code instructions might be generated if a register must be made available by the \texttt{getreg()} function, prior to the original instruction being produced by the back-end.

The assembly instructions are reviewed for a set of possible optimizations during the peephole optimization pass. These include the algebraic simplifications of addition/subtraction of an operand by 0 or 1 or the multiplication of an operand by 0 or 1. In such cases the instruction can be either removed or altered. Multiplication by 0 can be replaced by setting the operand to zero. Multiplication of an operand by a factor of two is replaced with a corresponding shift instruction.

A check for redundant register-to-memory and memory-to-register operations is also performed. It is unlikely that a redundant load/store will be generated within a basic block with the way the \texttt{getreg()} function works. A redundant load/store across blocks will not be removed since the peephole cannot span multiple blocks. Since a block may be reached by multiple other blocks, the load/store may not be redundant depending on the program control-flow. The variable-use analysis, as described in Section 4.8, addresses this issue.

4.9.2 Results

The peephole optimizer did very well for the specific types of optimizations addressed. For some programs, depending on the code, the overall execution speed can be enhanced significantly. However, there are a limited number of optimizations that can be addressed by the peephole optimizer.
The following example program illustrates the operation of the peephole optimizer. Since it is unlikely that most of these operations would be performed as shown here, this program is an example only.

```
UNLV Language Compiler
1:   program peep
2:   var a,b,c,d: integer;
3:   begin
4:   a := 1;
5:   b := 0;
6:   c := 2;
7:   d := c * 0;
8:   a := a * 1;
9:   c := a + 1;
10:  a := c - 1;
11:  c := a - 0;
12:  a := c * 2;
13:  c := a * 4;
14:  end.
```

The peephole optimizer will recognize the multiplication by 1 and remove the instruction. Multiplication by 0 will be replaced with an xor\(^{5}\) instruction. Addition or subtraction by 1 will be replaced with an inc or dec instruction respectively. For addition or subtraction by 0, the instruction will be removed.

\(^{5}\) The xor instruction is the fastest method of setting an operand to zero.
The following code fragment illustrates the generated code as optimized by the peephole optimizer.

```assembly
mov ax, 1
mov bx, 0
mov cx, 2
mov di, ax
mov ax, cx
mov bx, 0
imul bx
mov word ptr ss:[bp-8], di
mov di, ax
mov ax, word ptr ss:[bp-8]
mov bx, 1
imul bx
mov word ptr ss:[bp-4], ax
add ax, 0
mov word ptr ss:[bp-8], ax
add ax, 1
mov word ptr ss:[bp-4], ax
sub ax, 1
mov word ptr ss:[bp-8], ax
sub ax, 0
mov word ptr ss:[bp-2], di
mov di, ax
mov bx, 2
imul bx
mov word ptr ss:[bp-4], di
mov di, ax
mov bx, 4
imul bx
```

**Standard Code**

**Peephole Optimized Code**

Blank lines have been added where instructions have been removed for clarity. There are no blank lines in the final assembly file. A significant improvement has been made in the execution time of the program. This magnitude of gain is not expected for typical programs.

Overall, the peephole optimizer will perform architecture specific optimizations on the generated code. The program, depending on how it is written, might not present any instances for performing such optimizations. When these optimizations can be made, there will always be an improvement in execution time.
4.10 Other Optimizations

There are a series of architecture independent optimizations that can be performed on the quaduples in addition to the architecture specific register allocation mechanisms and optimizations [1][2][21][22]. These optimizations include common subexpression elimination, copy propagation, loop optimizations, code motion, and induction variables strength reduction. These optimizations will indirectly effect register allocation and directly effect overall program execution time.

Such optimizations can be combined with some of the register allocation techniques previously described. Many intermediate code optimizations require the flow-graph information. The flow-graph can be created once, and used for a series of optimizations. This would help reduce the overall optimization overhead required during compilation.
Chapter 5

Conclusion

The goal of this thesis was to categorize the register allocation and optimization techniques and the related implementation issues for the Intel architecture. The techniques investigated included usage counts, graph coloring, directed acyclic graphs, register descriptors, various lifetime analysis techniques, and peephole optimizations. These techniques were investigated and categorized as to their effectiveness for the Intel architecture.

Of the register allocation and optimization techniques investigated, the customized directed acyclic graph, register descriptors, customized variable-use analysis, and peephole optimization were the most effective for the given architecture. The final code generation and register allocation strategy for the Intel architecture is as follows:

![Figure 12, Final Code Generation and Register Allocation Strategy](image)

The usage counts, graph coloring, reaching definitions, and live-variable analysis techniques were not effective for the Intel architecture. This was due to the limited number of registers available and mandatory register use requirements of the instruction set.

