Assignment Description

In this assignment, you will generate assembly code from the Low IR representation:

1. Abstract Assembly Code Generation. Start with the low IR, build the DAG for the computation in each basic block, then tile the DAG and generate abstract assembly code. You will use the Maximal Munch algorithm for the tiling process. It is recommended that you write comments in the generated code to indicate what instruction sequences correspond to each Low IR instruction. You will be generating 32-bit Pentium code, so all variables and registers will take a full 32 bits of storage. Although it would be possible to generate code that stores booleans more compactly, this data type will be stored in 32-bit words as well. You must generate the appropriate code for each of the following:

- *Stack Frames.* Generate the calling sequences before and after invoking functions, and at the beginning and the end of each function (prologue and epilogue), as discussed in class. Registers eax, ecx, and edx are caller-save, and registers ebx, esi, edi, ebp, esp are callee-save. You must assume that the contents of caller-save registers may be destroyed at each method call. On the other hand, if a function modifies these registers, it must restore them to their original values before returning.

  The function parameters are being pushed on the stack by the caller, in reverse order. That is, the first parameter is pushed last on the stack. The return values are always passed in the eax register.

- *Objects.* You must provide support for objects, as discussed in class. Whenever the program allocates an object with an expression `new A`, you must generate code that allocates the object dynamically with `__allocateObject(s)`, where `s` is the size of the object. The object size must accommodate all of the fields plus a reference to the dispatch vector (DV).

  All of the method calls are virtual calls: for a call `o.m(...)` you must look up the dispatch vector of object `o` for method `m` and perform an indirect call. You must pass the object `o` as the first parameter, followed by the formal parameters of the method. For calls of the form `m()` you must pass `this` as the first parameter. For methods, use a naming scheme where a method `m` of a class `A` is named `_A_m`.

  For field accesses `o.f` you must access the memory location at address `o` plus the constant offset for field `f`. Field accesses of the form `f` are equivalent to `this.f`.

- *Arrays and Strings.* Arrays and strings will be stored in the heap. To create new arrays, use `__allocateArray`; to concatenate strings, use `__stringCat`. String constants will be allocated statically in the data segment. Strings don't have null terminators; instead, each string is preceded by a word indicating the length of the string. Similarly, the length of an array is stored in the memory word preceding the base address of the array (i.e., the location at offset -4).

- *Run-time checks.* You must implement the run-time checks `__checkArrayAccess`, `__checkSize`, and `__checkNull` as assembly instruction sequences. These functions are not provided as library functions.

- *Library functions.* A number of user-level library functions are provided for this assignment (a description of library functions is provided in the IC specification). Unlike methods, each of these functions must be invoked using direct calls. The name of each library must be preceded by two underscores in the generated code. For instance the instruction `call __println` represents a call to `println`. Modify your semantic analysis module to accept and type-check expressions of the form `Library.m(...)`.

- *Initialization.* Make sure you initialize local variables appropriately, with their default values. Dynamically allocated objects and arrays need not be initialized, because the allocation functions fill all of the allocated space with zeroes, which corresponds to the default field or element values.

- *Main function.* If your main function belongs to class `A`, you must generate a wrapper function `_ic_main` which just creates an object `A` and invokes the method `main` of that object.

2. Assembly x86 Code Generation. Use the abstract assembly code to generate x86 code. First, use a naive implementation which spills every local variable on the stack. This implementation will give you a complete compiler and will allow you to test the first part of the assignment.

Then, implement register allocation using the graph-coloring algorithm presented in class, to generate efficient x86 assembly code. You are not required to implement register coalescing and removal of redundant mov instructions. Make sure your register allocator must deal properly with callee-save and caller-save registers. In this part of the assignment, you will use the live variable analysis implemented in the previous project.

Command line invocation. As in the previous assignments, your compiler must be invoked with a single file name as argument, representing the program that you want to compile. In addition to all of the options from the previous assignments, your compiler must support the following command-line options:

- Option `-reg-alloc` to perform register allocation by graph coloring; without this flag, the compiler will perform trivial register allocation which spills all the variables to the stack;
- Option `-dump-aa` to print the abstract assembly code for each function (separately), along with the name of the function being printed.
- Option `-dump-ra` to print the register assignments for each function, in case the `-reg-alloc` option was also specified. The output for each function must print one line for each register, specifying which variables were allocated to that register, or that no variable is allocated to the register.

Invoking the Assembler and the Linker

Given an input file `file.ic`, your compiler will produce an assembly files `file.s`. You can then use the assembler to convert this assembly code into an object file `file.o`, and use the linker to convert this object file into an executable file. To run the gnu assembler `as` on `file.s`, use the following command line: `as -o file.o file.s`. To run the gnu linker `ld` on `file.o` and link this object file with the library `libic.a`, use the following command line:

```
ld -o file.exe file.o /lib/crt0.o libic.a -lcygwin -lkernel32
```

The library file `libic.a` is a collection of `.o` files bundled together, containing the code for the library functions that are defined in the language specification, along with run-time support for garbage collection.

Supporting Material. You can find the library file `libic.a` along with supporting material for this assignment on the web site for this course. The supporting web page includes documentation for the `as` assembler; documentation for the x86 instruction set; and several example IC programs along with the corresponding x86 assembly code.

Electronic submission

You will turn in all of your assignment electronically. We expect your submission at 11:59PM on the due date. Electronic submissions after that time will be considered a day late. Place your files in the directory `\\crane\courses\cs412-sp03\grpX\pa5`, where `grpX` is your group identifier. Please organize your top-level directory structure as follows:

- `src` : all of your source code (do not submit the class files).
- `doc` : javadoc output that we can browse on-line. Include in this directory your write-up (`writeup.ps`).
- `test` : any test cases you used in testing your project.

The write-up file `writeup.ps` from the documentation directory should contain a brief, clear, and concise document (at most 3 pages) describing the your code structure and high-level algorithms. Also, please continue to provide us feedback – tell us how much time you spent on it, what was the most interesting/difficult part, and how you think it could be improved.