COMP 412, Fall 2017
Lab 1: Local Register Allocation

Please report suspected typographical errors to the class Piazza site. We will issue corrections as needed.

Code	Due date:	9/15/2016	**Tutorial #1**	Date:	8/38/2016
	Submit to:	comp412code@rice.edu		Time:	5:00 PM
Code Checks	Check #1 due:	9/1/2016		Location:	SH 301
	Check #2 due:	9/8/2016	**Tutorial #2**	Date:	TBA
Report & Test Block	Due date:	4 days after code submission		Time:	5:00 PM
	Submit to:	comp412report@rice.edu		Location:	TBA

Table of Contents

Code

C-1. Local Register Allocation

You will build a local register allocator—that is, a program that takes as input a branch-free sequence of operations and produces as output an equivalent sequence of operations that uses a specified number of registers. Specifically, your allocator will take as input a file that contains a sequence of operations, expressed in a subset of the ILOC Intermediate Representation (IR) presented in *EaC2e*. The ILOC subset used in Lab 1 is described in § A-1. As output, your allocator will produce an "equivalent" sequence of ILOC operations.

> *For the purposes of this lab, an input program and an output program are considered* ***equivalent*** *if and only if they print the same values in the same order to* `stdout` *and each data-memory location defined by an execution of the input program receives the same value when the output program executes. It is expected that the output program will define some additional data memory locations that are not defined by the input program—locations that hold spilled values.*

To simplify matters, the test blocks do not read any input files. All data used by a given test block is either contained in the test block or entered on the command line using the ILOC simulator's `–i` option. (See § A-1 for information about the ILOC simulator.)

Your lab should perform register allocation, but no other optimization. Because all input data is encoded directly in some of the test blocks, your lab could, in theory, execute those blocks and replace them with a series of output statements and writes to memory. *Such optimization is forbidden and will result in a significant loss of points.* All `loads`, `stores`, and arithmetic operations that appear in the input program must appear in the output program. The output block must perform the original computation and cannot pre-compute the results. Allocators that violate these rules will lose substantial credit. Of course, the output block may use different register names than the input block.

Your allocator **may** remove `nops`; they have no effect on the equivalence of the input and output of programs. If your allocator performs *rematerialization* (see the Lab 1 Performance Lecture), then it may move `loadI` operations around in ways that cannot happen with a `load` or a `store`. Finally, your allocator must not perform optimizations based on the input constants specified in ILOC test block comments; when grading your allocator, the TAs may use other inputs.

Design and Implementation Timeline:

To produce a reasonably good allocator, you need to start your implementation during the first week.

— By September 1, 2017, you must pass code check #1. Code check #1 tests your allocator's front end, back end, and register renaming pass—that is, the code needed to read in a large ILOC program, to store it, to rename the registers, and to print the ILOC program stored in your data structures. Code check #1 also tests whether or not the renamed code generated by your allocator produces correct answers when executed using the ILOC simulator.

— By September 8, 2017, you must pass code check #2. Code check #2 tests your allocator's ability to perform register allocation for $3 \le k \le 64$ and whether or not the code generated by your allocator produces correct answers when executed using the ILOC simulator. Code check #2 also tests whether or not your allocator conforms to the user interface specifications described in § C-2.

— During the remaining period before Lab 1 is due, you should focus on improving the quality of the code that your allocator produces, your allocator's robustness, and its results on the timing tests.

The code checks are not exhaustive tests. You should test your allocator on a wider set of ILOC input programs.

C-2. Code Specifications

The COMP 412 teaching assistants (TAs) will use a script to grade Lab 1 code submissions. All grading will be performed on CLEAR, so test your allocator as well as your `makefile` and/or script on CLEAR. You must adhere to the following interface specifications to ensure that your allocator works correctly with the script that the TAs will use to grade your allocator.

- **Name:** The executable version of your allocator must be named `412alloc`.
- **Behavior:** Your allocator must work in three distinct modes, as controlled by parameters and flags on the command line. Your allocator must check the correctness of the command-line arguments. The output described in the table below must be printed to the standard output stream `stdout`; error messages must be printed to the standard error output stream `stderr`. The TAs will test each of these modes, using command lines that fit the syntax described below:

`412alloc –h`	When a `–h` flag is detected, `412alloc` must produce a list of valid command-line arguments that includes a description of all command-line arguments required for Lab 1 as well as any additional command-line arguments supported by your `412alloc` implementation. `412alloc` is not required to process command-line arguments that appear after the `–h` flag.
`412alloc –x` <file name>	The `–x` flag will be used for the Lab 1 code check. When a `–x` flag is detected, `412alloc` must perform register renaming, not register allocation, on the input block contained in <file name>. <file name> specifies the name of the input file. <file name> is a valid Linux pathname relative to the current working directory. <file name> must appear after the `–x` flag and is the only valid command-line argument that can appear after the `–x` flag. `412alloc` will produce, as output, an ILOC program that is *equivalent* to the input program. In the output program, each register name will be defined exactly once. `412alloc` is not required to process command-line arguments that appear after <file name>.
`412alloc` *k* <file name>	This command will be used to invoke register allocation for *k* registers on the input block contained in <file name>. *k* is an integer that specifies the number of registers, starting from `r0`, that `412alloc` should assume are available on the target machine when performing register allocation. For Lab 1, $3 \leq k \leq 64$. <file name> specifies the name of the input file. <file name> is a valid Linux pathname relative to the current working directory. <file name> must appear after *k* and is the only valid command-line argument that can appear after *k*. `412alloc` will produce, as output, an ILOC program that is *equivalent* to the input program. The output program will use only registers numbered from zero to *k*-1. Your allocator must produce a reasonable and informative message if *k* is anything other than an integer in the allowable range ($3 \leq k \leq 64$). `412alloc` is not required to process command-line arguments that appear after <file name>.