An additional observation from this effort is that the register use by the instruction set is as important as the number of registers available. This is especially true for target machines with a relatively small number of available registers. As part of categorizing the results of register allocation techniques in the literature, the only machine dependant issue considered is the number of registers available. Categorization for the Intel architecture based only on the number of free registers available is misleading.
Bibliography


Appendix A

Intel 80x86 Architecture Overview

This appendix presents a brief overview of the Intel 80x86 architecture. The instruction set and address translation mechanisms are unique to the Intel architecture.

This overview includes only the base architecture and not some of the processor specific extensions. The processor specific extensions are not addressed by the compiler. The processor specific extensions refer to the architecture enhancements for alternate memory addressing mechanisms (i.e., virtual memory).

One reason for the processor extensions not being addressed by the compiler is that the 80286 extension are not compatible with the 80386 extensions. This would require significantly different code to be generated based on the processor being used. Additionally, code generated would not be portable across different machines.

The primary reason the processor specific extensions are not used is that the 80286 and 80386 processor extensions are not used by the operating system (i.e., DOS). Code written using the processor extensions is unable to call operating system functions to perform system interaction (i.e., file or terminal input/output) as a result of the operating system not using the processor specific extensions.

Register Summary:

There are 14 registers for the base Intel 80x86 architecture. The registers are as follows:

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AX</td>
<td>Accumulator</td>
</tr>
<tr>
<td>BX</td>
<td>Base Register (only register allowing indexed addressing)</td>
</tr>
<tr>
<td>CX</td>
<td>Count Register</td>
</tr>
<tr>
<td>DX</td>
<td></td>
</tr>
<tr>
<td>SI</td>
<td>Index Registers (for indexed mode addressing)</td>
</tr>
<tr>
<td>DI</td>
<td></td>
</tr>
<tr>
<td>CS</td>
<td>Code Segment Register</td>
</tr>
<tr>
<td>DS</td>
<td>Data Segment Register</td>
</tr>
<tr>
<td>SS</td>
<td>Stack Segment Register</td>
</tr>
<tr>
<td>ES</td>
<td>Extra Segment Register</td>
</tr>
</tbody>
</table>
The data registers (AX, BX, CX, and DX) and the index registers are available for general use. However, there are some restrictions. The BX, SI, and DI registers are the only registers that can perform indirect addressing.

The segment registers are used for address translation, and are generally not available otherwise.

The SP, and BP registers are used to access the stack.

Instruction Summary:

The following is a brief overview of the instructions that perform arithmetic operations. These are presented to illustrate the instruction set and the mandatory register use requirements.

\[
\begin{align*}
\text{ADD} & \quad R, M \quad R = R + M \\
\text{ADD} & \quad M, R \quad M = M + R \\
\text{ADD} & \quad Ri, Rj \quad Ri = Ri + Rj \\
\text{ADD} & \quad M, M \quad \text{Illegal} \\
\text{SUB} & \quad R, M \quad R = R - M \\
\text{SUB} & \quad M, R \quad M = M - R \\
\text{SUB} & \quad Ri, Rj \quad Ri = Ri - Rj \\
\text{SUB} & \quad M, M \quad \text{Illegal} \\
\text{MUL} & \quad M \quad AX/DX = AX \times M \\
\text{MUL} & \quad R \quad AX/DX = AX \times R \\
\text{DIV} & \quad M \quad AX = AX \div M \\
\text{DIV} & \quad R \quad AX = AX \div R
\end{align*}
\]

Since most memory-to-memory operations are illegal, a register must be assigned to at least one operand. This is true for almost all instructions. Instructions such as the multiply, require the AX and DX registers.
Addressing Modes Summary:

The following is an overview of the addressing modes.

- **Direct** (contents of a variable or register).
- **Immediate** (immediate value).
- **Indirect** (contents of a specified address).
- **Displacement** (contents of a specified address plus a fixed or constant displacement).
- **Indexed** (contents of a specified address plus a variable displacement).

The indirect, displacement, and indexed addressing modes can only be performed certain registers.

```assembly
MOV R, [BX]  Indirect mode legal with BX, SI, DI only.
MOV [BX], R  Indirect mode legal with BX, SI, DI only.
MOV R, [BX][SI]  Indexed mode legal with BX SI/DI only.
MOV [BX][SI], R  Indexed mode legal with BX SI/DI only.
MOV M, M  Illegal
```

For indexed memory operations, the SI or DI register (typically used as an offset from a base) is added to the BX (typically used as a base) to form the final offset. The offset is during address translation.

Address Translation:

The address translation is how the final physical addresses are generated. Memory is logical divided into **work-areas** or **memory segments**. Then, information in the segment is accessed as an **offset** into the segment. This requires two registers, one for the segment address and another for the offset within that segment.

This allows the architecture to access $2^{20}$ (or 1 MB) of memory with 16 bit registers. The formula used for perform address translation is:

```
physical address = (segment register * 16) + offset
```

The multiple is perform using a bit shift operation for speed. The address translation is performed by the hardware automatically for all memory accesses. There is no way to circumvent address translation.

