# Module 3
# Program Analysis and Optimizations

**Compilers and Execution Environments (ID2202)**
7.5 hp

David Broman
KTH Royal Institute of Technology

Version 1.2, Monday 6th December, 2021

**Abstract**

Welcome to Module 3 of the course! In this module you will learn about how to make your compiled program run faster. In particular, we will look into register allocation, to avoid spilling. Furthermore, we will discuss program representations especially designed for solving program optimization problems.

This document outlines the different exercises for Module 3. The exercises are divided into three levels: Satisfactory (S), Good (G), and Very Good (VG). Depending on your goal of the final grade in the course, you may choose which exercises that you perform. Please note that the minimum for passing the course is to complete all exercises at level Satisfactory (S). To reach level (G) for this module, all exercises at level (S) and (G) have to be completed. Finally, to achieve level (VG), you need to finish all exercises at all three levels. For details about the grading, see the course memo on Canvas.

## Contents

**1 Level: Satisfactory (S): 2**
1.1 Individual theoretical exercise . . . . . 2

**2 Level: Good (G): 2**
2.1 Liveness Analysis . . . . . 2

**3 Level: Very Good (VG): 4**
3.1 Interference Graph and Graph Coloring . . . . . 4

# 1 Level: Satisfactory (S):

The goal of the tasks at level Satisfactory (S) is to practice some of the theories described in the lectures. The exercises cover theory from both Module 2 and Module 3.

## 1.1 Individual theoretical exercise

A number of theoretical examination tasks will be available in your Git repository under folder `exercises/module3/`. These tasks are individual, that is, each student in the course gets an individual set of exercises.

You are not allowed to talk to anyone about these exercises or to get any help from anyone to solve them. You are not allowed to publish or distribute your solutions or the exercises. You must write down your solution using pen and paper and then take a photo of the solution. For the exercises marked as satisfactory, submit your solution on Canvas (as one or several image files). You can find the submission page under "Assignments" on Canvas. Note that you might be asked to explain your solution (in detail) at the seminar.

# 2 Level: Good (G):

The goal of the tasks at level Good (G) is to understand how to perform liveness analysis in a real compiler setting.

## 2.1 Liveness Analysis

Your task is to extend your compiler to support liveness analysis at the assembly instruction level. You may solve the problem either in two steps (solve it first for the CFG and then for each basic block), or to generate a large graph of all instructions. The former may be more efficient, but the latter may be easier to implement.

If you run your compiler with flag `--liveness`, the compiler should output a pretty-printed graph that includes live-in and live-out information of each variable. For level G, it is enough if the compiler generates liveness information about virtual registers (variables). You do not have to consider pre-colored nodes, although it is OK to include them in the analysis.

We do not enforce a standard format when pretty printing using flag `--liveness`. The actual examination of your implementation will be done at the seminar. However, a format somewhat similar to the following example is desirable.

For instance, suppose we are compiling the following Cigrid program:

```
int main(){
  int x = 0;
  int y = 1;
  while(x < 5){
    x++;
    y = y * x;
  }
  return y;
}
```

If the assembly code is printed before register allocation (or spilling), the output may look like something similar to this:

```
        global      main
        section     .text
main:
        mov         x_1, 0          ;_inst3
        mov         y_2, 1          ;_inst4
        jmp         _while_guard0   ;_inst5

_while_guard0:
        cmp         x_1, 5          ;_inst7
        jl          _while_body1
        jmp         _after_while2   ;_inst6 (two instructions)

_while_body1:
        add         x_1, 1          ;_inst8
        mov         tmp_3, y_2      ;_inst13
        mov         tmp_4, x_1      ;_inst12
        mov         rax, tmp_3      ;_inst11
        imul        tmp_4           ;_inst10
        mov         y_2, rax        ;_inst9
        jmp         _while_guard0   ;_inst14

_after_while2:
        mov         rax, y_2        ;_inst15
        ret                         ;_inst16
```

If the compiler is executed using program argument `--liveness`, the compiler may print the following set of nodes (or something similar):

```
Node(main, succ = {_inst3}, def = {}, use = {}, live-in = {}, live-out = {})
Node(_inst3, succ = {_inst4}, def = {vreg1}, use = {}, live-in = {}, live-out = {vreg1})
Node(_inst4, succ = {_inst5}, def = {vreg2}, use = {}, live-in = {vreg1}, live-out = {vreg1, vreg2})
Node(_inst5, succ = {_inst7}, def = {}, use = {}, live-in = {vreg1, vreg2}, live-out = {vreg1, vreg2})
Node(_inst7, succ = {_inst6}, def = {}, use = {vreg1}, live-in = {vreg1, vreg2}, live-out = {vreg1, vreg2})
Node(_inst6, succ = {_inst8, _inst15}, def = {}, use = {}, live-in = {vreg1, vreg2}, live-out = {vreg1, vreg2})
Node(_inst8, succ = {_inst13}, def = {vreg1}, use = {vreg1}, live-in = {vreg1, vreg2}, live-out = {vreg1, vreg2})
Node(_inst13, succ = {_inst12}, def = {vreg3}, use = {vreg2}, live-in = {vreg1, vreg2}, live-out = {vreg1, vreg3})
Node(_inst12, succ = {_inst11}, def = {vreg4}, use = {vreg1}, live-in = {vreg1, vreg3}, live-out = {vreg1, vreg3, vreg4})
Node(_inst11, succ = {_inst10}, def = {}, use = {vreg3}, live-in = {vreg1, vreg3, vreg4}, live-out = {vreg1, vreg4})
Node(_inst10, succ = {_inst9}, def = {}, use = {vreg4}, live-in = {vreg1, vreg4}, live-out = {vreg1})
Node(_inst9, succ = {_inst14}, def = {vreg2}, use = {}, live-in = {vreg1}, live-out = {vreg1, vreg2})
Node(_inst14, succ = {_inst7}, def = {}, use = {}, live-in = {vreg1, vreg2}, live-out = {vreg1, vreg2})
Node(_inst15, succ = {_inst16}, def = {}, use = {vreg2}, live-in = {vreg2}, live-out = {})
Node(_inst16, succ = {}, def = {}, use = {}, live-in = {}, live-out = {})
```

