the other hand, when instruction reordering is performed before register allocation, the number of
five 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 schedul-
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
struction 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 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 instruc-
tions. 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 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 realization 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 dependences”.

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 definition, as in [2]. 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 peripheral resources, 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 Marlo [3], the following ones:

1. facilities to support register windows. It is possible to specify parameters and arguments separately,
   so 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 structural 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 Bradie [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 filling branch delay slots with no-ops as in Marlof [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 program-
ing 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-nde which specifies the
semantics of a description in LDA. The final result of this mapping is a piece of C programming lan-
guage code, which corresponds to a machine dependent portion of the code generator for the described
processor.

Function gen-nde receives as input a file with a LDA specification in the form of an abstract syntax
tree, and activates the functions elab-declare, elab-vfield, elab-label and elab-tables to ela-
borate the several parts of a machine description and produce the machine dependent code of the code