48
Appendix B

Single-Use Register Allocation

This appendix presents the some examples and the source code for the single-use register allocation.

The examples consist of the list file and the final generated assembly language file for each example. The source code follows the examples.
B.1 Examples

The following are example programs along with the assembly language source files.

UNLV Language Compiler

```plaintext
1: { Example Program }
2: 
3: program fib
4: 
5: var a, b, c, d, e: integer;
6: 
7: begin
8:   writeln ("fibonacci series");
9:   a := 1;
10:  b := 1;
11:  c := 1;
12:  e := 0;
13: 
14:   while ( (c < 30) and (e = 0) ) do
15:     begin
16:       if (a < 0) then
17:         begin
18:           writeln ("overflow.");
19:           e := 1;
20:         end
21:       else
22:         begin
23:           write (a);
24:           d := a + b;
25:           a := b;
26:           b := d;
27:         end
28:       if (e = 1) or (c mod 10 = 0) then
29:         begin
30:           writeln (" ");
31:         end
32:   end.
```

50
_TEXT segment byte public 'CODE'
assume cs:_TEXT,ds:_DATA,ss:_DATA
extrn _rdint:near
extrn _rdflt:near
extrn _wrflt:near
extrn _wrstr:near
extrn _wreol:near

fib proc far
finit
push ds
xor ax,ax
push ax
mov ax,_DATA
mov ds,ax
mov ss,ax
mov sp,_DATA-_STACK
push ax ; null display reg
push ax ; return address
push bp ; save activation record link
mov bp,sp
mov ds:disp+0,bp
sub sp,38 ; allocate space for local vars
mov word ptr ss:[bp-12],offset ds:_lit+0
push word ptr ss:[bp-12]
push ds:disp+0
call _wrstr
add sp,4
push ds:disp+0
call _wreol
add sp,2
mov word ptr ss:[bp-10],1
mov word ptr ss:[bp-8],1
mov word ptr ss:[bp-6],1
mov word ptr ss:[bp-2],0
L000000:
  mov ax,word ptr ss:[bp-6]
cmp ax,30
jnl short @000006
mov word ptr ss:[bp-14],1
jmp short @000007
@000006:
  mov word ptr ss:[bp-14],0
@000007:
  mov ax,word ptr ss:[bp-2]
cmp ax,0
jne short @000008
mov word ptr ss:[bp-16],1
jmp short @000009

51
@000008:  
    mov word ptr ss:[bp-16],0  
@000009:  
    mov ax,word ptr ss:[bp-14]  
    and ax,word ptr ss:[bp-16]  
    mov word ptr ss:[bp-18],ax  
    mov ax,word ptr ss:[bp-18]  
    or ax,ax  
    jnz @000001  
    jmp L000001  
@000001:  
    mov ax,word ptr ss:[bp-10]  
    cmp ax,0 ; if stmt  
    jl short @000002  
    jmp L000002  
@000002:  
    mov word ptr ss:[bp-22],offset ds:_lit+16  
    push word ptr ss:[bp-22]  
    push ds:_disp+0  
    call __wrstr  
    add sp,4  
    push ds:_disp+0  
    call __wreol  
    add sp,2  
    mov word ptr ss:[bp-2],1  
    jmp L000003  
L000003:  
    push word ptr ss:[bp-10]  
    push ds:_disp+0  
    call __wrint  
    add sp,4  
L000003:  
    mov ax,word ptr ss:[bp-10]  
    add ax,word ptr ss:[bp-8]  
    mov word ptr ss:[bp-24],ax  
    mov ax,word ptr ss:[bp-24]  
    mov word ptr ss:[bp-4],ax  
    mov ax,word ptr ss:[bp-8]  
    mov word ptr ss:[bp-4],ax  
    mov ax,word ptr ss:[bp-10],ax  
    mov word ptr ss:[bp-8],ax  
    mov ax,word ptr ss:[bp-6]  
    add ax,1  
    mov word ptr ss:[bp-26],ax  
    mov ax,word ptr ss:[bp-26]  
    mov word ptr ss:[bp-6],ax  
    mov ax,word ptr ss:[bp-2]  
    cmp ax,1  
    jne short @000010  
    mov word ptr ss:[bp-28],1
jmp short @000011
@000010:
    mov word ptr ss:[bp-28],0
@000011:
    mov ax,word ptr ss:[bp-6]
push bx
    mov bx,10
cwd
    idiv bx
    pop bx
    mov ax,dx
cmp ax,0
    jne short @000012
    mov word ptr ss:[bp-32],1
    jmp short @000013
@000012:
    mov word ptr ss:[bp-32],0
@000013:
    mov ax,word ptr ss:[bp-28]
or ax,word ptr ss:[bp-32]
    mov word ptr ss:[bp-34],ax
    mov ax,word ptr ss:[bp-34]
or ax,ax
    jnz @000004
    jmp L000004
@000004:
    mov word ptr ss:[bp-36],offset ds:_lit+26
    push word ptr ss:[bp-36]
push ds:_disp+0
call __wrstr
    add sp,4
    push ds:_disp+0
call __wreol
    add sp,2
    jmp L000005
L000004:
    mov word ptr ss:[bp-38],offset ds:_lit+28
    push word ptr ss:[bp-38]
push ds:_disp+0
call __wrstr
    add sp,4
L000005:
    jmp L000000
L000001:
    mov ah,4ch
    int 21h
fib endp
_TEXT ends
_DATA segment word public 'DATA'
disp dw 1 dup(0)
_lit db 102,105,98,111,110,97,99,105,32,115,101
db 114,105,101,115,0
db 111,118,101,114,102,108,111,119,46,0
db 32,0
db 44,32,0
_STACK label byte
_DATA ends
_NOUSE segment word stack 'STACK'
_NOUSE ends
end fib
UNLV Language Compiler