Here, the definition of "def" set follows the slide of lecture10

A few comments about the above example and the exercise:

- Note how we refer to the different instructions using names, and that each node contains successor nodes, definition nodes, use nodes, live-in, and live-out.
- Each of the nodes in the example represents one instruction, with two exception: (i) the start of the function has its own node, enabling live-in data to be encoded, and (ii) branching to two blocks is using two assembly instructions, but only one node (e.g. `jl` together with `jmp` in the example). inst6

- Note the mapping between virtual registers in the assembly code and in the graph: in the assembly code we keep the name related to the C code (e.g., `x_1`) and write out `tmp` when the instruction selector created a temporary virtual register (e.g., `temp_3`). In the graph, all virtual registers are called `vreg`, followed by a number. These numbers match the numbers in the assembly code. For instance, `vreg3` corresponds to `tmp_3`, and `vreg1` corresponds to `x_1`.
- You do not have to follow this style exactly, but it needs to be readable, so that you can explain and run different examples at the seminar.
- You may define your own data structures for handling graphs. You are also allowed to use standard graph libraries, but you must implement the liveness analysis yourself.

After that you have implemented the compiler extension, you should create a short document (not handwritten) where you explain the main ideas of how you implemented your solution. Please include a discussion about both strengths and weakness of your solution. The document should be a half to one page long. Save the document as a file `liveness.pdf` inside your cigrid folder, i.e., `/solutions/cigrid/liveness.pdf`.

**Grading tag**

Use the following tag when submitting this exercise: `#module3-cigrid-s-g`. It will only check that there exists a file `liveness.pdf` in the Cigrid folder, and that the program argument flag `--liveness` is valid and produces some output. The actual examination takes place at the seminar.

# 3 Level: Very Good (VG):

The goal of this task is to get complete understanding of implementing a register allocator using graph coloring.

## 3.1 Interference Graph and Graph Coloring

Your first task is to make use of the liveness analysis implemented at level G, and construct an interference graph. The interference graph is then used when performing the register allocation using graph coloring.

A valid solution must contain the following:

- an interference graph implementation. If you give argument `--interference` to your compiler, it should print out the interference graph in a readable format. There is no specified format, but you must be able to explain how to read it.
- The register allocator must use the heuristic for graph coloring as described in the lecture, or something similar. If program argument `--regalloc` is used, the compiler should perform register allocation using graph coloring, instead of spilling all variables, as done in module 2. If register allocation is selected, it clearly needs to use registers to some extent, compared to if the flag `--regalloc` is not used.

The following aspects are optional and may be implemented:

- You may use pre-coloring and include physical registers during liveness analysis and register allocation (see the video tutorial). However, you do not have to use pre-coloring. As an alternative, you may perform the register allocation using just a few safe registers, not involving special registers such as `rax`, registers for parameter handling etc. This will be less efficient, but an acceptable solution.

- You may extend the register allocator to also use coalescing, but you do not have to. Such a register allocated is more complicated, but also more efficient.

Your solution needs to be implemented. The test system will test that your submission can use program argument `--regalloc` and when it is used, that it gives correct results when executing a small benchmark. Note that all flags presented in this module should be possible to use individually, or together. If used together, the actual output may change. For instance, if `--regalloc` is used together with `--interference` you may print out several interference graphs (if your register allocator is using several iterations).

Besides the implementation, you should also create a document (a half to one page), where you explain your solution. This document should not be handwritten. Note that you must not repeat what you described for the G level. This part should focus on the interference graph and the graph-color-based register allocator. In particular, you should discuss how you handled callee-saved registers, special registers in certain instructions (e.g. `rax` and `rdx` in `imul`), and registers used for parameters and arguments. Also, discuss limitations with your approach and how you would extend it if you had more time to do so. Save the document as a file `regalloc.pdf` inside your cigrid folder, i.e., `/solutions/cigrid/regalloc.pdf`.

**Grading tag**

Use the following tag when submitting this exercise: `#module3-cigrid-s-g-vg`