A Generator of Code Generator for Superscalar Architectures

Mariza A. S. Bigonha[1]
José Lucas M. Rangel Netto[2]
Roberto S. Bigonha [3]

Abstract

Modern computer architectures have motivated research for more efficient compiler techniques. These new architectures delegate the solution of the most complicated problems in code generation to the compilers. This paper describes the design of a code generator system for superscalar architectures based on a formal machine description. We also discuss several problems related to code generation for these processors.

Keywords: code generation, superscalar architectures, instruction scheduling, formal definition.

[1]DSc (PUC/RJ - 1994). Department of Computer Science, Federal University of Minas Gerais, Belo Horizonte - MG - Brazil, E-mail:mariza@dcc.ufmg.br
[2]Ph.D. (Wisconsin, E.U.A). Mathematic Institute of UFRJ and DI/PUC/RJ. E-mail: rangel@inf.puc-rio.br
[3]PhD. (UCLA/USA 1981). Department of Computer Science, Federal University of Minas Gerais, Belo Horizonte - MG - Brazil, E-mail: bigonha@dcc.ufmg.br.

1 Introduction

Superscalar processors are the focus of the code generation system described in this paper. The architecture of these processors is an evolution of the RISC (*Reduced Instruction Set*) architecture, which includes several common features, among which the most important for our purposes are: (a) the ability to execute more than one instruction per cycle; (b) the incorporation of multiple functional units operating in parallel; (c) the inclusion of a pipeline mechanism. An important advantage of these features is the ability to exploit instruction level parallelism by executing concurrently a number of operations at the various pipeline stages and in different functional units[6]. Independently of the technique used to extract these concurrent operations from an essentially sequential instruction stream, the compiler must effectively take advantage of these features in order to generate high quality code. Register allocation and instruction scheduling play a very important role in this process.

The global register allocation algorithm maps user variables and compiler-generated temporaries to machine registers over an entire procedure. The allocation is considered good if user variables stay in registers during their entire lifetime. Instruction scheduling is the process of moving instructions in order to allow them to be scheduled to different units of the processor. This process minimizes the total execution time and produces code that uses pipelines and functional units of the target machine more efficiently. The two most important points for instruction scheduling are: (i) good utilization of the target architecture and (ii) preservation of the semantics of the original program, i.e., a valid scheduling must always preserve the execution order described by the edges of the graph given by instruction dependencies.

The most important unresolved problem is to determine the necessary degree of communication between register allocation and instruction scheduling that makes possible to generate an efficient scheduling. Questions like how much these two functions must be cooperate in order to improve the generated code still remains without answer, despite the work that has been done on this subject [21, 7, 3, 4, 2]. The machine description language issue is another defying problem in the sense that no language completely covers the RISC class [7].

The goal of this work is to present a tool based on the ideas proposed by Bradlee [3, 4, 5], to help the implementation of compilers in a superscalar machine environment. This work addresses the following issue: (a) the design and formal definition of the syntax and semantics of a machine description language (LDA) that allows specification of instruction scheduling requirements along with other code generation information related to superscalar architectures; (b) the project of a retargetable code generator (GGCO) whose objective is the automatic generation of tables from the machine specification of an architecture[4] in order to guide the work of the code generator kernel; (c) the identification of the necessary level of integration between instruction scheduling and register allocation.

[4] a retargetable code generator is one that can be changed automatically from the description of a new target machine, so that it can generate code for that new target