1:   ( Example Program )
2:  
3:   program v_use 
4:  
5:   var i, j, m, n: integer;
6:   var a, u1, u2, u3: integer;
7:  
8:   begin 
9:     m := 1;
10:    n := 1;
11:   u1 := 1;
12:   u2 := 1;
13:   u3 := 1;
14:   m := m + n;
15:   i := m - 1;
16:   j := n;
17:   a := u1;
18:  
19:     while ( i < 4 ) do 
20:       begin 
21:         i := i + 1;
22:         j := j - 1;
23:         if ( j > 4 ) then 
24:           a := u2
25:         else 
26:           i := u3;
27:       end;
28:   end.

55
mov    word ptr ss:[bp-24],ax
mov    ax,word ptr ss:[bp-24]
mov    word ptr ss:[bp-16],ax
mov    ax,word ptr ss:[bp-14]
sub    ax,1
mov    word ptr ss:[bp-26],ax
mov    ax,word ptr ss:[bp-26]
mov    word ptr ss:[bp-14],ax
mov    ax,word ptr ss:[bp-14]
cmp    ax,4 ; if stmt
jg     short @000002
jmp    L000002
@000002:
    mov    ax,word ptr ss:[bp-4]
    mov    word ptr ss:[bp-8],ax
    jmp    L000003
L000002:
    mov    ax,word ptr ss:[bp-2]
    mov    word ptr ss:[bp-16],ax
L000003:
    jmp    L000000
L000001:
    mov    ah,4ch
    int    21h
v_use   endp
_TEXT    ends
_DATA    segment word public 'DATA'
    disp    dw    1 dup(0)
_STACK   label    byte
_DATA    ends
_NOUSE   segment    word    stack 'STACK'
_NOUSE   ends
    end    v_use
B.2 Source Code

The following is the source code for the single-use register allocation technique. This is
the complete back-end. No additional support requires are required. The include files
are located in Appendix J for reference.

B.2.1 Source Code, Single-Use Register Allocation

#include <stdio.h>
#define __TURBOC__
#include <alloc.h>
#else
#include <malloc.h>
#endif
#include "backend.h"
#include "defs.h"
#include "sym.h"
#include "code.h"
#include "errcodes.h"
#include "opstack.h"
#include "optimize.h"
#include "regs.h"
#include "tuples.h"

extern FILE *asmout;

extern int debug;
extern int gen_code;
extern Sym *display[];
extern Sym *progname;
extern struct reg_desc registers[];
extern struct stack stack[];
extern int stack_size;
extern int temp_labels;
extern int locals;
extern int level;
extern int tuple_index;
extern int max_level;

extern struct basic *blocks;
extern struct basic *block_start;

extern struct tuple tuples[];

struct data_item *data_list = NULL;
struct data_item *data_list_head = NULL;
int data_list_offset = 0;
int last_level = 0;
/*
 * backend() — generate actual assembly language instructions from
 * the quadtuples...
 */

extern char *reg_str[];
extern char *char_reg_str[];

void backend()
{
    extern char *gen_addr();
    extern char *gen_addr2();
    extern void add2reg();
    extern void allocate_temp_storage();
    extern void free_reg();
    extern void make_segments();
    extern void store_registers();
    extern void print_reg_chain();
    extern void free_ax_dx();
    int x_reg;
    int y_reg;
    int z_reg;
    char x_reg_buf[30];
    char y_reg_buf[30];
    Sym *x_sym = NULL;
    Sym *y_sym = NULL;
    Sym *z_sym = NULL;
    struct opstack *x = NULL;
    struct opstack *y = NULL;
    struct opstack *z = NULL;
    struct reg_desc *work;
    int i;
    int j;
    int count;
    int reg;
    int arg_size = 0;
    int if_stmt;
    struct tuple *cur;

    if (!gen_code)
        return;

