Final Assignment -- Target Code Generation

The final part of the compiler is the back end which generates assembly language from quads, with some help from the symbol table. It is recommended that you do X86 architecture since that's what was covered in class. You could do either X86-32 or X86-64. There is no requirement to support both! If you'd like to do another architecture, such as ARM, you'll have to be prepared to demonstrate it on your own environment, as the class VM won't support it.

Since quads are theoretically independent of the source language, there are no more requirements and exemptions having to do with the C language. Whatever you had to support in Assignment 5 you need to support in Assignment 6, and likewise all optional things continue to be optional.

That being said, you might find it helpful, in the interests of time, to stick with just int and pointer data types. Floating point has always been optional, and dealing with conversions between different integer types, while not difficult, could be tedious.

When you have successfully completed this assignment, your compiler should be able to accept a simple test program (as usual, OK to use cpp/gcc -E to pre-process the source) which demonstrates:

- Accessing global variables
- Declarative pseudo-opcodes for global declarations
- Accessing local variables including stack frame allocation
- Computation of expressions
- Reading and writing array elements
- Dereferencing pointers and taking address-of
- String constants
- Control flow (if statements and at least one type of loop)
- Calling external function, with arguments (int or pointer)

The result of your program (on stdout or to a file specified on the command line) is a .s file. You should then be able to run

```
        gcc -m32 file.s
OR
        gcc -m64 file.s
```

which will run the system's assembler on your output, invoke the system linker to link in the standard C library and run-time startup code, and produce an executable a.out which, when run, should produce correct output! You could call the standard C `printf()` function to generate run-time output. Your compiler will probably not be able to deal with the system header files (which may contain constructs that are optional). Just create a simple prototype declaration such as `int printf();` someplace in your test C source, prior to calling printf. Or, if you supported implicit declaration of functions, you'll be fine. *Note: on X86-64, remember to set the %eax register to zero before calling printf or other variadic functions*

The -m32 flag is to be used for X86 32 bit code. If you have generated 64-bit assembly code, use -m64. Because

the function calling APIs are different between the 32 and 64 architecture, it is important that you select the correct flag!

As discussed in class, it is not required that your assembly code be "optimal" in any way, but it must be correct.

Instruction Selection

Instruction selection is a hard problem. The recommended approach is to write a very simple instruction selector which looks at the quad opcode and each operand addressing mode in an ad-hoc fashion, one quad at a time, and generates a sequence of one or more assembly instructions per quad.

X86 is generally a 2-address architecture, and has additional address mode combination restrictions which are mentioned in the lecture notes. You might want to reserve two of the "scratch" registers for dealing with situations where the quad is too rich to express in one step. E.g.
`i{lvar}=ADD j{lvar},k{lvar}`

```
##OK sequence:
movl      (-16)%ebp,%eax              #j is at offset -16 in stack frame
movl      (-20)%ebp,%edx              #k is at offset -20
addl      %edx,%eax          #add, result in eax
movl      %eax,(-24)%ebp              #move result to i at offset -24

##More optimal sequence:
movl      (-16)%ebp,%eax              #j is at offset -16 in stack frame
addl      (-20)%ebp,%eax              #k is at offset -20
movl      %eax,(-24)%ebp              #move result to i at offset -24
```

Quad vs Assembly Addressing Modes

In our examples, we have often elided the operand "addressing mode" with quads. For example, `a=MOV 1` doesn't say what type of variable `a` is (local, global, parameter). But that information is essential for assembly generation.

Our quad schema were for instructional purposes and were not meant to be "externalizable." Comparing with LLVM, there is a significant amount of information from the symbol table that we haven't been presenting in our quad output. This is OK for your demonstration compiler and Assignment 6, since all of the data structures you have built (in particular, the symbol table) are still in memory when you get to the assembly generation phase.

Declarations, Strings, Assembler Directives

Refer to Unit 7 lecture notes for information on assembly directives. Note that for all global variable declarations (including static variables in function/block scope) you will need to emit the appropriate directives when you see those declarations. You could do this as you see each declaration, or you could make a second pass over the symbol table prior to generating the first assembly language opcodes.

Initialized declarations were optional since assignment 3, but if you implemented them, take careful note of the

difference between initialized and uninitialized global variables when it comes to assembler directives.

Also make sure you understand how to create string literals in assembly, as this will be essential for calling printf to get output from your test cases.

Local Variables and Stack Frame

Local variables do not result in directive emission, but you will need to keep track of the stack frame offset of each variable, assigning a unique "slot" to each variable. You'll also need a slot for register allocator spills. If you aren't doing any kind of register allocation, nor any kind of live value analysis, then each temporary value winds up requiring a slot. This will be a big waste of stack space, but it will work for demo purposes.

Each function will require the typical "prologue" to save the base pointer and move the stack pointer to create room for the local variables, along with the typical "epilogue" to unwind that at function exit.

Register Allocation

As discussed in lecture notes #8, register allocation can be a very difficult problem, far beyond our ability to tackle in the limited time left in the course.

The most primitive yet still functional approach to register allocation is to use the same scratch registers each time to bring memory operands into registers, as needed for each operation. There is no attempt to have local variables reside in registers in between quads, and even temporary values (such as arise in complicated expressions) would be considered "phantom" local variables and would own a specific stack frame slot for the duration of the function, even after the tempvar is no longer live.

You can attempt to do a primitive register allocator for tempvals using a "scoreboard" approach by making some assumptions based on how you generated quads. If you only assign to a temporary value once and only use it once, you do not need to worry about live value analysis. You can just assume that the temporary is dead after its first and only use as a *source* operand. But of course if you don't generate quads this way, don't expect it to work!

A more sophisicated local register allocator would use the "scoreboard" approach (within a given basic block) for local variables too. Global register allocation with full live variable analysis is unrealistic given the time constraints.

General Hints

If you are unsure of what opcode to pick, or how the instruction works, the best approach is to create a tiny test program which exercises the operation under consideration (e.g. integer division), run it through gcc -S to see what gcc picks, then look up the opcode and addressing mode in the assembly language reference manual for the architecture in which you are working. The Unit 7 lecture notes also contain a lot of information about X86 architecture, opcodes, addressing modes, and pseudo-opcodes. Remember, you may need to give gcc the -O0 flag to turn off the optimizer, otherwise your entire test program may get optimized away because the compiler is smarter than you anticipated.

When looking at GCC output, do not be concerned with comment directives, or sections other than text, data and bss (.comm). You may see some very confusing gcc output which is intended for debugging, integrity checking,

exception handling, etc. In the Unit 7 notes, this extraneous output was sanitized for your psychological protection.

You might also find that running GCC on some of the test cases from Unit 7 generates different sequences of instructions and/or registers than the examples. Different versions of GCC produce different code. You can assume that both the examples and the code that your GCC generates are correct (i.e. that there are no bugs in GCC). And of course, a compiler such as clang might produce significantly different code.

You may find it helpful for debugging to append a comment to each line of assembly that you output, stating the basic block # and/or quad # within that block which caused you to generate that line, along with (if applicable) the original source and destination operands. E.g.

```
a{lvar} = MOV       %T1

          movl      %ecx,-12(%ebp)                #Q0003 a{lvar}->%T1{tempvar}
```

(this example assumes that you have done register allocation and %T1 was assigned to ecx at this point)

Some students have found the web site `godbolt.org` to be helpful. It contains a tool which allows you to input C and view the AST, IR, CFG, and assembly output, from various compilers.