2 Compiler Construction Methodology

In the last decade instruction selection for *Complex Instruction Set Computers* (CISC) was the biggest issue in code generators by compiler developers, as it can be seen in systems like PO [12] and successors, in CODEGEN [20] and AutoCode [11]. Since CISC architectures implement common operations in many different ways, their code generators concentrated on machine specifications so as to allow instructions to be selected by pattern matching [1]. In the case of *Reduced Instruction Set* (RISC), the phase of instruction selection in compilers for these architectures was simplified. In these new processors, all arithmetic, logical, or conditional instructions are register-based. All memory accesses are done with loads and stores, the functional units and pipeline cost are exposed to the code generator. Furthermore, RISC architecture implement most operations in only one way. As a consequence, compilers do not need to choose anymore from instructions with multiple addressing modes. Therefore, the compiler's emphasis is shifted from code selection to instruction scheduling and register allocation. As the emphasis has been changed, problems related with code generators for RISCs architectures become different from the ones related to CISC architectures. Now, to produce an efficient code generator the compiler should capture most scheduling information, like operation latencies and resource conflicts. Taking into account that the scheduler needs registers to overlap the execution of independent operations, the interaction between register allocation and instruction scheduling is very important. Less attention can be devoted to the interaction between code selection and register allocation, given the relative simplicity of the code selection process.

In practice, retargetable code generators systems specifically designed for RISC architecture do not exist. Even less for superscalar machines. The Gnu [24] and Marion systems [3] are the only ones found in the literature. Until today, Marion [3] is the only system that includes a machine description language, but it cannot model complicated features found in some superscalar architectures, like SPARC's register windows, instruction side effects, such as setting the condition code, general multiple instruction issues, and the 88000's resource contention priority scheme.

Until recently, the interpreted machine description present in the GNU system did not contain scheduling information. Now, there exists at least two GNU versions that include a method to schedule instructions. One of them uses Gibbons (et al) algorithm [16], in which register allocation is made before instruction scheduling and there is no communication between these two phases. The other one includes an algorithm developed by Tiemann [25], with target-dependency latency and resource information encapsulated.

3 The GGCO System

The code generator generator we have designed is named GGCO. The GGCO design was based on work by Bradlee [3, 4, 5]. Its architecture, depicted in Figure 1, comprises the following parts: (a) `MD.c`, which is a file containing a set of automatically generated tables and routines. (b) **gen-mdc**, a module containing a semantics description in LDA (see Section 5); (c) `MD.h`, a module containing definitions of data structures and types used in `MD.c`; (d) The *front-end* modules which correspond to the LCC file written by Fraser and Hanson [15]; (e) The *back-end* module [3] which contains

the instruction scheduling and register allocation strategies. The GGCO system receives as input a processor specification in the machine description language LDA (see Section 4), and produces and provides automatically a set of tables and functions that represent the result of executing directives of LDA. File `MD.c`, and the *front-end* and *back-end* files are processed by MAKE to produce the *mcc* compiler for the desired architecture. The *mcc* compiler receives a C input file, convert it into an intermediate language and gives control to code generator that produces an object code semantically equivalent to the input file. Figure 1 shows its architecture.

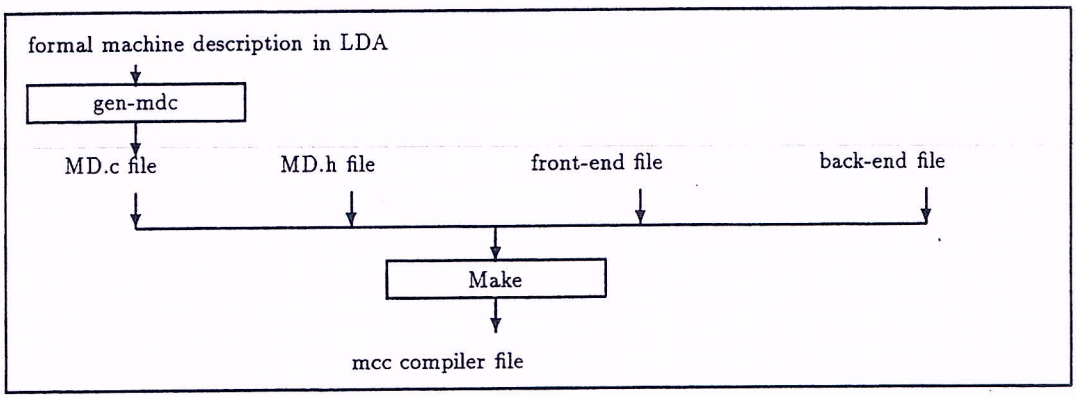

Figura 1: Code Generator Generator Architecture–GGCO

The front end of the compiler accepts ANSI C and generates code in an intermediate language of directed acyclic graphs (DAGs). This language provides the initial configuration for the code DAG. The DAG edges represent all possible operators present in the instruction set of the architecture under analysis. The front end transforms all control flow operators (i.e. for, while, if, etc.) into low level compare and branch operations. Operators with side effects of C language are changed into explicit arithmetic and branch operations.