    (void) allocate_temp_storage();

    /*
 * generate code with register assignment for each basic block.
 */

blocks = block_start;
while (blocks != NULL)
{
    /*
    * initialize register descriptor table
    */
    for (i = 0; i < NUMREG; i++)
    {
        registers[i].count = 0;
        registers[i].symbol = NULL;
        registers[i].next = NULL;
    }

    for (i=blocks->start; i<blocks->start+blocks->number; i++)
    {
        if (debug)
        {
            fprintf(asmout, "; tuple #%d%s
", i, i == blocks->start ? " — start of basic block" : ");
            printf("\n tuple #%d%sn", i, i == blocks->start ? " — start of basic block" : ");
        }
        cur = &tuples[i];
        x = &cur->result;
        y = &cur->op1;
        z = &cur->op2;
        if (x->type & OPSTACK_VAR)
            x_sym = x->value.symtab;
        if (y->type & OPSTACK_VAR)
            y_sym = y->value.symtab;
        if (z->type & OPSTACK_VAR)
            z_sym = z->value.symtab;

        switch (cur->code)
        {
        case IM_AVAL:
            z_sym = z->value.symtab;
            arg_size += z_sym->size;
                        if (z->type & OPSTACK_INT)
                fprintf(asmout,"\tpush\t%s
", gen_addr(z));
            else if (z->type & OPSTACK_STRING)
                fprintf(asmout,"\tpush\t%sn", gen_addr(z));
            else if (z->type & OPSTACK_REAL)
            {
                fprintf(asmout,
                        "\tpush\tword ptr ss:[bp+%d]\n", z_sym->offset+2);
                fprintf(asmout,
                        "\tpush\tword ptr ss:[bp+%d]\n", z_sym->offset);
            }
            break;
}
case IM_AREF:
    z_sym = z->value.symtab;
    arg_size += z_sym->size;
    if ((z_sym->type & SYM_TYPE_STRING) == 0)
    {
        fprintf(asmout, "\texttt{lea}\textrm{ax},%s\n", gen_addr(z));
        fprintf(asmout, "\texttt{tpush}\textrm{ax}\n");
    }
    else
        fprintf(asmout, "\texttt{push}\textrm{ax}\n", gen_addr(z));
        break;

case IM_CALLP:
    case IM_CALLF:
    case IM_CALLF:
        z_sym = z->value.symtab;
        fprintf(asmout, "\texttt{tpush}\texttt{ds:disp%+d}\n", level*2);
        fprintf(asmout, "\texttt{tcall}\texttt{sp},%s\n", z_sym->symbol);
        fprintf(asmout, "\texttt{add}\texttt{sp,%d}\n", arg_size+SIZE_INT);
        arg_size = 0;
        break;

case IM_END:
    fprintf(asmout, "\texttt{TEXT}\texttt{ends}\n");
    (void) make_segments();
    fprintf(asmout, "\texttt{tend}\texttt{sp}\n", z->value.symtab->symbol);
    break;

case IM_ENDF:
    case IM_ENDP:
        z_sym = z->value.symtab;
        if (z_sym == progname)
        {
            fprintf(asmout, "\texttt{mov}\texttt{ah,4ch}\n");
            fprintf(asmout, "\texttt{int}\texttt{21h}\n");
        }
        else
        {
            if (cur->code == IM_ENDF)
            {
                if (z_sym->type & SYM_TYPE_REAL)
                {
                    if (z_sym->reg != R0_REG)
                        fprintf(asmout, "\texttt{tfld}\texttt{sp}\n", gen_addr2(z_sym));
                }
                if (z_sym->level+1 != last_level)
                { /* last variable was indexed off of another display register */

61
f printf asmout, "t mov\tbp,ds:disp%d\n", (z_sym->level+1)*SIZE_INT);
  last_level = z_sym->level;
}
  printf asmout, "t mov\tbp,ax\n";
  printf asmout, "t mov\tsp,bp\n";
  printf asmout, "t pop\tbp\n";
  printf asmout, "t ret\n";
}
  printf asmout, "%s\tendp\n", z->value.symtab->symbol);
break;

case IM_PROG:
  printf asmout, "\_TEXT\tsegment\tbyte public 'CODE'\n")
  printf asmout, "\tasssume\tcs:TEXT,ds:DATA,ss:DATA\n")
    printf asmout, "\textrn\t_rdint:near\n")
    printf asmout, "\textrn\t_rdflt:near\n")
    printf asmout, "\textrn\t_rwrint:near\n")
    printf asmout, "\textrn\t_wrflt:near\n")
    printf asmout, "\textrn\t_wreol:near\n")
break;