- **Input:** The input file specified in the command-line will contain a single basic block that consists of a sequence of operations written in the ILOC subset described in § A-1. If your allocator cannot read the input file, or the code in the file is not valid ILOC, your allocator should produce an informative message. If the ILOC code in the input file uses a value from a register that has no prior definition, your allocator should handle the situation gracefully.

  Scanning the Input: You must write your own code to scan the input. You may not use a regular expression library or other pattern-matching software, unless you write it yourself. Your allocator may read an entire line of input and scan that line character-by-character.

  Note: The timer described in § A-3.2 uses input files containing up to 128,000 lines of ILOC. Your allocator is expected to handle such large files correctly, efficiently, and gracefully. The use of efficient algorithms and data structures in your allocator is key to handling large input files.

- **Makefile & Shell Script:** Lab 1 submissions written in languages that require a compilation step, such as C, C++, or Java, must include a `makefile`.[1] (Lab 1 submissions written in languages that do not require compilation, such as Python, do not need a `makefile`.)

  The `makefile` must produce an executable named `412alloc`. Lab 1 submissions written in languages in which it is not possible to create an executable and rename it `412alloc`, such as Python and Java, must include an executable shell script named `412alloc` that correctly accepts the required command-line arguments and invokes the program. For example, a project written in Python named `lab1.py` could provide an executable shell script named `412alloc` that includes the following instructions:

  ```
  #!/bin/bash
  python lab1.py $@
  ```

  A project written in Java with a jar file named `lab1.jar` or a class named `lab1.class` that contains the main function could provide, respectively, one of the following two executable shell scripts named `412alloc`:

  ```
  #!/bin/bash                    #!/bin/bash
  java –jar lab1.jar $@          java lab1 $@
  ```

  To ensure that your `412alloc` shell script is executable on a Linux system, execute the following command in the CLEAR directory where your `412alloc` shell script resides:

  ```
  chmod a+x 412alloc
  ```

  To avoid problems related to the translation of carriage return and line feed between Windows and Linux, we recommend that you write your shell script on CLEAR rather than writing it on a Windows laptop and transferring the file.

- **Grading Script:** The TA's grading script will first execute a `make` if a `makefile` is present. The grading script will then execute commands similar to the following command:

  ```
  ./412alloc  12  test_block_17.i  > output_block.i
  ```

  which should invoke your allocator on the file `test_block_17.i` with the number of registers set to `12`. The output of your allocator, which is redirected to `output_block.i` by the command, will be tested on CLEAR using the Lab 1 simulator. Use the timer described in § A-3.2 to test your allocator's adherence to the interface used by the grading script.

- **CLEAR:** Your code and `makefile`/shell script must compile, run, and produce correct output on Rice's CLEAR systems. CLEAR provides a fairly standard Linux environment.

[1] If you prefer to use a build manager that is available on CLEAR to create your executable, you may invoke that build manager in your `makefile`.

- **Programming Language:** You may use any programming language available on Rice's CLEAR facility, except for Perl. Your goal should be to use a language that is available on CLEAR, in which you are comfortable programming, for which you have decent debugging tools, and that allows you to easily reuse part of your Lab 1 code in Lab 3. When coding, be sure to target the version of your chosen programming language that is available on CLEAR**.**
- **README:** You are required to submit a `README` file that provides directions for building and invoking your program. Include a description of all command-line arguments required for Lab 1 as well as any additional command-line arguments that your allocator supports. To work with the grading scripts, the first two lines in your `README` file must be in the following form. (Do not include whitespace after "`//`".)

```
//NAME: <your name>
//NETID: <your NetID>
```

Assistance with Lab 1: The COMP 412 staff is available to answer questions:

Piazza: Post questions to Piazza, where COMP 412 students, TAs, and professors respond to questions.
Tutorials: There will be two tutorials for Lab 1, with time and place announced in class and on Piazza. The tutorials will offer advice, clarifications, and implementation suggestions. Attend the tutorials.
Office Hours: Visit the TAs during the hours posted on the Piazza COMP 412 course page under "Staff".

C-3. Code Checks

The Lab 1 code checks must be run on CLEAR, so test your code on CLEAR prior to the deadlines. See § C-5 for details related to the Lab 1 grading rubric, late penalties, and honor code policy.

Code Check #1: Code check #1 is due at 11:59 PM on Friday, September 1, 2017. To pass code check #1, your code must correctly scan, parse, perform register renaming, and print the resulting renamed ILOC block to `stdout`. The code check #1 script and test blocks are located on CLEAR in the `/clear/courses/comp412/students/lab1/code_check_1/` directory.

Code Check #2: Code check #2 is due at 11:59 PM on Friday, September 8, 2017. To pass the code check, your code must conform to the user interface described in § C-2 and must perform register allocation correctly and print the resulting allocated ILOC block to `stdout`. The CLEAR directory `/clear/courses/comp412/students/lab1/code_check_2/` contains the code check #2 script and test blocks.

Each code check directory on CLEAR contains a `README` file that describes how to invoke the code check script and interpret the results. Note that code written in languages in which it is not possible to create a named executable, such as Python and Java, will need an executable shell script that accepts the required command-line arguments and invokes the program. A working `makefile` is not critical for the code checks, but we recommend that you create your `makefile` before the first code check and use it while developing your code. See "Makefile & Shell Script" in § C-2 for details.

Advice: If you have never produced a `makefile` and/or shell script, develop your `makefile`/shell script well in advance of the Lab 1 code check #1 deadline. Seek TA assistance if you encounter difficulties creating your `makefile`/shell script or accessing CLEAR. If you wait until the day that code check #1 is due to seek assistance, the TAs may not have time to help you due to the logistics of handling code check #1 submissions.

Submitting Code Checks: The COMP 412 teaching assistants (TAs) will post on Piazza directions for submitting Lab 1 code checks.

