# Lab 5
# Code generation with smart IRs

## Objective

- Construct the CFG.
- Compute live ranges, construct the interference graph.
- Allocate registers and produce final "smart" code.
- This is due **on TOMUSS (NO EMAIL PLEASE)**: deadline and info on the course's webpage.

During the previous lab, you wrote a dummy code generator for the MiniC language. In this lab the objective is to generate a more efficient RISCV code. We recall you that you have slides on the course webpage to help you: `https://compil-lyon.gitlabpages.inria.fr/mif08-20/capmif_labs.pdf`

**You will extend your previous code, in the same `MiniC` project, but in the `TP05/` subdirectory**. People in advance are encouraged to keep their current code, students with more difficulties will be provided a working 3 address code generation Visitor named `TP04/MiniCCodegen3AVisitor-correct.py`.

**Installations** We are going to use graphviz for visualization. If it is not already installed (e.g. on your personal machine), install it, for instance with:

```
sudo apt-get install graphviz graphviz-dev
```

You may have to install the following PYTHON packages:

```
python3 -m pip install --user networkx
python3 -m pip install --user graphviz
python3 -m pip install --user pygraphviz \
  --install-option="--include-path=/usr/include/graphviz" \
  --install-option="--library-path=/usr/lib/graphviz/"
```

If the last command errors out complaining about a missing `Python.h`, run:

```
sudo apt-get install python3-dev
```

and then relaunch the command `python3 -m pip install ...` On the university machines, you might have to update existing already installed packages:

```
python3 -m pip install --user --upgrade networkx graphviz pygraphviz
```

## 5.1 CFG construction

In class we have presented CFGs with maximal basic blocks. In this lab we will implement CFGs with minimal basic blocks that is CFG with one node per line of code/instruction (even comments in assembly code).

EXERCISE #1 ▶ **CFG By hand**
What are the expected result of the CFG construction from the 3-address code of Lab5 for each of these programs?

```
int n,u,v;
n=6;
u=12;
v=n+u;
print_int(v);
```

```
int x;
x=2;
if (x < 4)
   x=4;
else
   x=5;
print_int(x)
```

```
int x;
x=0;
while (x < 4){
   x=x+1;
}
```

EXERCISE #2 ► **CFG Construction**
The `APIRiscV` is able to deal with CFGs. `Instructions` have a list of predecessors (`self._in`) and successors (`self._out`) and a `RiscVFunction` contains the initial control point (`self._start`) from which we can traverse the graph. This feature allows us to easily construct the CFG of a program.

We give you the construction of the CFG. Each time your Visitor creates a new RISCV instruction, the CFG updates itself automatically: when adding an instruction, it creates an edge between the last instruction (`self._end`) and the instruction to be added (except for jumps and branches, where it "does the right thing").

In this exercise, you only have to understand (look at `APIRiscV.py`) and test the provided code.

When ran with `--graphs`, `MiniCC.py` prints the CFG as a PDF file (using the tool "dot"). The file is printed as `<name>.dot.pdf` in the same directory as the source file and opened automatically.

Now you can launch:

```
python3 MiniCC.py --reg-alloc smart --graphs /path/to/example.c
```

1. Test for lists of assignments (for instance `TP05/tests/provided/dataflow/df01.c`) You should see a chain of blocks.
2. Same for boolean expressions, and tests.
3. Same for while loops (the loop creates a cycle in the CFG) ....
4. Propose appropriate examples and draw nice pictures!

## 5.2 Liveness analysis and Interference graph

**From now on, you have to modify `Allocations.py`.** You already used `NaiveAllocator` and `AllInMemAllocator` in the previous lab (the mapping from temporary to physical register or memory location was provided to you, and you had to modify the 3 address code to take this mapping into account). We well now write the `SmartAllocator`, that maps temporaries to physical registers in an optimized way, and use memory (spilling) only when necessary. Read the body of `SmartAllocator.run()`, that gives the main steps of the allocation: liveness dataflow analysis, conflict graph, graph colouring, and finally 3 address code modification to get the final executable.