Each code generator is produced from a machine description of a specific machine architecture and performs code selection by pattern matching and then moves control to the code generation strategy. The code generation strategy is responsible for: (1) activation of instruction scheduling and global register allocation; (2) establishement of the necessary degree of communication between these two functions; (3) inclusion of the scheduling algorithm.

The GGCO code generation strategy is the same proposed by Bradlee in the Marion system [3]. It consists of two parts, the first being strategy-independent, and the second strategy-dependent. The strategy-independent part of the back end has three components: the builder of the code DAG, the global register allocator and the constructor scheduling support. The DAG code builder is responsible for the construction of a DAG from machine instructions for each basic block. A basic block is a sequence of code that has no internal branching. Scheduling support handles low-level scheduling details; for instance, it controls the list of instructions that can be scheduled without causing delay, and verifies resource conflicts. The strategy-dependent part includes the scheduling algorithm, and tables and functions generated from the machine specification. Its modular structure permits quick reconfiguration of new strategies of instruction scheduling and register allocation. Based on this modularity, we have incorporated this new strategy in GGCO.

3.1 Instruction Scheduling

The most important data structure in the scheduling process is the scheduling graph, the code DAG. In this data structure the basic block instructions in which the program was divided are represented. In the DAG code, nodes represent instructions, and directed labeled edges represent dependences between instructions. As scheduling considered in this project is performed inside basic blocks, the precedence restrictions considered are those based on data and on control dependences. Control dependence exists only between basic blocks and their corresponding edges are derived from the control flow graph of the program. The data dependence between instructions can be a true-dependence or a false-dependence. A true-dependence, also called flow dependence, is an edge from a definition to a use. A false-dependence is classified as output dependence and anti-dependence. An anti-dependence is an edge from a use to a definition. An output-dependence is an edge between two definitions [27].

The approach used for instruction scheduling is list scheduling [14], [18], [19], [13], [16], [3], [26], [17]. It works as follows: given a code DAG, the scheduler keeps a list of instructions that are ready to be scheduled without causing a delay. On each iteration it selects the highest priority node in the ready list to be scheduled using a heuristic and then updates the list. According to [3] this approach, in general, has worst-case running time of $O(e)$, where e is the number of edges in the DAG, but the heuristic can increase the complexity. All list scheduling algorithms found in the literature use heuristics to assign priority to nodes in the ready list. The difference between these systems is in the order in which the heuristic is applied. A frequently used heuristic for assigning priority is called *maximum distance*. This heuristic is defined in terms of the length of the longest path in the code DAG from an instruction node to a leaf node. The length of a path is the sum of all edge labels along the path. The idea behind this heuristic is that the node which is the farthest from completion is the most critical, so the other nodes can be scheduled later. Another heuristic gives higher priority to nodes with more successors. The point here is that scheduling a node with several successors creates more opportunities for the scheduler in the next cycles because it permits more nodes to become ready sooner. A third choice is a heuristic which chooses a node with greater operation latency to have higher priority than that of its successors. The motivation is that scheduling such a node first will generate more opportunities to overlap the latency with other instructions. GGCO's code generation strategies use list scheduling algorithms with *maximum distance* as the primary heuristic, as in [3].