C-4. Code Submission Requirements

Due Date: Lab 1 code is due at 11:59 PM on the code due date (Friday, September 15, 2016). Individual extensions to this deadline will not be granted.

Early-Submission Bonus: Code received before the code due date will be awarded an additional three points per day up to a maximum of nine points. For example, to receive nine points, you must submit your code by 11:59 PM on Tuesday, September 12, 2016.

Late Penalty: Lab 1 code received after the code due date will lose three points per day. Late code will be automatically accepted until 11:59 PM on Friday, September 22, 2016. Permission from the instructors is required to submit Lab 1 code after this deadline. Contact both instructors by e-mail to request permission.

Late Penalty Waivers: To cover potential illness, travel, and other conflicts, the instructors will waive up to six days of late penalties per semester when computing your final COMP 412 grade.

Submission Details: Create a `tar` file that contains your submission, including (1) the source code for your allocator, (2) your `makefile` and/or shell script, (3) your `README` file, and (4) any other files that are needed for the TAs to build and test your code.

If you not created a `tar` file previously, you should learn how to create one before the code submission deadline. You do not want to be reading the `man page` ten minutes before the due date. Note that a `tar` file is different than a `zip` archive. You must submit a `tar` file.

If your allocator does not work, include in your `tar` file a file named `STATUS` that (1) contains a brief description of the current state of the passes in your allocator (complete/incomplete/not implemented, working/broken, etc.) and (2) states which code checks your submitted implementation passes.

Name the `tar` file with your Rice `NetID` (e.g., `jed12.tar` for a student with the Rice `NetID` `jed12`). Email the `tar` file as an attachment to `comp412code@rice.edu` before 11:59 PM on the code due date. Use "Lab 1 Code" as the subject line of your e-mail submission.

Note: `comp412code@rice.edu` should only be used for submitting code since it is only monitored during periods when code is due. Questions should be posted to the course discussion site on Piazza.

C-5. Code Grading Rubric & Honor Code Policy

The Lab 1 code grade accounts for 14% of your final COMP 412 grade. The Lab 1 code rubric is based on 100 points, which will be allocated as follows. TAs will award more points to allocators that both produce correct output code and insert fewer spills, based on the output and cycle count reported by the ILOC simulator.

- **10 points** for passing code check #1 by 11:59 PM Friday, September 1, 2017.
  Late Policy: 5 points will be awarded for passing code check #1 by 11:59 PM on Tuesday, September 5, 2017. No points for code check #1 will be awarded after September 5, 2017.
- **10 points** for passing code check #2 by 11:59 PM on Friday, September 8, 2017.
  Late Policy: 5 points will be awarded for passing code check #2 by 11:59 PM on Monday, September 11, 2017. No points for code check #2 will be awarded after September 11, 2017.
- **5 points** for adherence to the Lab 1 code specifications and submission requirements.
- **75 points** for a combination of the correctness of the code produced by your allocator and the number of cycles required to run the allocated ILOC code produced by your allocator on the Lab1 simulator. (The speed of your allocator, as measured on the timing blocks (see § A-3.2), factors into the lab report grade, not the code grade.) Partial credit for allocators that do not produce correct code will be awarded at the discretion of the instructors.

Your submitted allocator source code and `README` file must consist of code and/or text that you wrote, <u>not edited or copied versions of code and/or text written by others or in collaboration with others</u>. You may not look at COMP 412 code from past semesters. You may not invoke the COMP 412 reference allocator, or any other allocator that you did not write, from your submitted code.

You are welcome to collaborate with current COMP 412 students when preparing your `makefile` and/or shell script and to submit the results of your collaborative `makefile` and/or shell script efforts. However, as indicated in the previous paragraph, all other Lab 1 code and text submitted must be your own work, not the result of a collaborative effort.

You are welcome to discuss Lab 1 with the COMP 412 staff and with students currently taking COMP 412. You are also encouraged to use the archive of test blocks produced by students in previous semesters. However, you may not make your COMP 412 labs available to students (other than COMP 412 TAs) in any form during or after this semester. In particular, you may not place your code anywhere on the Internet that is viewable by others.

Advice: The most frequent advice provided by past COMP 412 students in their Lab 1 reports is:

- Start Lab 1 on the day that it is assigned.
- Design your allocator, particularly data structures and classes, before beginning your implementation.
- Consider the full scope of the project, not just the first part of the project, during your design phase. Incremental design tends to lead to shortsighted decisions.
- Use a programming language that works on CLEAR, in which you are comfortable programming, for which you have decent debugging tools, and that allows you to easily reuse part of your Lab 1 code in Lab 3. In particular, do not use Lab 1 as an opportunity to learn a new programming language.
- Produce a working, thoroughly tested, version of your allocator before implementing heuristic improvements. (Use the timer as well as a variety of inputs, including the report blocks, when testing.)
- Save a backup copy of your working allocator before experimenting with heuristics to ensure that you have a working allocator to submit when the code is due.
- Before submitting your final allocator, thoroughly test it on CLEAR and ensure that it can correctly allocate all of the blocks required for your Lab 1 report.

Report & Test Block

R-1. Report Contents and Format

Your Lab 1 report will contain: your replies to a brief questionnaire, two tables of data, and a graph. The specifications for the tables and the graph are given in Section R-1.2. The questionnaire should be in twelve-point font. It can be either single-spaced or double-spaced. The tables and graph should each be on a separate page.

Run a spell checker on your questionnaire.

R-1.1. Questionnaire

The Lab 1 Report Questionnaire will be available on the course web site before the Code Check 2 deadline. It will be provided in both a Microsoft Word format (`.docx`) and a plain ASCII file. Run a spelling checker on your questionnaire.

R-1.2. Results

In the results section, you will document your experimental results as well as the effectiveness and efficiency of your allocator. This section consists of two tables and a chart. The Questionnaire will ask specific questions about your results, based on the data in this section.