case IM_FUNC:
  case IM_PROC:
    last_level = level;
    z_sym = z->value.symtab;
    if (z_sym == progname)
      /* main program setup */
      printf asmout, "%s\tproc\tfar\n", z_sym->symbol);
      printf asmout, "\tfinit\n")
      printf asmout, "\tpush\tds\n")
      printf asmout, "\txor\tax,ax\n")
      printf asmout, "\tpush\tax\n")
      printf asmout, "\tmov\tax,DATA\n")
      printf asmout, "\tmov\tds,ax\n")
      printf asmout, "\tmov\tssp,ax\n")
      printf asmout, "\tmov\tsp,DATA-_STACK\n")
      printf asmout, "\tpush\tax/t; null display reg\n")
      printf asmout, "\tpush\tax/t; return address\n")
    }
else
  printf asmout, "%s\tproc\tnear\n", z_sym->symbol);
  printf asmout, "\tpush\tbp/t; save activation record link\n")
  printf asmout, "\tpush\tsp\n")
  printf asmout, "\tmov\tbp,sp\n")
  printf asmout, "\tmov\tds:disp%d,bp\n", last_level*SIZE_INT);
fprintf(asmout,
   "tsub\tsp,%d\t; allocate space for local vars\n",
   stack[level].size);
break;

case IM_STORE:

    if (!(x->type & OPSTACK_VAR))
    error(ERR_INTR, "IM_STORE result not a variable",
        NULL, ABORT);

    if (y->type & OPSTACK_COND)
    {
        /* special case - load a constant into a variable */
        x_sym->mem = FALSE;
        if (x_sym->type & SYM_TYPE_REAL)
        {
            x_sym->reg = NO_REG;
            fprintf(asmout, "tfd\ts\n", gen_addr(y));
            fprintf(asmout, "tfs\tps\n", gen_addr2(x_sym));
            fprintf(asmout, "tfwait\n");
        } else
        {
            x_sym->reg = NO_REG;
            fprintf(asmout, "tmov\ts,%s\n",
                gen_addr2(x_sym), gen_addr(y));
        }
    break;
    } else
    {
    if (x_sym->type & SYM_TYPE_REAL)
    {
        x_sym->reg = NO_REG;
        fprintf(asmout, "tfd\ts\n", gen_addr(y));
        fprintf(asmout, "tfs\tps\n", gen_addr2(x_sym));
        fprintf(asmout, "tfwait\n");
    } else
    {
        x_sym->reg = NO_REG;
        fprintf(asmout, "tmov\tax,%s\n", gen_addr(y));
        fprintf(asmout, "tmov\ts,ax\n",
            gen_addr2(x_sym));
    }
    } break;

63
case IM_F2I:
    fprintf(asmout, "%s\n", gen_addr(y));
    fprintf(asmout, "%s\n", gen_addr(x));
    fprintf(asmout, "%s\n", gen_addr(z));
    break;

case IM_I2F:
    fprintf(asmout, "%s\n", gen_addr(y));
    fprintf(asmout, "%s\n", gen_addr(x));
    fprintf(asmout, "%s\n", gen_addr(z));
    break;

case IM_RDIV:
    fprintf(asmout, "%s\n", gen_addr(y));
    fprintf(asmout, "%s\n", gen_addr(x));
    fprintf(asmout, "%s\n", gen_addr(z));
    break;

case IM_ADD:
    case IM_SUB:
        if (x_sym->type & SYM_TYPE_REAL)
        {
            /* real addition/subtraction */
            fprintf(asmout, "%s\n", gen_addr(y));
            if (cur->code == IM_ADD)
                fprintf(asmout, "%s\n", gen_addr(z));
            else
                fprintf(asmout, "%s\n", gen_addr(z));
            fprintf(asmout, "%s\n", gen_addr(x));
            fprintf(asmout, "%s\n", gen_addr(z));
        }
        else
        {
            /* integer add/sub */
            fprintf(asmout, "\tmov\tax,%s\n", gen_addr(y));
            if (cur->code == IM_ADD)
                fprintf(asmout, "\tadd\tax,%s\n", gen_addr(z));
            else
                fprintf(asmout, "\tsub\tax,%s\n", gen_addr(z));
            fprintf(asmout, "\tmov\t%ax,%s\n", gen_addr(x));
        }
        break;
case IM_LOR:
case IM_AND:
    fprintf(asmout, "\tmov\tax,\$s\n", gen_addr(y));
    if (cur->code == IM_LOR)
        fprintf(asmout, "\tor\tax,\$s\n", gen_addr(z));
    else
        fprintf(asmout, "\tand\tax,\$s\n", gen_addr(z));
    fprintf(asmout, "\tmov\t$ax,\%s\n", gen_addr(x));
    break;

case IM_REQ:
case IM_RNE:
case IM_RLE:
case IM_RLT:
case IM_RGE:
case IM_RGT:
    if (tuples[i+1].code == IM_JMPZ)
    {
        tuples[i+1].code = IM_NOP;
        if_stmt = TRUE;
    }
    else
        if_stmt = FALSE;