For the liveness analysis, we recall the notations. A variable at the left-hand side of an assignement is *killed* by the block. A variable whose value is used in this block (before any assignment) is *generated*.

$$LV_{exit}(\ell) = \begin{cases} \emptyset & \text{if } \ell = \text{final} \\ \bigcup\{LV_{entry}(\ell') | (\ell,\ell') \in flow(G)\} & \end{cases}$$

$$LV_{entry}(\ell) = \left(LV_{exit}(\ell) \setminus kill_{LV}(\ell)\right) \cup gen_{LV}(\ell)$$

The sets are initialised to $\emptyset$ and computed iteratively, until reaching a fixpoint.

EXERCISE #3 ► **Liveness Analysis, Initialisation**
In this exercice, you have to complete the method `set_gen_kill()` of the `SmartAllocator` class. This method initialises the Gen($B$) and Kill($B$) sets for each instruction (add, let, ...) in the program. Be careful to properly handle the following cases:

```
addi temp1, temp1, 12
```

and

```
bge temp_1, temp_3, lbl_foo  # temp_1 is read from
```

To test/debug this initialisation, the following statements in `Allocations.py` (in `SmartAllocator.run()`) should help you (use with `MiniCC.py --debug`, which sets `debug=True` for you):

```
if debug:
    self.print_gen_kill()
```

As an example, once initialization is implemented, running the command

```
python3 MiniCC.py --debug --reg-alloc smart TP05/tests/provided/dataflow/df04.c
```

should give the expected initialisation for `TP05/tests/provided/dataflow/df04.c`:

```
Dataflow Analysis, Initialisation
instr 0: li temp_0, 0
gen: {}
kill: {temp_0}

instr 1: li temp_1, 0
gen: {}
kill: {temp_1}

instr 2: comment
gen: {}
kill: {}

instr 3: li temp_2, 2
gen: {}
kill: {temp_2}

instr 4: mv temp_1, temp_2
gen: {temp_2}
kill: {temp_1}

instr 5: comment
gen: {}
kill: {}

instr 6: li temp_3, 4
gen: {}
kill: {temp_3}

instr 7: li temp_4, 0
gen: {}
kill: {temp_4}

instr 8: bge temp_1, temp_3, lbl_end_relational_3_main
gen: {temp_3,temp_1}
kill: {}

instr 9: li temp_4, 1
gen: {}
kill: {temp_4}

instr 10: lbl_end_relational_3_main
gen: {}
kill: {}

instr 11: beq temp_4, zero, lbl_else_2_main
gen: {temp_4}
kill: {}

instr 12: li temp_5, 4
gen: {}
kill: {temp_5}

instr 13: mv temp_1, temp_5
gen: {temp_5}
kill: {temp_1}

instr 14: j lbl_end_if_1_main
gen: {}
kill: {}

instr 15: lbl_else_2_main
gen: {}
kill: {}

instr 16: li temp_6, 5
gen: {}
kill: {temp_6}

instr 17: mv temp_1, temp_6
gen: {temp_6}
kill: {temp_1}

instr 18: lbl_end_if_1_main
gen: {}
kill: {}
```

```
instr 19: comment
gen: {}
kill: {}
```

```
instr 20: li a0, 0
gen: {}
kill: {}
```

EXERCISE #4 ► **Liveness Analysis, fixpoint. (Only test!)**

We implemented for you the fixpoint iteration as a method (`run_dataflow_analysis`) in `Allocations.py` "while it is not finished, store the old values, do an iteration, decide if its finished". The `run_dataflow_analysis` method makes calls to `dataflow_one_step` instruction methods. The result (live in, live out sets of variables, are stored in `_mapin` and `_mapout` member sets of the `SmartAllocator` class).

What you have to do in this exercice is to check that the results that are obtained with with analysis are correct at least for the examples of the `TP05/tests/provided/dataflow/` directory.