Table 1: Allocator Effectiveness:

- Table 1 should show, for each of the following scenarios, the number of cycles reported by the ILOC simulator for each Lab 1 report block and your ILOC test block:
  - Use the original (unallocated) ILOC blocks as input to the ILOC simulator. (Do not use the ILOC simulator's `–r` parameter.) *For the report blocks, you may use the "Original ILOC Code" results shown in Table 1.*
  - Perform register allocation for *k* = 3, 4, 5, 6, 8, and 10 on the ILOC blocks using your bottom up allocator. Use the resulting allocated blocks as input to the ILOC simulator. (Use the value of *k* for the ILOC simulator's `–r` parameter.)
  - Perform register allocation for *k* = 3, 4, 5, 6, 8, and 10 on the ILOC blocks using the reference implementation allocator, which is described in § A-3.1. Use the resulting allocated blocks as input to the ILOC simulator. (Use the value of *k* for the ILOC simulator's `–r` parameter.) *For the report blocks, you may use the results for the reference implementation shown in Table 1 in the "lab1_ref" rows.*
  - Indicate in either your table or in the text if (1) your allocator failed on an input block for a particular value of *k*, (2) the ILOC simulator generated an error message when run with the allocated code for a particular input block and value of *k*, or (3) the ILOC simulator generated the wrong output when run with the allocated code.
  - The example shows the information that must be included in your table. The results columns are labeled with the names of the two allocators being compared: `412alloc` and `lab1_ref`. Include the results for your allocator in the "412alloc" column.
  - For *k* = 3, 4, 5, 6, 8, and 10, use the following formula to compare the number of cycles reported by the simulator for blocks allocated by `412alloc` versus the number of cycles reported for blocks allocated by `lab1_ref`. Report the results in the "Difference (percent)" columns.

    100 * (`lab1_ref`'s cycles – `412alloc`'s cycles) / (`lab1_ref`'s cycles)

Table 1: Total Cycles Required for Lab 1 Report Blocks & Submitted Block *

Input Block	Allocator	Units	Available Registers						Original ILOC Code
			k=3	k=4	k=5	k=6	k=8	k=10	
report1.i									
	lab1_ref	cycles	215	161	125	95	65	54	54 cycles
	412alloc	cycles	223	169	133	103	73	61	
	Difference	percent	-3.7%	-5.0%	-6.4%	-8.4%	-12.3%	-13.0%	
report2.i									
	lab1_ref	cycles	171	117	105	99	92	86	56 cycles
	412alloc	cycles	176	120	112	111	104	92	
	Difference	percent	-2.9%	-2.6%	-6.7%	-12.1%	-13.0%	-7.0%	
report3.i									
	lab1_ref	cycles	409	351	312	266	217	179	82 cycles
	412alloc	cycles	388	324	280	242	202	166	
	Difference	percent	5.1%	7.7%	10.3%	9.0%	6.9%	7.3%	
report4.i									
	lab1_ref	cycles	169	162	156	150	138	124	68 cycles
	412alloc	cycles	169	161	156	151	141	129	
	Difference	percent	0.0%	0.6%	0.0%	-0.7%	-2.2%	-4.0%	
report5.i									
	lab1_ref	cycles	84	61	48	36	30	30	30 cycles
	412alloc	cycles	87	67	52	46	38	30	
	Difference	percent	-3.6%	-9.8%	-8.3%	-27.8%	-26.7%	0.0%	
report6.i									
	lab1_ref	cycles	276	243	196	185	166	154	105 cycles
	412alloc	cycles	282	242	209	197	174	160	
	Difference	percent	-2.2%	0.4%	-6.6%	-6.5%	-4.8%	-3.9%	
report7.i									
	lab1_ref	cycles	115	90	73	68	60	46	44 cycles
	412alloc	cycles	123	98	84	82	72	60	
	Difference	percent	-7.0%	-8.9%	-15.1%	-20.6%	-20.0%	-30.4%	
jed12.i									
	lab1_ref	cycles	181	133	106	100	80	68	68 cycles
	412alloc	cycles	189	149	107	96	77	68	
	Difference	percent	-4.4%	-12.0%	-0.9%	4.0%	3.8%	0.0%	

* The simulator runs used to generate the results shown in this table ran without errors and produced correct output.

Table 2: Allocator Efficiency:

- Table 2 should show the results of running the timer described in § A-3.2 on CLEAR with your allocator. Note instances where the timer did not produce results because your allocator either required five minutes or more to allocate a smaller file or failed to allocate one or more blocks.
- The example above shows the information that must be included in your table. "lab1_ref" refers to the reference allocator implementation. "412alloc" refers to the allocator that you submitted for grading. (The data shown for `412alloc` is from an allocator written in Python.)

Table 2: Allocator Timing Results (k=15)*

Input (lines)	Allocation Time (seconds)	
	lab1_ref	412alloc
1000	0.003395	0.089987
2000	0.004204	0.162975
4000	0.005614	0.310953
8000	0.009171	0.597909
16000	0.016065	1.190819
32000	0.029343	2.379638
64000	0.056002	4.781273
128000	0.111982	9.657532

