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 an intermediate language and gives control to the code generator that produces an object code semantically equivalent to the input file. Figure 1 shows its architecture.

![Code Generator Generator Architecture-GGCO](image)

**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 operations 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. Operations 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) establishment 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 Marlin 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 delays 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 is 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 [8] this approach, in general, has worst case running time of $O(n^2) + k$, where $n$ 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 they cause interference between them, i.e., if they cannot simultaneously use the same register. The objective of the register allocator algorithm is to assign a register to each variable such that each one has a different color from any of its neighbors [10]. With interferences, an "optimal" coloring of interference graph does not necessarily results in a good machine. 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 dependences that are introduced with the reuse of registers.