To do so, the following lines should help you (again, using `--debug`) in the same file:

```
mapin, mapout = self.run_dataflow_analysis()
if debug:
    self.print_map_in_out()
```

As an example, here is the expected output for `TP05/tests/provided/dataflow/df04.c`:

```
Dataflow Analysis
finished in 6 iterations
In: {li temp_0, 0: {}, li temp_1, 0: {}, comment: {}, li temp_2, 2: {},
 mv temp_1, temp_2: {temp_2}, comment: {temp_1}, li temp_3, 4: {temp_1},
 li temp_4, 0: {temp_1,temp_3},
 bge temp_1, temp_3, lbl_end_relational_3_main: {temp_1,temp_4,temp_3},
 li temp_4, 1: {}, lbl_end_relational_3_main: {temp_4},
 beq temp_4, zero, lbl_else_2_main: {temp_4}, li temp_5, 4: {},
 mv temp_1, temp_5: {temp_5}, j lbl_end_if_1_main: {}, lbl_else_2_main: {},
 li temp_6, 5: {}, mv temp_1, temp_6: {temp_6}, lbl_end_if_1_main: {},
 comment: {}, li a0, 0: {}}

Out: {li temp_0, 0: {}, li temp_1, 0: {}, comment: {}, li temp_2, 2: {temp_2},
 mv temp_1, temp_2: {temp_1}, comment: {temp_1}, li temp_3, 4: {temp_1,temp_3},
 li temp_4, 0: {temp_1,temp_4,temp_3},
 bge temp_1, temp_3, lbl_end_relational_3_main: {temp_4}, li temp_4, 1: {temp_4},
 lbl_end_relational_3_main: {temp_4}, beq temp_4, zero, lbl_else_2_main: {},
 li temp_5, 4: {temp_5}, mv temp_1, temp_5: {}, j lbl_end_if_1_main: {},
 lbl_else_2_main: {}, li temp_6, 5: {temp_6}, mv temp_1, temp_6: {},
 lbl_end_if_1_main: {}, comment: {}, li a0, 0: {}}
```

EXERCISE #5 ► **Interference graph**

The interference graph contains an edge $(x, y)$ if temporaries $x$ and $y$ are in conflict. It is built by `SmartAllocator.build_inter` that calls the `interfere` method.

We recall that two temporaries $x, y$ are in conflict if they are simultaneously alive after a given instruction, which means:

- There exists a block (an instruction) $b$ and $x, y \in LV_{out}(b)$
- OR There exist a block $b$ such that $x \in LV_{out}(b)$ and $y$ is defined in the block
- OR the converse.

To understand why the last two cases are needed, consider the following list of instructions:

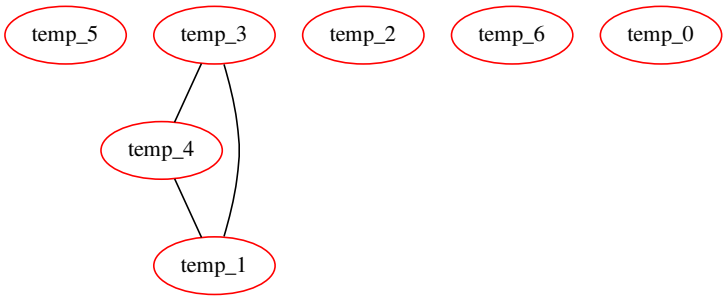
```
y=2
x=1 // Can x and y be mapped to the same place? Obviously not.
z=y+1
```

where $x$ is not alive after the `x=1` statement, however $x$ is in conflict with $y$ since we generate the code for `x=1` while $y$ is alive[1].

From the result of the previous exercise, construct the interference graph (the job is done by function `build_interference_graph`) of your program. The code iterating over temporaries is given, but calls a function `interfere(temp1, temp2)` for each pair of temporaries, which you have to implement (between 5 and 10 lines of code implementing the definition above). We give you a undirected graph API (`TP05/LibGraph.py`) for that. Use the `print_dot` method and relevant tests to validate your code.

In this exercise, we care about correctness more than complexity. It is OK to write an $O(n^3)$ algorithm (for each $t_1$, for each $t_2$, for each control point $c$, check whether $t_1$ and $t_2$ have a conflict).

As an example, here is the conflict graph that should be obtained for `df04.c` (command line as usual):




## 5.3 Register allocation and code production

We will implement the following algorithm for $k$ register allocation:

- Color the interference graph with an infinite number of colors, using the first ones as much as possible.
- The first $k-3$ colors will be mapped to registers.
- All the other variables will be allocated on the stack. For each color, we use a memory location according to their coloring number.