    /* load y into register */
    fprintf(asmout, "\tmov\tax,\$s\n", gen_addr(y));
    strcpy(x_reg_buf, gen_addr(x));

    if (if_stmt)
    {
        int code;
        fprintf(asmout, "\tcmp\tax,\$s\t; if stmt\n", gen_addr(z));
        code = cur->code;
        i++;
        cur = &tuples[i];
        if (code == IM_REQ)
            fprintf(asmout, "\tje\tshort @%6.6d\n", cur->label);
        else if (code == IM_RNE)
            fprintf(asmout, "\tjne\tshort @%6.6d\n", cur->label);
        else if (code == IM_RLT)
            fprintf(asmout, "\tjl\tshort @%6.6d\n", cur->label);
        else if (code == IM_RLE)
            fprintf(asmout, "\tjle\tshort @%6.6d\n", cur->label);

        // more cases
    }
else if (code == IM_RGT)
    fprintf(asmout, "\tjgt\tshort @%6.6d\n", cur->label);
else if (code == IM_RGE)
    fprintf(asmout, "\tjge\tshort @%6.6d\n", cur->label);
    fprintf(asmout, "\tjmp\tL%6.6d\n", cur->label);
    fprintf(asmout, "%@%6.6d:\n", cur->label);
}
else {
    fprintf(asmout, "\tcp\tax,%s\n", gen_addr(z));
    if (cur->code == IM_REQ)
        fprintf(asmout, "\tjne\tshort @%6.6d\n",
                temp_labels);
    else if (cur->code == IM_RNE)
        fprintf(asmout, "\tte\tshort @%6.6d\n",
                 temp_labels);
    else if (cur->code == IM_RLT)
        fprintf(asmout, "\tjnl\tshort @%6.6d\n",
                 temp_labels);
    else if (cur->code == IM_RLE)
        fprintf(asmout, "\tjne\tshort @%6.6d\n",
                 temp_labels);
    else if (cur->code == IM_RGT)
        fprintf(asmout, "\tcgt\tshort @%6.6d\n",
                 temp_labels);
    else if (cur->code == IM_RGE)
        fprintf(asmout, "\tnge\tshort @%6.6d\n",
                 temp_labels);
    fprintf(asmout, "\tmov\tax,%s\n", x_reg_buf);
    fprintf(asmout,
            "\tjmp\tshort §%6.6d\n@%6.6d:
\tmov\tax,0\n",
            temp_labels+1,temp_labels,x_reg_buf);
    fprintf(asmout, "%@%6.6d:\n",temp_labels+1);
    temp_labels += 2;
}
break;
case IM_IDIV:
case IM_MOD:
    if (y_sym == NULL)
        y_reg = NO_REG;
    else
        y_reg = y_sym->reg;
    (void) free_ax_dx(); /* DX:AX are required */
    strcpy(y_reg_buf,gen_addr(y));
x_reg = AX_REG;
x_sym->reg = AX_REG;
    strcpy(x_reg_buf,gen_addr(x));
    if (y_reg != AX_REG)
        fprintf(asmout, "\tmov\tax,%s\n",y_reg_buf);
if (z->type & OPSTACK_CONST)
{
    /* special case - cannot have immediate mode */
    if (z->type & OPSTACK_INT)
    {
        count = 0x7fff;
        z_reg = NO_REG;
        for (j = LOWER_INT_REG; (z_reg == NO_REG) &&
            (j <= UPPER_INT_REG); j++)
        {
            if ((j != AX_REG) && (j != DX_REG))
                if (registers[j].count < count)
                    {
                        count = registers[j].count;
                        z_reg = j;
                    }
        }
        if (z_reg == NO_REG)
            (void) error(ERR_INTERR,
                "cannot assign constant to reg.",
                NULL, ABORT);
        printf(asmout, "\tpush \%s\n", reg_str[z_reg]);
        printf(asmout, "\tmov \%s,%s\n",
            reg_str[z_reg], gen_addr(z));
    } /* general case */

    printf(asmout, "\tcwd\n");
    printf(asmout, "\tidiv \%s\n",
        (z->type & OPSTACK_CONST) ? reg_str[z_reg] :
        gen_addr(z));
    if (cur->code == IM_MOD)
        x_reg = DX_REG;
/* build register descriptor - register-based result */
(void) add2reg(x_reg, x);
/* free registers for y and z -- if necessary */
if (debug)
    { (void) print_reg_chain("free_ax_dx free", AX_REG);
      (void) print_reg_chain("free_ax_dx free", DX_REG);
    }
if (x_reg == AX_REG)
    { free(registers[DX_REG].next);
      registers[DX_REG].count = 0;
      registers[DX_REG].next = NULL;
    }
else
{
    free(registers[AX_REG].next);
    registers[AX_REG].count = 0;
    registers[AX_REG].next = NULL;
}

