

Assignment Description

In this assignment, you will generate assembly code from the LIR representation. You will be generating code for Intel's x86 32-bit architecture **using the AT&T syntax**, so all variables and registers will take a full 32 bits of storage. Although it is possible to generate code that stores Booleans more compactly, this data type will be stored in 32-bit words as well. It is recommended that you write comments in the generated code to indicate what instruction sequences correspond to each LIR instruction. You must generate the appropriate code for each of the following:

Instructions in function bodies. Your code should translate each LIR instruction into a sequence of assembly instructions by accessing the parameters/variables/LIR registers using their offsets from the frame pointer (ebp register). To achieve this, your translation should first assign offsets to parameters, local variables, and LIR registers. For parameters, use positive offsets, starting from +8 for the first parameter (which is `this` for virtual functions) and increase by 4 bytes for each parameter. For local variables offsets should start from -4 for the first local variable and decrease by 4 for each variable. LIR registers should be treated as additional local variables (i.e., should have negative offsets starting from the last local variable).

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. You must assume that the contents of caller-save registers may be destroyed at each method call. Registers `ebx`, `esi`, `edi`, `ebp`, `esp` are callee-save. Your code should not save/restore these registers, unless you decide to perform register allocation optimizations that work across function calls.
- Function parameters should be pushed on the stack by the caller, in reverse order. That is, the first parameter is pushed last on the stack. In virtual function calls, the first parameter is the receiver object (`this` variable).
- Function calls for static functions and library functions are translated into assembly `call` instructions. Virtual function calls include looking up the address of the function at the given offset in the dispatch table and making an indirect call.
- The return values are always passed in the `eax` register.
- Pushing and popping the frame. This requires computing the size of each frame, which is the number of bytes needed for the local variables and LIR registers that are stored in the frame. Each function should include a prologue and epilogue section. The prologue section includes the label where the function's instructions start, using the convention `._A.foo` for a function named `foo` in class `A`. The prologue section saves the previous frame pointer, adjust its value and sets a new value for the stack pointer to provide space for the function's local variables and LIR registers. The epilogue should start with a label of the form `._A.foo.epilogue` for a function `foo` in class `A`. This label is used by the translation of `Return` instructions.
- Copying the returned value from `eax` into the target register for function calls that assign the return value to a register (different from `Rdummy`).

Objects. You must provide support for objects, as discussed in class. This includes generating the dispatch vectors in the data section of the assembly file. Dynamically allocated arrays need not be initialized, because the allocation function fills all of the allocated space with zeros, which corresponds to the default values. Dynamically allocated object fields are also initialized by the `__allocateObject` library function to zero, but your compiler must generate code to properly initialize the `DVPtr` field (using an assembly instruction `movl $ _DV_A, (%ebx)` where `%ebx` holds the address of the new object).

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 (see examples on the web-site). 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). Your translation of the `ArrayLength` instruction retrieves this information.

Runtime checks. You must implement the runtime error handlers `__checkArrayAccess`, `__checkSize`, `__checkNullRef`, and `__checkZero` as assembly instruction sequences. These handlers are not provided as library functions. The runtime handlers must print an error message and gracefully terminate the execution. To implement this, use the code shown in slide 35 of [T12](#). Fragile instructions should be preceded by code that checks erroneous situations and calls the appropriate error handlers (see slides 30–34 of [T12](#)). In addition, each call to a library function should be preceded by code to check that reference type arguments are non-null (e.g., that the arguments of `__stringCat` are non-null references).

Main function. Your main function should be called exactly `__ic_main` (notice the 2 leading underscores) and contain a proper prologue/epilogue sections (like any other function that you translate). This function will be called from an external library function which will take care of passing the command-line arguments to `__ic_main`.

Invoking the Assembler and the Linker

Given an input file `file.ic`, your compiler will produce an assembly file `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](#) under the [Cygwin](#) environment 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 `libc.a`, use the following command line:

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

The library file `libc.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 runtime support for garbage collection.

Supporting Materials

You can find the library file `libc.a` along with supporting material for this assignment on the course web site. 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. The supporting materials also include information for assembling and linking on a Linux environment.

Optional Optimizations

You may choose to implement any of the following optimizations to reduce the number of machine registers used for each function, using the techniques taught in the recitation, and to reduce the number of labels and jump instructions:

Register allocation for single statements (10 pts) The goal of this optimization is to associate LIR registers with machine registers as much as possible, thus avoiding unnecessary memory accesses.

Eliminating unnecessary labels and jumps (5 pts) The goal of this optimization is to: (i) eliminate consecutive labels; (ii) eliminate jump instruction to an immediately following label (i.e., `jmp _label13` `_label13;`); and (iii) eliminate labels that are not mentioned by any jump instruction.

We recommend that you implement any optimizations only **after** you validate your assignment against a suite of tests. The most important thing is to ensure you have a baseline translation that you trust to be correct. We also recommend that you keep two versions of the translation—the optimized and non-optimized translation—separately (instead of rewriting the baseline translation with optimizations). This reduces the possibility of introducing translation errors with the optimizations.

Command line invocation: It should be possible to invoke your compiler with a single file name as argument and possible options:

```
java IC.Compiler <file.ic> [options]
```

With this command, the compiler parses the input file (and `libc.sig`), constructs the AST, conducts semantic analysis (reporting any errors it encounters), translates the program to LIR representation, and then generates x86 assembly code using AT&T syntax into `file.s` (where `file.ic` is the name of the input program). Your compiler must support the command-line options specified in the previous assignments, as well as the command line option `-opt-asm`, which tells the compiler to activate the code generation optimizations you choose to implement (otherwise it has no effect).

You are also expected to design a suite of tests (IC programs) that demonstrate the correctness of your translation and the effect of optimizations on the translation.

Package Structure: You will implement the translation to x86 assembly in a new sub-package `asm`.

What to turn in

You must turn in your code electronically in the team account on the due date, including a documentation write-up. **Please include only the source files in your submission, not the compiled class files or any other temporary files.**

As in any other large program, much of the value in a compiler is in how easily it can be maintained. For this reason, a high value will be placed here on both clarity and brevity – both in documentation and code. Make sure your code structure is well-explained in your write-up and in your javadoc documentation.

Turn in a document `PA5-DOC.XXX` (XXX can be one of `txt/doc/pdf`) with the following information:

- A brief, clear, and concise description of your code structure and testing strategy.
- A description of the class hierarchy in your `src` directory, brief descriptions of the major classes, any known bugs, and any other information that we might find useful when grading your assignment. Documented bugs will be graded more forgivingly than non-documented bugs.
- A high-level description of how you translated LIR to assembly code. You should also describe any optimizations you implement.
- Feedback. We are interested in hearing your overall thoughts about the assignment: how much time you spent on it, what was the most difficult or most interesting part, and how you think it could be made better. This part is optional.

GOOD LUCK!