3.2 Register Allocation

The register allocation problem is similar to that of coloring a graph. In this approach, nodes in the graph represent variables and edges represent interference. Therefore, we connect two variables in the graph if there is interference between them, i.e., if they cannot simultaneously use the same register at some point in the program. The objective of the register allocator algorithm is to assign a register (*color*) to every variable such that each one has a different color from any of its neighbors [10]. With the advent of new architectures, such as superscalar and parallel, which allows the parallelism between instructions, an "optimal" coloring of interference graph does not necessarily results in a good machine utilization. This happens because in these architectures it is also necessary to take into account the reordering of instructions performed by the instruction scheduler algorithm. When instruction reordering is done after register allocation, the selection of registers may limit the possibilities to reorder instructions due to false dependencies that are introduced with the reuse of registers. On

the other hand, when instruction reordering is performed before register allocation, the number of live registers increases, implying longer register lifetimes, and thus more registers are needed and more spills may be introduced. In addition, in some cases register allocation must precede instruction scheduling since an exact register assignment is needed by the scheduler [19].

Several compilers use different graph models to implement register allocators and instruction scheduler functions [3, 17, 26]. Since the meanings of nodes and edges in these graphs are different, a simple combination is impossible. Nevertheless, the strategy used in GGCO for register allocation and instruction scheduling uses a simple common graph called *the parallel interference graph*, for representing the input program for both tasks. In this framework the emphasis is on register allocation, and the method used to allocate registers is based on Chaitin's work [9]. This strategy was originally proposed by Pinter [23].

Pinter's algorithm works as follows: to generate a parallel interference graph, first we introduce all scheduling constraints explicitly in the schedule graph. In this approach, the more edges are present in the graph the better the result will be; that happens so because what we really use are the edges that are *in the complement* of the constructed graph. The edges in the complement graph present the parallelism available in the machine for the given program. The next stage of his algorithm is to integrate those edges with the interference graph. With this new graph the register allocation algorithm can take the available parallelism into account. Scheduling is done after register allocation.

Since the minimum coloring problem is NP-complete, the number of registers is, in general, smaller than the number of colors. Thus, in practice a spilling stage is carried out. With this in mind, the problem of register allocation for superscalar machines becomes that of finding an optimal register mapping with fewer number of registers, a minimized cost of spilling and whose scheduling graph does not have false-dependences so, it is necessary to apply on the *parallel interference graph* the same heuristics used during register allocation or scheduling. One type of heuristic could eliminate edges from the graph, but to do so it is necessary to know which edges may be eliminated. This involves consideration of both the scheduler and the allocator. For instance, if removing edges that prevent false-dependences is considered, some parallelization options are lost because of register pressure. On the other hand, it is possible to remove interference edges which may lead to spill and preserve some edges that yield good parallelization.

Chaitin [9] and Pinter's [23] algorithms do not deal with the register pairs problem. Pairs of registers are often needed in processors to represent half registers in double precision load, store and move instructions. In these algorithms, a pseudo-register can be removed from the graph if it is guaranteed that there exists one physical register for it during the coloring phase, which we call an unconstrained node. This means that its degree, i.e., the number of its neighbors in the interference graph, is fewer than the number of available physical registers. Register pairs change the definition of an unconstrained node. A node now is considered unconstrained if the sum of the number of physical registers required by of its neighbors plus the number of physical registers required by itself is less than the total number of available registers. With this new definition it is possible to get a good coloring of the *parallel interference graph*, even though the demand of the neighbors of a node may be greater than the number of available registers. The reutilization of colors in neighbors whose edges do not constrain can generate a coloring of this graph that reaches the objective proposed by Pinter, namely to "find an optimal register allocation whose scheduling graph does not have false dependence".