if (z->type & OPSTACK_VAR)
{
    if ((z->live == FALSE) && (z->nextuse == 0)
        && (z_sym->reg != NO_REG))
    {
        (void) free_reg(z);
    }
}
else if (z->type & OPSTACK_CONST)
{
    if (z->type & OPSTACK_INT)
    {        
        fprintf(asmout, "\top\%s\n", reg_str[z_reg]);
    }
    break;
}
case IM_MULT:
    if (x_sym->type & SYM_TYPE_REAL)
    {
        /* real multiplication */
        fprintf(asmout, "\tfld\%s\n", gen_addr(y));
        fprintf(asmout, "\tfmul\%s\n", gen_addr(z));
        fprintf(asmout, "\tfsstp\%s\n", gen_addr(x));
        fprintf(asmout, "\tfwait\n");
    }
    else
    {
        /* integer multiply */
        strcpy(y_reg_buf, gen_addr(y));
        fprintf(asmout, "\tmov\%s\n", y_reg_buf);
        if (z->type & OPSTACK_CONST)
        {
            fprintf(asmout, "\tmov\%s\n", gen_addr(z));
            fprintf(asmout, "\timul\%s\n", gen_addr(z));
        }
        else
        {
            fprintf(asmout, "\timul\%s\n", gen_addr(z));
            fprintf(asmout, "\tmov\%s,ax\n", gen_addr(x));
        }
    }
    break;
}
case IM_LABEL:
    fprintf(asmout, "L%6.6d:\n", cur->label);
    break;

case IM_JMP:
    fprintf(asmout, "%tjmp\tL%6.6d\n", cur->label);
    break;

case IM_JMPZ:
    /* make sure y is in a register, currently, we are always the first tuple in a basic block so all registers are free. !!! *** !!! this will have to be changed if data-flow analysis is done... */
    if (registers[AX_REG].count != 0)
        (void) error(ERR_INTERR, "IM_JMPZ - AX not free", NULL, ABORT);
    fprintf(asmout, "\tmov\tax,%s\n", gen_addr(y));
    fprintf(asmout, "\tor\tax,ax\n");
    fprintf(asmout, "\tjnz\t@%6.6d\n", cur->label);
    fprintf(asmout, "\tjmp\tL%6.6d\n", cur->label);
    fprintf(asmout, "@%6.6d:\n", cur->label);
    break;

case IM_JMPNZ:
    /* make sure y is in a register, currently, we are always the first tuple in a basic block so all registers are free. !!! *** !!! this will have to be changed if data-flow analysis is done... */
    if (registers[AX_REG].count != 0)
        (void) error(ERR_INTERR, "IM_JMPNZ - AX not free", NULL, ABORT);
    fprintf(asmout, "\tmov\tax,%s\n", gen_addr(y));
    fprintf(asmout, "\tor\tax,ax\n");
    fprintf(asmout, "\tjz\t@%6.6d\n", cur->label);
    fprintf(asmout, "\tjmp\tL%6.6d\n", cur->label);
    fprintf(asmout, "@%6.6d:\n", cur->label);
    break;

blocks = blocks->next;

printf("%d bytes allocated for temporary and local variables.\n", stack[level].size);
printf("%d intermediate code tuples generated.\n", tuple_index);
tuple_index = 0;
/*
 * make_segments() — generate assembly language statements to
 * define the contents of the data and stack
 * segments.
 */
void make_segments()
{
    fprintf(asmout, "\tsegment\tword public 'DATA'\n");
    fprintf(asmout, "\tdw\t%ld dup(0)\n", max_level+1);
    data_list = data_list_head;
    if (data_list != NULL)
    {
        fprintf(asmout, "\lit\n");
        while (data_list != NULL)
        {
            if (debug)
            {
                printf("make_data_segment: type: %d, offset: ",
                        "%d, length: %d",
                        data_list->type, data_list->offset, data_list->length);
                if (data_list->type & DATA_ITEM_STRING)
                    printf("\tstring: %s\n", data_list->value.string);
                else if (data_list->type & DATA_ITEM_REAL)
                    printf("\treal: %f\n", data_list->value.rval);
            }
            if (data_list->type & DATA_ITEM_STRING)
            {
                int i, j;
                char *str;

                fprintf(asmout, "\t\tdb\t\n");
                str = data_list->value.string;
                j = strlen(str)+1;
                for (i = 0; i < j; i++)
                {
                    fprintf(asmout, "%d", *(str+i));
                    if ( ((!(i % 10)) && (i != 0)) || (*(str+i) == '\0'))
                        fprintf(asmout, "\n");
                    else
                        fprintf(asmout, "\t\tdb\t\n");
                }
            }
            else if (data_list->type & DATA_ITEM_REAL)
                fprintf(asmout, "\t\tdd\t%f\n", data_list->value.rval);
            data_list = data_list->next;
        }
    }