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. Since we have covered the X86 architecture in class, that is the recommended one to use. You could do either X86-32 or X86-64.

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.

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

```
* Reading and writing local and global variables
* Computation of expressions
* Reading and writing array elements
* Creating and dereferencing pointer values
* 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

```
cc -m32 file.s
```

OR

```
cc -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! Call the standard C printf() function to generate run-time output. Since 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.

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

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

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

You could implement a primitive register allocator without having to do live variable analysis by assuming that all variables mentioned explicitly in the program will always reside in their memory locations and will be brought into registers only while they are being operated on. Then temporary values could be considered as "phantom" local variables which "own" a slot in the local stack frame.

Alternatively, you can attempt to do a primitive register allocator with the tempvals 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!

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 (manuals for SPARC and X86 have been placed on the course web site). 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.

Remember that assembly output needs to be generated by declarations of global or static variables, and the treatment is different depending on the presence of an initializer. This was covered in Unit 7. You also need to handle string literals if you expect printf to work.

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, exception handling, etc. In the Unit 6 notes, this extraneous output was sanitized for your protection.

You might also find that running GCC on some of the test cases from Unit 6 generates different sequences of instructions and/or registers than the examples showed. 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)

As usual, if you have any questions about what GCC is doing, or how to proceed, ask!