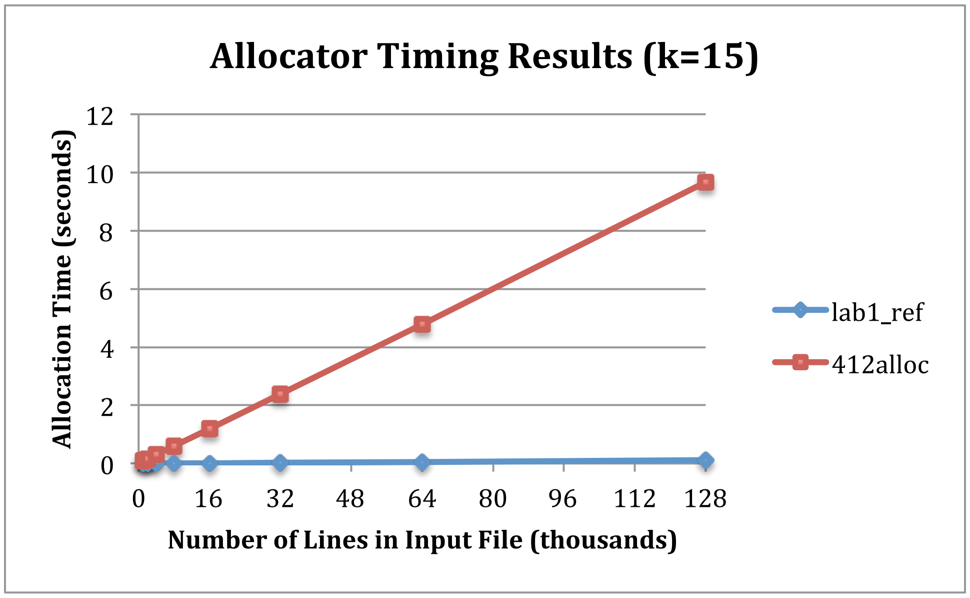
* None of the timing runs generated error messages.

Java Programmers: The results presented in Table 2 and its associated graph must be obtained using CLEAR's default JVM maximum heap size. If garbage collection is slowing down your allocator, you may include supplementary timing data that shows your allocator's efficiency results for larger maximum heap sizes. Document the maximum heap sizes that you use when measuring the supplementary timings.

See the material on Garbage Collection in Java posted on the course web site. It can be found at the bottom of the "Lectures" page, under the heading "Additional Materials."

Graph 1: Timing Results

- Graph your allocator timing results using a format similar to the following graph. To receive full credit, you must use a linear scale (not a logarithmic scale) for **both** axes of your graph.



Note: The base points for the "Register Allocator Efficiency" section of the Lab 1 report are awarded for the inclusion of all requested information and the quality of your responses. The base points are **<u>not</u>** determined by the speed of your register allocator, as shown in Table 2 and the associated graph. Extra credit points may be awarded for `412alloc` implementations that demonstrate exceptional, reproducible efficiency results when compared to `412alloc` implementations written in the same language (C, C++, Java, Python, etc.).

R-2. Test Block

You are required to submit an original, commented ILOC test block that either accurately tests one or more aspects of your allocator, or implements an interesting algorithm or computation. Submitted test blocks may be added to the archive of contributed input blocks. (See § A-2.) Points awarded for your test block will be based on an instructor's evaluation of the quality, correctness, and originality of your ILOC code, and adherence to the following specifications:

- Your test block must be submitted in a file named with your Rice `NetID` (e.g., `jed12.i` for a student with the Rice `NetID` `jed12`). Use the `.i` suffix to indicate that the file contains ILOC.
- Your test block must produce output (i.e., contain <u>at least</u> one ILOC `output` statement) that can be used to convincingly determine whether or not the ILOC code executed correctly.
- The instructors will assess the correctness of your submitted test block with the ILOC test block verification scripts available at `/clear/courses/comp412/students/lab1/scripts`.
- To work with the scripts, the <u>first four lines</u> in your test block must be in the following form:

  ```
  //NAME: <your name>
  //NETID: <your NetID>
  //SIM INPUT: <required simulator input, for example: -i 1024 1 2 3 4 5 6 7 8 9 10>
  //OUTPUT: <expected simulator output, for example: 55 0 13>
  ```

  Do not include whitespace after "`//`". Since the `SIM INPUT` string specifies input flags and constants (not variable names) that will be used when invoking the simulator, it must be in a form accepted by the simulator. The `OUTPUT` string must specify the string of integers (not variable names) that the simulator is expected to produce when invoked with your test block and the string specified by `//SIM INPUT:`. See the blocks described in § A-2 for examples of blocks that have comments that meet this specification.
- Your test block must only use ILOC operations described in § A-1. Each ILOC operation must begin on a new line. All memory accesses that occur when your test block is invoked with the input specified in `//SIM INPUT:` must be word aligned. Additionally, your test block must not use labels, square bracket notation, ILOC pseudo operations, or registers with undefined values.
- The comments that follow the first four lines of comments must include a brief description of:
  - The block's input requirements (via the ILOC simulator's `-i` parameter) and expected output. You can use variables in these comments to describe your block's required input and expected output, as appropriate.
  - Either the algorithm implemented or the aspect of your allocator tested by your test block.

R-3. Report & Test Block Submission Requirements

Your Lab 1 report and test block are due <u>four days after you submit your Lab 1 code</u>. Combine your completed questionnaire, the two tables, and the graph into one file, in either PDF format (`.pdf`) or Microsoft Word format (`.doc` or `.docx`). Use your Rice `NetID` as the basename for the resulting file (e.g., `jed12.pdf`, `jed12.doc`, or `jed12.docx`).

Send your report and test block as attachments to comp412report@rice.edu. Use "Lab 1 Report and Test Block" as the e-mail subject line.

Late Penalty: Late Lab 1 reports and test blocks will lose three points per day. Late reports and test blocks will automatically be accepted until 11:59 PM on Tuesday, September 26, 2016. Permission from the instructors is required to submit late Lab 1 reports and test blocks after this deadline. Contact both instructors by e-mail to request permission.

Late Penalty Waivers: To cover potential illness and conflicts, the instructors will waive up to six days of late penalties per semester when computing your final grade. Priority will be given to waiving penalties for late code over penalties for late lab reports and test blocks.

Note: comp412report@rice.edu should only be used for submitting lab reports, test blocks, and test-block-related bug reports since it is not monitored regularly.

