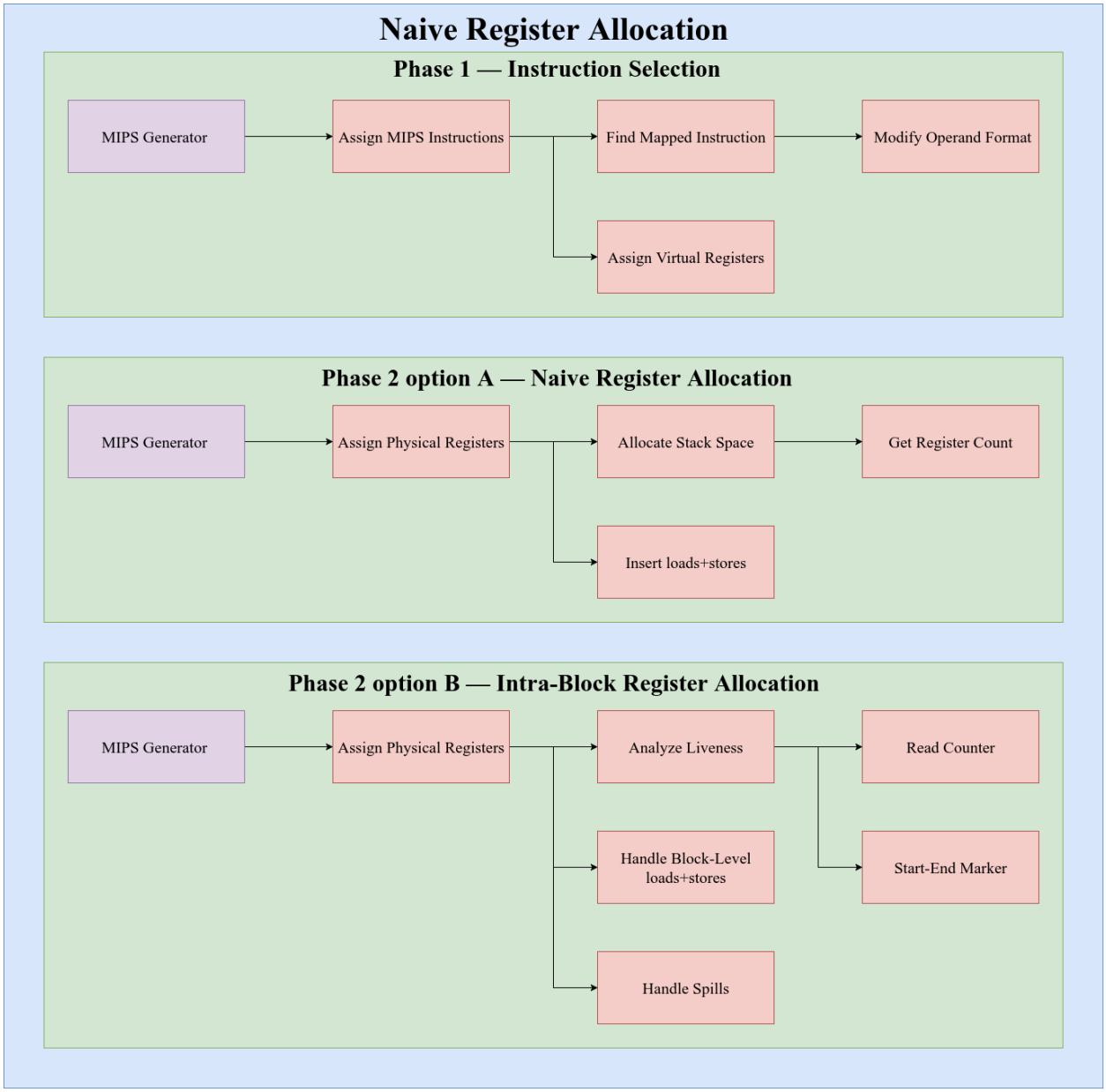
CS 4240 Compilers and Interpreters Project 2 Design Doc

High-Level Design (HLD):



- MIPS Generator is marked purple to highlight that it is repeated and that it is the orchestrator of the entire backend compilation algorithm.
- Implemented algorithms: Simple Instruction Selection, Naive Register Allocation, and Intra-Block Register Allocation with Liveness Analysis.
- Reasons for choosing the approach in the diagram: elegant and modular design.

