**Part 1: Setting up environment and scanning a .i file. (Day 1, Thursday)**

1. Get skeleton code for the program. Includes .c, .h, and makefile.
2. Create a main() function that takes the appropriate arguments. Prints out the number parameter, and the contents of the file line-by-line plus the line number. Be kind and rewind()!
3. Print out the contents of the file while ignoring blank lines and the comments, in main(). Line number should ignore comments/blank lines. Be kind and rewind()!
4. Define enums for each of the ILOC operations present in the test/report blocks in the .h file.
5. Set up logic in main() that sets an enum variable depending on the ILOC operation present on a valid (non-blank/commented) line. We'll have 3 register variables for each line, which we'll use to store up to 3 register ID's from the operation. Have a switch branch after that which prints out the involved virtual register ID's and the operation name, for each valid line.
   1. fprintf to stdout. This means that we'll get print messages, but someone deciding to pipe stdout to a file can also get it. See https://stackoverflow.com/questions/16430108/what-does-it-mean-to-write-to-stdout-in-c .
   2. ILOC is very well-formed, so given the first word, we can figure out the format of the whole thing. For a register, search until an r is hit... and so on.
   3. Account for whitespace ("add r25, r12 => r27" is just as legal as "add r1,r2=>r3", and so on). Just as a reminder.
6. Save this skeleton code (or keep it in a comment block), as we'll be using it to handle file-scanning operations in all three of the methods.

**Part 2: The supporting data structures and generally-applicable related functions (Day 1, Thursday)**

Note: the virtual register structs are defined with information that is useful for both the top-down allocators and the bottom-down allocator. A list of occurrences is useful for top-down for obvious reasons (counting number of occurrences) and not-so-obvious reasons (determining MAXLIVE and live ranges), while for bottom-down we use them to determine "next" occurrence of a register. All of the information is necessary, and it's all the information that is necessary (pending realizations).

1. Define the registerNode struct, typedef'ing a pointer to it as a regNode (typedef registerNode \*regNode). It will contain the following members:
   1. uint regID. This is the "number" of the virtual register in the input program. For example, for "r3", regID would be 3.
   2. An enum indicating whether the register is currently stored in a physical register, or in memory. regStatus status.
   3. A pointer to a linked list of integers, indicating the list of occurrences of this register by line in the file (not including blank lines/registers, as mentioned in Part 1).
   4. uint physID. Indicates the ID of the current physical register, if at all applicable.
   5. uint offset. Indicates the current memory offset where this register is stored, if at all applicable.
   6. regNode next. A pointer to another registerNode struct, as regNodes will be stored in a linked list.
2. Define the intNode (do so as a struct pointer like regNode).
   1. int val. Indicates the value this node stores.
   2. intNode next. Indicates the next intNode in the linked list.
3. Make constructor for a regNode given the needed input parameters. Every regNode assumes storage in a physical register by default, which makes sense by ILOC standards.
4. Make constructor for intNode given an input int.
5. Make a function that can scan through a file line-by-line (using our aforementioned file-scanning skeleton) that will populate and return a linked list of regNodes with the corresponding information on each line (for the first pass, just a linked list of occurrences). Be kind and rewind()!
   1. The other information besides occurrences will be filled in either at the beginning (top-down) of logic, or as we go along (bottom-down).
6. Make a function that can free a list of regNodes and the intNodes stored in each one.

**Part 3: First top-down allocator (Friday). EAC rules: no MAXLIVE heuristic, just go by number of occurrences.**

Note: we're doing the textbook version first, because it only counts the number of occurrences. The second version (that spills on MAXLIVE >= k instructions, and uses live range as a tie-breaker) will be implemented afterwards, because it draws from techniques used here.

***SPECIAL NOTE: Don't spill r0 in top-down or bottom-down! It is present at all times in its own register, the machine register. Keep in mind that PHYSICAL REGISTERS r1 and r2 are physical registers, but VIRTUAL registers r1 and r2 are in a different position. Begin labeling allocatable registers from r3 and on, when k > 2 (allocate r3 for k == 3, and so on).***

1. Define a function that takes an input file and linked list of regNodes (having been populated with number of occurrences by the support function in Part 2) and determines which of the nodes are spilled and which are in physical registers. We know what our feasible (r1, r2) and allocatable (r3, ...) registers are (if any) depending on arguments passed in from main(). The status/location members of each regNode are changed accordingly. This is really the control logic of a top-down allocator.
   1. I plan to create an array that stores pointers to the regNode structs, and then sort it in descending order of number of occurrences. Write a comparator function that can be plugged into qsort(), which can be used to sort an array in descending order. It will return a negative number if the first input has a greater number of occurrences, 0 if equal, and positive if fewer.
   2. Create a function that takes the regNode list head as input, and returns an array of regNodes (remember, these are pointers) sorted in descending order of the number of occurrences they have. It will use the comparator described above. This will be used in both top-down phases.
   3. Once we have the regNodes in descending order of number of occurrences, we assign them to allocatable physical registers (if any) until we run out. Then we spill everything following the last allocated virtual register (if any) into decreasing offsets (starting at -4).
2. In main(), perform the aforementioned operations that get us to having a populated, top-down allocated regNode list, making use of the arguments for number of registers (Section 2) and the filename (converted to a file pointer that's passed into the support functions). With that in mind, pass the information to a function specifically for simple top-down allocation.
3. Then parse through the file line-by-line using the skeleton (nextNum() calls and so on) and rewrite the operations, performing spill operations as needed (search for a virtual register in the line and see if we need to spill it back to a feasible register, or spill a value from a feasible register to a virtual register, etc). Use the status member of a regNode, offset, and physId accordingly.

**Part 4: Second top-down allocator (Friday to Saturday). Lecture rules: MAXLIVE spilling with live range as a tie-breaker.**

**Part 5: Bottom-down allocator (Saturday to Sunday). Simplest one possible.**