R-4. Report & Test Block Grading Rubric & Honor Code Policy

The Lab 1 report and test block grade accounts for 4% of your final COMP 412 grade. The grading rubric for the Lab 1 report and test block is based on 100 points:

- 25 points for adherence to the test block specifications in § R-2 and the quality, correctness, and originality of the submitted test block.
- 75 points for adherence to the lab report specifications and the grammatical correctness, quality, and content of the resulting lab report.
- Your submitted ILOC test block and Lab 1 report must consist of code and text that you wrote, not edited or copied versions of test blocks or text written by others. The data used to generate the tables and graph required for the report must be produced using the tools and methodology described in § R-1.2.

You may not submit a test block or report that you jointly wrote with another person. You may not look at COMP 412 reports from past semesters. You may not make your COMP 412 reports available to students in any form during or after this semester. In particular, you may not place your lab report anywhere on the Internet where it is viewable by others.

Checklist

The following high-level checklist is provided to help you track progress on your Lab 1 code, report, and test block.

- ☐ Implement a bottom-up local register allocator based on the algorithm in § 13.3.2 (pages 686–689) of *Engineering a Compiler (EAC2e)*, and the discussion in class and the tutorial sessions. Submit your allocator by Friday, September 16, 2016.
- ☐ Use the ILOC simulator, which is described in § A-1, to ensure that the allocated code produced by your program is correct—that is, it produces an equivalent sequence of operations (see § C-1 for a definition of "equivalent")—and to measure the number of cycles that the ILOC simulator requires to execute each allocated block. A variety of ILOC input blocks are available for you to use when testing your allocator. See § A-2 for details.
- ☐ Test your allocator thoroughly on CLEAR. It will be graded on CLEAR.
- ☐ Ensure that your code passes Lab 1 code check #1. To pass the code check, your code must correctly scan, parse, perform register renaming, and print the resulting renamed ILOC block to `stdout`. See § C-3 for details. To receive full credit, submit on or before Friday, September 2, 2016 code check #1 results that demonstrate that your allocator passes code check #1.
- ☐ Ensure that your code passes Lab 1 code check #2. To pass the code check, your code must conform to the user interface described in § C-2, and must perform register allocation correctly and print the resulting allocated ILOC block to `stdout`. See § C-3 for details. To receive full credit, submit on or before Friday, September 9, 2016 code check #2 results that demonstrate that your allocator passes code check #2.
- ☐ Create an original ILOC test block, as described in § R-2. Report the number of cycles required by the ILOC simulator to run both your original test block and the allocated version of your test block, as described in § R-1.2.
- ☐ For each Lab 1 report block, report the number of cycles required by the ILOC simulator to run the block on CLEAR both before and after allocation, as described in § R-1.2.
- ☐ Invoke the timer described in § A-3.2 on CLEAR to generate the data that you are required to include as part of your lab report, as described in § R-1.2. Note that you should run the timing tests long before you submit your final code. These blocks provide a useful test of your algorithms and implementation at scale.
- ☐ Create your lab report, which will consist of a completed questionnaire, two tables, and a graph. Submit the lab report, in one of PDF format or Microsoft Word format, along you're your test block. Specifications for the report and test block are in section R of this document.s

Appendix

A-1. ILOC Simulator & Subset

ILOC Simulator: An ILOC simulator, its source, and documentation are available on CLEAR. The executable simulator is `/clear/courses/comp412/students/lab1/sim`. Its source and documentation are in the `/clear/courses/comp412/students/lab1/simulator` directory. See § 7.1 of the simulator documentation for Lab 1 configuration details.

The simulator builds and executes on CLEAR. You can either run the simulator as `/clear/courses/comp412/students/lab1/sim`, or copy it into your local directory. The simulator appears to work on other OS implementations, but is not guaranteed to work on other OS implementations. Your allocator will be tested and graded on CLEAR, so you should test it on CLEAR.

ILOC Subset: Lab 1 input and output files consist of a single *basic block*[2] of code written in a subset of ILOC. Your allocator must support and restrict itself to the following case-sensitive operations:

Syntax	Meaning	Latency
`load r1 => r2`	`r2 ← MEM(r1)`	3
`loadI x => r2`	`r2 ← x`	1
`store r1 => r2`	`MEM(r2) ← r1`	3
`add r1, r2 => r3`	`r3 ← r1 + r2`	1
`sub r1, r2 => r3`	`r3 ← r1 - r2`	1
`mult r1, r2 => r3`	`r3 ← r1 * r2`	1
`lshift r1, r2 => r3`	`r3 ← r1 << r2`	1
`rshift r1, r2 => r3`	`r3 ← r1 >> r2`	1
`output x`	prints `MEM(x)` to `stdout`	1
`nop`	idle for one cycle	1

All register names have an initial lowercase `r` followed immediately by a non-negative integer. Leading zeros in the register name are not significant; thus, `r017` and `r17` refer to the same register. Arguments that do not begin with `r`, which appear as `x` in the table above, are assumed to be non-negative integer constants in the range 0 to $2^{31} - 1$. Assume a register-to-register memory model (see page 250 in *EaC2e*) with one class of registers.

ILOC test blocks contain commas and assignment arrows, which are composed of an equal sign followed by a greater than symbol, as shown (=>).

Each ILOC operation in an input block must begin on a new line.[3] Whitespace is defined to be any combination of blanks and tabs. ILOC opcodes must be followed by whitespace. Whitespace preceding and following all other symbols is optional. Whitespace is not allowed within operation names, register names, or the assignment arrow. A double slash (`//`) indicates that the rest of the line is a comment and can be discarded. Empty lines and `nop`s in input files may also be discarded.

[2] A *basic block* is a maximal length sequence of straight-line (*i.e.*, branch-free) code. We use the terms *block* and *basic block* interchangeably when the meaning is clear.
[3] Carriage returns (CR, \r, 0x0D) and line feeds (LF, \n, 0x0A) may appear as valid characters in end-of-line sequences.

