Lab 7
Final Code Generation

Objective

In this lab 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. First, download the full compiler and unzip it.

EXERCISE #1 ▶ **Lab6 review : Control Flow, Liveness, ...**
In the `tests` directory, you will find the running example for this lab:

- Generate the 3-address code, then the CFG on paper. Also compute live-in and live-out ranges for all variables of the 3-address code.
- Open `digcpp.ypp` and find the rule for functions. Uncomment the line to produce the CFG after the basic block extraction (A file named `cfg.dot` is produced in the current directory).
- Compare your code/CFG and the generated graph [1].
- What will be the generated DAG for the second block (`label alloc_0`)? Recall that the constructor of the DAG will perform elimination of redundancies, and constant propagation.

At this step, we have a (simplified) DAG for each block. Some work remains to be done.

Code review : `Backend.cpp` Open and read the file `Backend.cpp`:

- **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).

7.1 Instruction selection, scheduling

In this phase, for each bloc, we will choose instructions in the DIGMIPS assembly, and find an execution order.

Code review : `CodeDigmips.cpp` This class inherits from `Code` which allows to build a DIGMIPS instruction CFG and to compute the liveness of the temporaries. 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).

EXERCISE #2 ▶ **Instruction scheduling + (pre-) code generation inside blocks**
First, comment-out your code of the first exercise and uncomment the call to the `backend()` function in the parser. For the moment, this function only computes a DAG per block (like in the previous exercise) and dump them in files.

[1] The bug in the CFG construction (there should not be a link between a node and its successor if the node is a jump.) has been corrected.

- Test on the given example. The output should be 14 dot files.
- In the file `Backend.cpp`, uncomment the line that performs the instruction selection / scheduling for all blocks. Test this phase: print the dag for the second block inside the selection phase (`select` function) with:
  `if (i==1) dag->show_dot();`
- Produce the dot file of the CFG obtained after the instruction selection phase:
  `digmips_virtual_cfg->extract_basic_blocks()->show_dot();`
- Save it in your directory (convert into pdf/png).
- Comment out all the calls to `show_dot` (CFG, DAGs).

7.2 Liveness and register 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 also realized by `Backend.cpp` (lines 287–307). After recomputing the liveness information of all temporaries (done at the end of `select()`), an interference graph is build. We only need to declare a conflict between two temporaries that are alive at the same time in the live-in of an instruction, or in the live-out.

EXERCISE #3 ▶ **Interference graph**
In `Backend.cpp`:

- How many temporaries are there in the CFG of the previous exercise ? Find two temporaries that conflict, and two independant temporaries. Draw the beginning of the interference graphs for the three first blocks.
- Comment out the return to continue the algorithm in the `backend()` function.
- Compute the interference graph and produce `interferences.dot` by:
  `interferences->print_dot();`
  Visualise this graph (be careful, this could be a chock). Verify that your (non) conflicting temporaries are handled properly in this graph.

To allocate the registers and the stack space, we now need to color the graph. As we want the r_2 temporary to be stored into the r_2 register, we pre-color the graph[2].

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).

Register Allocation via graph coloring The `Allocation` class contains the final mapping for the temporaries: temporary → physical temporary (register or stack).

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.

EXERCISE #4 ▶ **Register allocation**
In `Backend.cpp`, uncomments the line to display the allocation. Test over our running example. How many temporaries will be spilled with this allocation algorithm for `example.c`?

7.3 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.

[2] with r_0, that will actually become r_2 ...

Open the file `CodeDigmips.cpp` 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.

EXERCISE #5 ▶ **Code Generation**
After the code review of `CodeDigmips.cpp`:

- Produce the final code over `tests/example.c`.
- With the help of the produced code and its allocation, check its application and the production of *spill-code*.

7.4 Exercise: SSA and register allocation

EXERCISE #6 ▶ **SSA, dominance frontier**
On the following example:

```
i = 1
j = 1
k = 0
while ( k < 100 ) {
 if ( j < 20 ) {
    j = i
    k = k + 1
 }
 else {
    j = k
    k = k + 2
 }
}
```

- Compute the CFG.
- Compute the dominance frontier of each node.
- Compute the SSA form.