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 Computer) 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 allocation. 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 defining 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 Bradie [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 (GCGO) whose objective is the automatic generation of tables from the machine specification of an architecture1 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.

2 Compiler Construction Methodology

In the last decade instruction selection for Compiler Instruction Set Computers (CISC) was the biggest issue in code generators by compiler developers, as it can be seen in systems like PO [1] 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 and most of the instructions have an immediate operand. 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 RISC 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 Tienmann [25], with target-dependency latency and resource information encapsulated.

3 The GCGO System

The code generator generator we have designed is named GCGO. The GCGO design was based on work by Bradie [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-mds, 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 GCC files written by Fraser and Hanson [15]; (e) The back-end module [3] which contains