The syntax of ILOC is described in further detail in Section 3 of the ILOC simulator document, and, at a higher level, in Appendix A of *EaC2e*. (See the grammar for *Operation* that starts on page 726.) Note that your allocator is not required to support either labels on operations or the square bracket notation used to group multiple ILOC instructions.

Note: You will use the same subset of ILOC instructions, albeit with different latencies, when you write a scheduler for Lab 3. We encourage you to reuse in your Lab 3 scheduler both your Lab 1 routines for reading ILOC files and your Lab 1 routines for register renaming.

Simulator Usage Example: If `test1.i` is in your present working directory, you can invoke the simulator on `test1.i` in the following manner to test your register allocator:

```
/clear/courses/comp412/students/lab1/sim -r 5 -i 2048 1 2 3 < test1.i
```

This command will cause the simulator to execute the instructions in `test1.i`, print the values corresponding to ILOC `output` instructions, and display the total number of cycles, operations, and instructions executed.

The `-r` parameter is optional and restricts the number of registers the simulator will use. (In the example, the number of registers is restricted to 5, so the simulator will recognize `r0`, `r1`, `r2`, `r3`, and `r4` as valid input.). You can use the `—r` parameter to verify that code generated by your allocator uses at most *k* registers. You should not use `-r` when running the original (non-transformed) Lab 1 report and timing blocks.

The `-i` parameter is used to fill memory, starting at the memory location indicated by the first argument that appears after `-i`, with the initial values listed after the memory location. The Lab 1 ILOC simulator has byte addressable memory, but the ILOC subset for Lab 1 and Lab 3 only allows word-aligned accesses. So, in the above example, 1 will be written to memory location 2048, 2 to location 2052, and 3 to location 2056. (This means that, before the memory locations are overwritten during program execution, "`output 2048`" will cause 1 to be printed by the simulator, "`output 2052`" will cause 2 to be printed, etc.) When computing addresses for new spill locations, your allocator must generate word aligned addresses (e.g., *addr* MOD 4 = 0).

The `-x` parameter will be used to verify that your allocator passes the Lab 1 code check. For example, if your `412alloc` implementation produces a file of renamed ILOC code called `renamed_block.i`, the following command can be used to check whether or not renaming was correctly performed:

```
/clear/courses/comp412/students/lab1/sim -x < renamed_block.i
```

See the ILOC simulator document for additional information about supported command-line options. (Note that the command-line options `-d`, `-s`, and `-c` are not relevant for Lab 1.)

A-2. ILOC Input Blocks

A collection of ILOC input blocks is available on CLEAR. The comments at the beginning of each input block specify whether or not the input block expects command-line input data via the ILOC simulator's `-i` parameter. If you experience problems with the input blocks, please submit bug reports to comp412report@rice.edu so that the COMP 412 staff can either fix or delete the problematic blocks.

Input Blocks for Lab 1 Report: The Lab 1 report blocks, which must be used to produce Table 1 for your Lab 1 report, as described in § R-1.2, are available on CLEAR in:

```
/clear/courses/comp412/students/lab1/report
```

The Lab 1 timing blocks, which will be used by the timer described in § A-3.2 to produce timing information for the Lab 1 report, as described in § R-1.2, are available on CLEAR in:

```
/clear/courses/comp412/students/lab1/timing
```

Other Available ILOC Input Blocks: Since the ILOC subset used in Lab 3 is the same as the ILOC subset used in Lab 1, you can further test your allocator by using the Lab 1 and Lab 3 ILOC test blocks available on CLEAR in subdirectories of:

```
/clear/courses/comp412/students/ILOC
```

Input blocks created by past COMP 412 TAs are available in the `blocks` subdirectory. Input blocks contributed by past COMP 412 students are available in the `contributed` subdirectory.

A-3. Tools

A-3.1. Reference Allocator

To help you understand the functioning of a local register allocator and to provide an exemplar for your implementation and debugging efforts, we provide the COMP 412 reference allocator. The reference allocator is a C implementation of a bottom-up local register allocator. The reference allocator follows the basic outline of the algorithm presented in class; it pays careful attention to how it generates spill code. You can improve your understanding of register allocation by examining its output on small blocks. You will also use the reference allocator to determine how well your allocator performs in terms of effectiveness (quality of the allocation that your allocator generates) and efficiency (runtime of your allocator).

The COMP 412 reference allocator can be invoked on CLEAR as follows:

```
/clear/courses/comp412/students/lab1/lab1_ref k <file name>
```

where `k` is an integer (`k` > 2) that specifies the number of registers that the allocator will assume is available on the target machine[4] and `<file name>` both specifies the name of the input file and is a valid Linux pathname relative to the current working directory. For a description of the complete set of flags supported by the COMP 412 reference allocator, enter the following command on CLEAR:

```
/clear/courses/comp412/students/lab1/lab1_ref –h
```

A script for invoking the reference allocator on a directory of ILOC test blocks is available on CLEAR:

```
/clear/courses/comp412/students/lab1/scripts
```

Note that the COMP 412 reference allocator can only be run on CLEAR.

[4] The reference allocator handles k > 2. Your allocator is required to handle 3 ≤ k ≤ 64.

A-3.2. Timer

To produce efficiency data for your Lab 1 report, you will use the COMP 412 timer to determine how the runtime performance of your allocator compares to the runtime performance of the COMP 412 reference allocator over a set of eight timing blocks (`T1k.i`, `T2k.i`, `T4k.i`, `T8k.i`, `T16k.i`, `T32k.i`, `T64k.i`, and `T128k.i`). The timer produces results for *k*=15.

The COMP 412 timer can only be run on CLEAR. To use it, copy the tar file available on CLEAR at

```
/clear/courses/comp412/students/lab1/timer.tar
```

into a directory in your own file space on CLEAR. Unpack the tar file. Read the `README` file. Invoke the timer as follows:

```
./timer <f1>
```

where `<f1>` specifies the path name of your allocator (or the shell script that invokes it). The timer assumes that your allocator accepts the command-line arguments and input described in § C-2. `<f1>` must be a valid Linux pathname relative to the current working directory. The eight timing blocks included in the tar file must appear in the same directory as the timer.