4 LDA, The Machine Description Language

GGCO's machine description in LDA, has three main sections: (a) resource declaration, (b) compiler writer's virtual machine description, and (c) instruction definitions as in [3]. In the declaration section, the registers, machine resources, functional units, constants, memory size and other features of the architecture are specified. The compiler writer's virtual machine description contains a description of the runtime model. It offers directives to specify general purpose registers, pipeline stages, memory size, etc. The instruction section introduces the machine instructions, its functions and scheduling requirements. It also includes tree transformations necessary to match intermediate language patterns with machine language patterns. To get a perspective of the applicability of LDA features and of the GGCO system as well, we list in the sequel common architectural features not yet treated by other compiler systems. LDA possesses, in addition to all features of Marion [3], the following ones:
(1) facilities to support register windows. It is possible to specify parameters and arguments separately, to model register renaming;
(2) the machine description language, LDA, establishes resources necessary for each instruction. With this information, LDA constructs a resource vector for each instruction. Each element of the resource vector contains all resources needed on a particular cycle. In the Motorola 88010 processor [22] the priority is defined as follows: integer instructions have the highest priority, then floating point instructions, and lastly load instructions. To solve the structured hazard like a priority scheme to regulate the use of the register write-back-bus of Motorola 88010, GGCO adopts the following scheme suggested by Bradlee [3]: (a) a priority range is associated with a resource declaration in the machine description; (b) an element of a resource vector is associated with an instruction to indicate its priority; (c) priorities are examined when checking for structural hazards and schedules contain structural hazards, if they are caused by higher priority resources.
(3) To avoid control hazards, LDA specifies the number of delay slots in a instruction directive. To avoid fills branch delay slots with no-ops as in Marion [3], we are going to implement a Gross and Hennessy [19] algorithm in GGCO, including it as a separated intra-procedural pass after instruction scheduling. This algorithm attempts to fill delay slots for instructions that occur before the branch, with instructions that follow the branch target, and with instructions that follow the branch. Gross and Hennessy found that, on a machine whose branches have one delay slot that is always executed, their algorithm filled, counted statically, 90% of the delay slots.

5 Philosophy of the LDA Formal Definition

The formal definition of LDA follows the denotational method for specifying the semantics of programming languages. Figure 2 shows the most important mapping of a machine description architecture to its corresponding code generator. This mapping is defined by function `gen-mdc` which specifies the semantics of a description in LDA. The final result of this mapping is a piece of C programming language code, which corresponds to a machine dependent portion of the code generator for the described processor.

Function `gen-mdc` receives as input a file with a LDA specification in the form of an abstract syntax tree, and activates the functions `elab-declare`, `elab-vmdecl`, `elab-instr` and `elab-tables` to elaborate the several parts of a machine description and produce the machine dependent code of the code

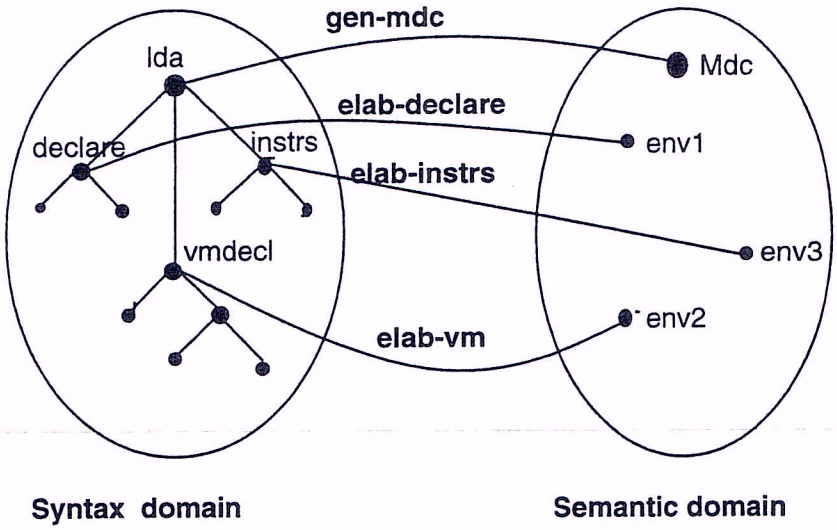

Figura 2: Mapping of LDA in Mdc.

generator. This formal definition can be seen as a code generator generator system produced from a machine description. The piece of code generated is stored in file `MD.c`. Function `elab-declare` processes LDA declarations; function `elab-vmdecl` elaborates information about the virtual machine; function `elab-instr` processes the instructions of LDA, and function `elab-tables` elaborates the final tables, producing the C code. The complete formal specification of LDA can be found in [7].

