Compilation

TP 8: Code generation

A. Isoard & C. Alias

credits: G. Iooss

In this TP we will study the last part of the compilation process: the final Digmips code generation. As seen in the lecture, it is usually decomposed intro three stages: (i) instruction selection, (ii) instruction scheduling, and (iii) allocation of temporaries

Exercise 1. *Selection, scheduling*

The first step is to choose the instruction in the Digmips assembly to evaluate the DAG of each basic block, and to find an execution order.
Download the full compiler `digcc.tgz` and unzip it.
Open the file `Backend.cc`.

- **backend()** (line 255) takes the intermediate representation (cfg) and produce the final code by applying successively the three following steps:
- **select()** (line 226) execute the first two steps. For each basic block, it produce a DAG and select/schedule the instructions, thanks to the `select_dag` function. The result is a CFG for which nodes are Digmips instructions over temporaries. The liveness of those temporaries have to then be computed.
- **select_dag()** (line 210) visits each root of the DAG, and produce the code in a depth first order, from left to right, with a call to `select_root`. The ordering of the instructions is therefore direct, without any attempt to optimize.
- **select_root()** (line 47) performs the actual instruction selection. As the grain of the Digmips instructions is finer that the intermediate representation, the tiling strategy does not apply, and we only need to translate each node of the DAG independently. This translation is done by the `digmips_translate` that adds a new Digmips instruction to the resulting CFG. Naturally, we should not forget to copy the results of a node into each register marked as live-out (because they are used outside the current block).

Open the file `CodeDigmips.h`.
The `CodeDigmips` class describes the Digmips instructions. It inherits from `Code` which allows to build a Digmips instruction CFG and to compute the liveness of the temporaries by using the method of CFG. The file also provides:

- constructors (lines 105–123) that adds an instruction to the provided CFG.
- macros (lines 126–137) that produces the instruction sequence equivalent to each pseudo-instructions.
- and the main translation functions (lines 144–end).

Manip.

- In the file `Backend.cc`:

  1. uncomment lines 257–274 (to display the DAG of each basic block)
  2. uncomment the line 284 (do display the Digmips code produced after selection/scheduling)
  3. comment the line 336 (to not display the final code)

  Compile and run it on `tests/simple.c`.

- We are interested by the block that starts by `alloc_0` (second DAG, lines 19–36). Explore the translation. How is `r14 = 1` translated? Why?
- Put the file into its initial state.

Exercise 2. *Allocation*

The produced code from the previous stage still manipulate temporaries. We now have to allocate registers and stack space for them. This stage is realized by `Backend.cc` (lines 287–307).

Manip.

- The last stage of the select function is to compute the liveness of the temporaries. Display the result by uncommenting the corresponding line. Tests again on tests/simple.c. Comment back the line.
- From those informations, we build an interference graph. We only need to declare a conflict between two temporaries live at the same time in the live-in of an instruction, or in the live-out. Display the interference graph by uncommenting the corresponding line. Note that certain reserved temporaries (r0,r1,r6,r7) are ignored. Its fine, as they are already allocated to corresponding registers. Comment back the line.

To allocate the registers and the stack space, we now need to color the graph. As we want the r2 temporary to be stored into the r2 register, we pre-color the graph[1]. We then start the register allocation over 4 registers. Indeed, r0 and r1 are reserved for the spill, and r6 and r7 for the stack. We only have r2-r5 free to use (that the allocator name 0-3).

Open the file `Allocation.h`. The Allocation class contains the final mapping for the temporaries: temporaru $\rightarrow$ physical temporary (register or stack).

Open the file `Allocation.cc`. The constructor starts at line 146. We use the Chaitin algorithm to build our allocation. We therefore need a way to add/remove nodes to the interference graph. We use a simple node list that list existing nodes and that we initialize with all the nodes and that the allocator will update. The allocator takes in parameter the interference graph, the "existing node" list and the number K of physical registers. It is a direct implementation of the algorithm of the lecture. Remark the recursive call over $G - \{s\}$. Once registers are allocated, we need to allocate stack space when temporaries are spilled. We simply color with $+\infty$ colors the part of the graph that will be spilled. The color with then be seen as a shift of the temporary, in the stack.

Manip.

- In `Backend.cc`, uncomments the line to display the allocation. Test over `tests/simple.c`.

Exercise 3. *Code generation*

Now that we have all we need, we simply have to produce *spill-code* each time a temporary is allocated on the stack. For each instruction produced at the first step, the `digmips_apply` function produce the stack allocated version.

Open the file `CodeDigmips.cc` and go to line 309. We start by replacing the lucky temporaries by their physical registers. Then we produce the code by inserting the *spill-code* of read registers. We then add the instruction, "patched" with physical registers, and we insert the *spill-code* of the writen register when needed.

Manip.

- Produce the final code over `tests/simple.c`.
- With the help of the produced code of exercise 1 and its allocation, check its application and the production of *spill-code*.
- How and why do we avoid spill-code in the prelude and postlude of a function?

Exercise 4. *Have fun!*

Try the other tests programs, and/or write your own.

[1] with r0, but which will actually become r2...