    - Each arrow indicates what calls what (e.g. Assign Physical Registers calls Analyze Liveness, which calls Read Counter).
    - Each block only does one thing and relies on function calls to accomplish more low level tasks for it. For instance, the Assign Physical Registers box does not care about the nitty gritty details of how the liveness analysis was performed. It only cares that the call to Analyze Liveness provides a valid liveness graph.

Low-Level Design Decisions

- Instruction Selection is done on a function-by-function basis.
    - There is no need to have inter-function instruction selection, especially for such a simple instruction selection algorithm. A case can be made that even in more advanced compilers, each function should be entirely compilable by itself given valid function calls.
- Each IR write to a variable results in a new virtual register and the liveness analysis is performed on virtual registers.
    - Since writing to a virtual register would end its liveness, we just assign a new virtual register to make keeping track of liveness easier during register allocation.
    - We perform the liveness analysis on virtual registers instead of variables because of the above bullet point. If a variable is written to, as far as we are concerned its old state is dead. Since our virtual registers reflect this, we perform our liveness analysis on our virtual registers.
- We used the same basic block identification algorithm from Project 1 and put the resulting MIPS instructions in the same basic block.
    - The only exception to this is callr where the stream of MIPS instructions it maps to should be technically in two different blocks since we need to move $v0 to a virtual register to prevent it from getting overwritten after another function call. However, since this is only a simple move instruction, we pretend as though it is in the same basic block as the rest of the MIPS instructions that result from callr. This does not affect our intra-block register allocation in any way since the lifespan of all physical registers other than $v0 and the register it is moved to end before the function call.
- If in a basic block a virtual register is read without being initialized, it is alive for that line but if it is written to without being read first, it is alive only starting with the next line.
    - This is because a register being read without being written to first in a basic block would need to be loaded from the stack and would thus need to be alive on the line it is being read. But the virtual register that is being written to does not need to be alive on that line, which means it can potentially take on the physical register of one of its arguments.

Software Engineering Challenges

- VS Code + Intellij Usage
    - My partner uses Intellij and is very familiar with it. I use VS Code and did not have too much experience using Java on it. This led to my partner doing most of the debugging while I did other work such as theorizing or writing this doc. We often pair programmed too. Eventually I did what I should have done long ago and took 30 minutes or so to learn how to debug java projects in VS Code. After that, we were both able to debug. This was definitely our #1 software engineering challenge.
- Refactoring/Method Organizations
    - Because of the complex nature of the project and the interdependence of logic such as register allocation and instruction selection (who adds the load/stores on the stack? Does instruction selection create a new virtual register for each write?), we had a challenging time organizing the software. One of us even refactored one class (the Register class) 7 times!

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.

Test Result Summary

- We created a test case that wrote to 26 registers and then used them all in a function call (see public_test_cases/ours/bloated.ir) so that they would be alive the entire time. This test case allowed us to do the following:
    - Clearly see the difference between naive and intra-block allocation
    - Test spilling with intra-block allocation
    - Test liveness overlap
    - Test more than 4 arguments
    - Test array arguments
    - We uncovered many bugs thanks to this test case such as incorrect spillage/instruction printing when using greedy intra-block register allocation
    - It now matches what one would roughly expect by eyeballing the IR
- We also tested using quicksort and prime with SPIM.
    - quicksort in conjunction with SPIM was particularly useful for uncovering our final bug where we were performing an incorrect sbrk call when allocating space for arrays (we forgot to multiply by 4).