Currently, the formal definition is written in *SCRIPT* [8], which is a functional language that offers a simple notation to describe the denotational semantics of a programming language in a modular and legible style. The abstract syntax is defined in *SCRIPT* as a production list of a context free grammar. Non-terminals represent syntactic domains and tokens are "quotation" domains. The formal definition of all semantic domains are grouped in modules, which control the visibility of their denotations and provide the services associated with each domain. For each of the most important semantic domains used in the definition of mapping `gen-mdc` there exists a module *SCRIPT*, which encapsulates their denotations and provides the associated services.

The most high level function of the LDA formal definition, `gen-mdc`, maps LDA descriptions into portions of C code, which comprise the machine dependent part of the code generator. This piece of program stored in file `MD.c` includes the data structure and type definitions declared in the previous `MD.h` file (see Section 3).

The most important generated tables are those of resources and productions. The resource table describes the resources used by instructions and is one of the most important information for the scheduler. Its contents is used essentially in basic routines of instruction scheduling to verify existing conflicts and to group instructions. The production table is an array that contains information about instructions. Each array element corresponds to an instruction directive given in the description and contains: (a) a pattern tree and a replacement symbol derived from the expression given in the directive; (b) an array that indicates, for each instruction, its operand kind and their location within

the pattern and subject trees; (c) an index into an array of resource vectors; and (d) cost, latency and delay slot data. Information in this table is used by the code generator when performing instruction scheduling, register allocation and code selection. Other generated tables contain declarations, information about the virtual machine, classes and elements, auxiliary latencies, patterns, etc.

Two important target-dependent functions are the one that returns the general purpose register set for a given type and the one that returns register overlapping information and register sizes. This information is used for the creation of pseudo-registers and register allocation.

6 Conclusions

This paper described a code generator generator, named GGCO. The most important contribution is the formal specification of the processor dependent part of the code generator by means of a special purpose language. Its formal definition may be seen as a code generator system which provides, from the machine description a mapping from a processor description to its code generator, and whose final result is a program piece in C, which corresponds to the machine dependent part of the code generator for the described architecture.

The system prototype is not yet fully implemented. At the moment, for example, facilities to support register windows for the Sun SPARC station permit only one window. Additional studies must be done before the incorporation of the register allocation algorithm presented in Section 3.2 into the system, in order to evaluate and compare it with the algorithm proposed by Bradlee for scheduling and register allocation. Gross and Henessy's algorithm must still be implemented to fill the *delay slots* with valid instructions during scheduling. The implementation of the proposed solution to attend the priority schema present in some superscalar architecture is still yet to be accomplished. On the other hand, all above facilities are already available in LDA.

Referências

[1] Aigrain, P. and others, *Experience with a Graham-Glanville Code Generator*, Proceedings of the Sigplan'84 Symposium on Compiler Construction, pages: 13-24, ACM Sigplan Notices 19(6), june/1984.

[2] Benitez, Manuel E. and Davidson, Jack W., *A Portable Global Optimizer and Linker*, ACM Sigplan Notices, 23, 7, July/1988.

[3] Bradlee, David G. et al.. *The Marion System for Retargetable Instruction Scheduling*, ACM Sigplan Conference on Programming Language Design and Implementation, ACM Sigplan Notices 26(7), July/1991.

[4] Bradlee, David G. et al., *Integrating Register Allocation and Instruction Scheduling for RISCs*, ASPLOS-IV Proceedings - Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, Santa Clara, California, April/1991.

[5] Bradlee, David G., *Retargetable Instruction Scheduling for Pipelined Processors*, University of Washington, 1991, Department of Computer Science and Engineering,FR-35.

[6] Benitez, Manuel E. and Davidson, Jack W., *Code Generation for Streaming: an Access/Execute Mechanism*, ASPLOS-IV Proceedings - Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, 1991, Santa Clara California, April.

[7] Bigonha, Mariza A. Silva, *Otimização de Código em Máquinas Superescalares*, Tese de Doutorado, DI-PUC/RJ, Abril/1994.

[8] Bigonha, Roberto S. , *SCRIPT* An Object Oriented Language for Denotational Semantics (User's Manual and Reference), DCC - UFMG, RT 03/94.