Note: The timer will run `<f1>` on the input files in order from the smallest to the largest file and print timing results when it completes each input file. The timer will quit if your allocator requires five minutes or more to process a single input file.

A-4. Allocator Timing Results

This section of the lab report has not been updated as of August 24, 2017. We will issue a revised version before the Code Check 2 deadline. The 2014—2015 results are, however, good enough to provide you with an idea of how a "good" allocator performs on the timing tests.

The timing results shown in Figure 1 on page 18 were produced using allocators submitted by COMP 412 students in Fall 2014 and Fall 2015. The featured allocators all produced correct code and received high scores for both allocator effectiveness and efficiency. These graphs document the most efficient C, C++, Java, Haskell, OCaml, Python, Ruby, and R implementations submitted and show the expected relative speed differences and the curve shapes for these languages. When discussing the impact of your programming language choice on the efficiency of your allocator implementation, if your allocator is written in C, C++, Java, Haskell, OCaml, Python, Ruby, or R, you are expected to compare your allocator timing results to the Figure 1 results for allocator(s) written in your chosen programming language. (See § 1.2.) While direct comparisons with results from past years do not account for software and hardware changes on CLEAR that may have occurred since the results were generated, the observed trends are consistent enough to justify rough comparisons.

When viewing the timing results in Figure 1, note the difference in the y-axis scales across the graphs. If your allocator is written in Java, also note the shape of the two Java curves. Java performance curves can be deceptive; they show complex timing behavior at small input sizes. This behavior arises out of Java's complex runtime system. To judge the asymptotic behavior of a Java program, you need to look at the relationship between runtime and data set size on the large inputs, not the small inputs. We will discuss Java runtime behavior in one of the evening tutorial sessions.

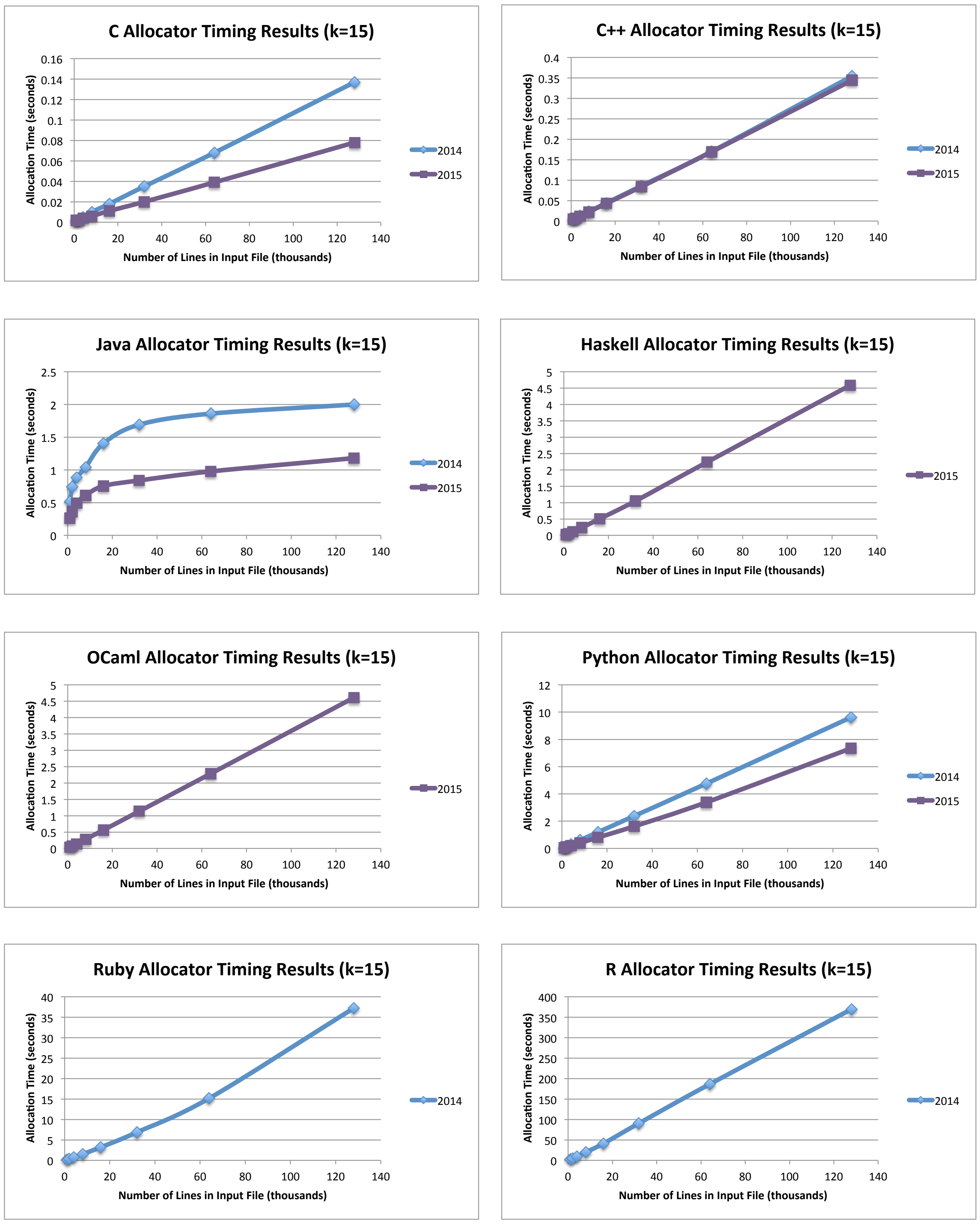


Figure 1: 2014 & 2015 Allocator Timing Results

These numbers will be updated