# CS 4240 Compilers & Interpreters Bonus Project Design Document

We implemented the Chaitin-Briggs algorithm for global register allocation to optimize the number of loads/stores emitted by the compiler, severely improving the performance of the program on a real processor. Here is the high-level design and implementation of our algorithm for register allocation.

# Implementation

In order to perform the algorithm, our implementation followed these steps.

1. Build a CFG from the provided IR
2. Allocate IRVariableOperands to virtual registers
3. Perform liveness analysis on the entire CFG
    a. Construct `def` and `use` sets for each basic block
    b. Derive `live_in` and `live_out` sets for each basic block
    c. Iteratively recompute the `live_in` and `live_out` sets until they stabilize
4. Construct a liveness interference graph for which registers are “alive” at the same time
5. Construct a spill stack using the liveness graph
    a. Push registers onto the spill stack and remove them from the interference graph
    b. If there is no node with less than K neighbors where K is the # of physical registers, then just remove it and mark it as spilled
6. Perform graph coloring to assign virtual registers to physical registers
    a. Pop virtual registers off the stack, one at a time.
    b. For each register, check the original graph and assign it a physical register so that it doesn’t share the same color as its neighbors
7. Naively allocate the remaining unallocated virtual registers
    a. If there aren’t enough physical registers left, spill the node to the stack and assign it one of the last 2 temporary registers ($t8 and $t9) which are reserved for spilling. This avoids having to swap multiple registers onto/off of the stack.
8. Implement calling convention
    a. Unlike for naive or greedy intra-block allocation where loads and stores were done at the start/end of each block, when there are cross-function calls, we have no guarantee that the values in the registers will be preserved so we should save the minimal set of registers
    b. To decide the minimal set of registers we computed which registers are written to preceding the function call and which registers are read from after the function call. (This includes registers in the live_in and live_out sets). In the evaluated cases, this often resulted in only 1 to 3 registers being saved to the stack.

This approach for register allocation gave us a 50 to 80% improvement in the number of loads emitted by the compiler which is significant for a program of the given size.

## Building and Usage

- Building: run `$ ./build.sh` in the root project directory.
- Usage: run
  `$ ./run.sh <Input IR file> <Output MIPS file> <Reg Alloc Mode>`
  in the root project directory. <Reg Alloc Mode> can either be
  - --opt or -o for the optimal Chaitin-Briggs register allocation
  - --naive for the naive baseline register allocation

Our chosen IR program is an implementation of the pow function that computes exponents (e.g. a^b). Our program computes exponents by naively multiplying the same number `a` `b` times. We have provided a custom test input of a = 5, b = 10, such that the expected output is 5^10 = 9765625. When testing this program, we obtain the desired output.

The number of loads for the naive implementation is 19 whereas for the optimized implementation, there are only 6 loads, which is a 68% improvement in the static frequency, not to mention the dynamic frequency of the loads in the program which is even more significant.