Then the 3 address code modification:

- For non-spilled variable: replace the temporary with its associated color/register, as we did for the naive allocator.
- For spilled variables: add `ld` / `sd` statements as needed and replace the temporary with one of `s1`, `s2`, `s3` as we did for the "all in mem" allocator.

Some help:

- `GP_REGS` is an array of registers available for the register allocator.
- An element of type Register can be obtained from a given register color with the helper function `GP_REGS[coloringreg[xxx]]`, where `coloringreg` is graph coloring returned by the `.color()` function, and for offsets you have a constructor `Offset(FP, xxx)` (all in `Operands.py`).
- Be careful with types when dealing with the graph. As the comment in `Allocations.py` states, `self._igraph` contains only elements of type string, while the `alloc_dict` map given to `self._pool.set_temp_allocation()` must have `Temporary` objects as keys. There is no easy way to retrieve a `Temporary` object from its name, but it is easy to get the name as a string from a `Temporary`: just use `str()`. The easiest way to build `alloc_dict` is probably to iterate over all the temporaries of the program (available in `self._pool._all_temps`), and for each temporary check the corresponding color to associate it to the right register or memory location in `alloc_dict`.

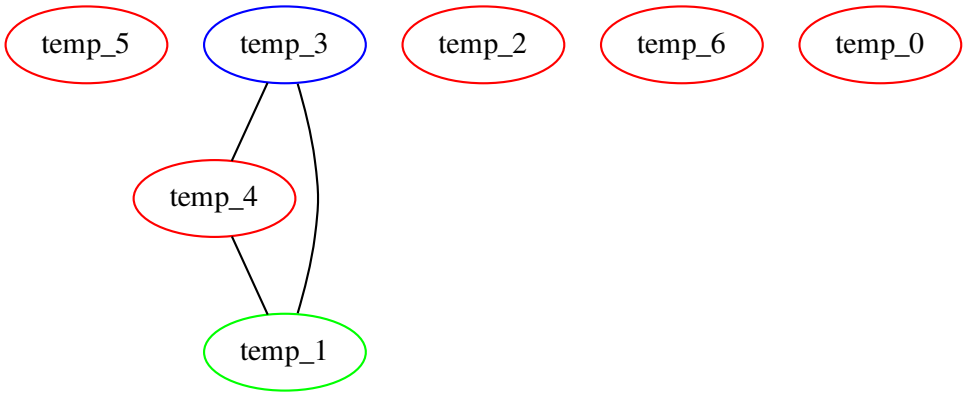
[1] Another solution consists in eliminating dead code before generating the interference graph.

EXERCISE #6 ▶ **Smart Register Allocation: implement!**
In this exercice, you have first to complete method `smart_alloc()` to perform an allocation based on a graph colouring. The purpose of this method is to allocate a physical register or a memory location for each temporary in the program. Next, you will have to complete the function `replace_smart()` that replaces the temporary operands of a given instruction according to the allocation computed by `smart_alloc()`.

Use the algorithm and the coloration method of the `LibGraphes` class to allocate registers (or a memory location) in `smart_alloc()`. Comments will help you design this (non trivial) function. The allocation is followed by statement rewriting, like in previous lab. You need to implement it in `Allocations.py` (`replace_smart`): it is very similar to the previous lab's version, but you have to deal with both memory locations and registers in the same function.

Validate your allocation on tiny well chosen test files (especially tests that augment the register pressure) and all the benchmarks of the previous lab. We adapted the previous script for that.

On the `df04.c` example, the graph coloring succeeds with:




EXERCISE #7 ▶ **Massive tests**
Comment out all the `print_dot` instructions, debug, ... and test on all test files you have. **In particular, we do not want any pdf file to be opened when we will use `make tests` on your delivered code.**

## 5.4 Final delivery

EXERCISE #8 ▶ **Delivery**
This lab is due on TOMUSS (deadline on the course's homepage). Python code and C testcases will be graded. We recall that your work is **personal** and code copy from any source or sharing is **strictly forbidden**. As usual, upload an archive containing the whole `MiniC` directory (`make tar` does that for you).

The `README-gencode.md` file won't be taken into account in the grade. It is good habit to document your work, but you may keep it unmodified if you don't have anything specific to mention for this lab.