[9] Chaitin, Gregory J., *Register Allocation and spilling via graph coloring*, ACM Sigplan Notices, 17, 6, ACM Sigplan Symposium on Compiler Construction, June/1982.

[10] Callahan, David and Koblenz, Brian, *Register Allocation via Hierarchical Graph Coloring*, ACM Proceeding of the ACM Sigplan'91 Conference on Programming Language Design and Implementation. Toronto, Ontario, Canada, 26-28, 1991. 192-203

[11] Costa, Paulo S. S., *Um Gerador Automático de Geradores de Código*, Tese de Mestrado, PUC-RJ, Fevereiro/1990.

[12] Davidson, Jack W., *Simplifying Code Generation Through Peephole Optimization*, DCSc - The University of Arizona, Tucson Arizona, December/1981.

[13] Fisher, Joseph A. et al., *Parallel Processing: A smart compiler and a dump machine*, ACM Sigplan Notices, 19, 6, Proceedings of the ACM Sigplan, Symposium on Compiler Construction, June/1984,

[14] Fisher, Joseph A., *Trace Scheduling: A Technique for Global Microcode Compactation*, IEEE Transactions on Computers, 30, 7, July/1981.

[15] Fraser, Christopher W. and Hanson, David R., *A Code Generation Interface for ANSI C*, Department of Computer Science, Princeton University, 1992, Research Report, CS-TR-270-90, September, Last Revised September 1992.

[16] Gibbons, Phillip B. and and Muchnick, Steven S., *Efficient Instruction Scheduling for a Pipelined Architecture*, Proceedings of the ACM Sigplan'86 - Symposium on Compiler Construction, ACM Sigplan Notices 21(7), July/1986.

[17] Goodman, James R. and Wei-Chung-Hsu, *Code Scheduling and Register Allocation in Large Basic Blocks*, International Conference on Supercomputing - Conference ACM-PRESS Proceedings, St. Malo France, July/1988.

[18] Hennessy, John and Gross, Thomas, *Code Generation and Reorganization in the Presence of Pipeline Constraints*, Conference Record of the 9^{th} Annual ACM Symposium on Principals of Programming Languages, 128-133, Albuquerque New Mexico, January/1982.

[19] Hennessy, John and Gross, Thomas, *Pospass Code Optimization of Pipeline Constrains*, ACM Transactions on Programming Languages, 1983, 5(3).

[20] Henry, Robert R., *Code Generation by Table Lookup*, Computer Science Department, University of Washington, Technical Report, # 87-07-07, FR-35 Seattle, WA 89195 USA, July/1987.

[21] Kerns, Daniel R., *Balanced Scheduling: Instruction Scheduling when Memory Latency is Uncertain*, Proceedings of the ACM Sigplan'93 Conference on Programming Language Design and Implementation, Albuquerque NM, ACM Sigplan Notices - Vol. 28 number 6 june-1993.

[22] Motorola, Inc., *MC88100 RISC Microprocessor User's Manual*, Prentice-Hall, Englewood Cliffs, second edition, New Jersey/1990.

[23] Pinter, Shlomit S., *Register Allocation with Instruction Scheduling: a New Approach* , Proceedings of the ACM Sigplan'93 Conference on Programming Language Design and Implementation, Albuquerque NM, June/1993, ACM Sigplan Notices - Vol. 28 number 6 june-1993.

[24] Stallman, Richard M., *Using and Porting GNU C*, Free Software Foundation Incorporation, Cambridge Massachusetts, 1989

[25] Tiemann, Michael D., *The GNU Instruction Scheduler*, Stanford University, Class Report, CS 343, June/1989.

[26] Warren Jr, H. S., *Instruction Scheduling for the IBM RISC System/6000 Processor*, IBM Journal of Research and Development, 34, 1, January/1990.

[27] Fernandes, Edil T and Santos Anna Dolejsi, *Arquiteturas Super Escalares: Detecção e Exploração do Paralelismo de Baixo Nível*, /vIII Escola de Computação, 3 a 12 de agosto, Gramado -RS, 1992.