CS 302 Homework 4 – due 3:30 p.m., Wednesday, June 2, 1999

Generating SPARC Assembly Code for PCAT

Working individually or in teams of two, write a SPARC code generator for (a subset of) the PCAT language. As before, *exclude* anything to do with real numbers, records, and the `LOOP` and `EXIT` statements. As a reminder, the abstract syntax for this subset of **PCAT** is described in `/u/cs302acc/3/ast.txt`.

To ease the implementation task, you can continue to assume (as in Homework 3), that the `WRITE` statement should output the strings "`1`" and "`0`" for booleans, instead of "`TRUE`" and "`FALSE`".

As in Homework 3, you should generate code for array references that handles array bounds violations by calling a specified runtime library function, which prints "`Array bounds violation`" to `stderr` and exits immediately.

Don't worry about the runtime behavior of PCAT programs that attempt to allocate arrays whose size exceeds the available heap memory at runtime, or that fail to return values for function procedures.

Finally, in order to avoid the need to spill registers (see Implementation, below), it is acceptable if your generator fails for programs containing procedures with very large numbers of arguments, arrays with very large numbers of initializers, or expressions with very large numbers of terms.

You can base your code generator on the intermediate code generator from assignment 3, but you must deal with various complications we ignored there, including addressing of local variables and arguments (from both caller and callee perspectives), restricted operand formats, unnesting of procedures, etc.

Your intermediate code generator should be an executable `gen` that reads an `.ast` file from `stdin` and produces an intermediate code (`.s`) file on `stdout`. This file can then be assembled and linked via `gcc` with a "standard library" `/u/cs302acc/4/pcatlib.o` to make a SPARC executable. When executed, programs should behave as they did under the various interpreters.

As usual, a "correct" generator is available in `/u/cs302acc/4/gen`. It is not necessary that you generate code identical to my `gen`, but it must behave the same way as my code does when assembled and executed.

Linkage and Library

The "main program" (top-level code) of a PCAT program should look like a C main program, i.e., it should generate a global assembly routine called `main`. If the resulting object code is linked under `gcc`, the linker will arrange to invoke `main` automatically when the executable starts. `main` should return the value `0` to indicate normal, successful completion of the program.

IO and heap memory allocation are performed by issuing C-style procedure calls to the library functions `read_int` (which returns an integer result value), `write_int` (which takes an integer argument), `write_string` (which takes the address of a string as argument), `write_newline`, `bounds_error` (which takes no arguments, issues the message `Array bounds violation` to `stderr`, and exits), and `alloc` (which takes an integer size argument in bytes and returns the address of the allocated storage). The library is written in C; the source is in `/u/cs302acc/4/pcatlib.c`. Remember that C-style calls put their arguments in `%o0`, `%o1`, etc. (*not* on the stack) and return their result in `%o0`.

Implementation

`/u/cs302acc/3/sparc.[ch]` has definitions and supporting code for generating SPARC code. A skeleton for a working generator is in `gen0.c`; feel free to use this as the basis for your generator if you wish. The control flow mechanisms from assignment 3 should carry over unchanged.

The SPARC's operands are considerably more restrictive than those of the intermediate code in assignment 3. In particular, note that only the second operand of a 3-operand form may be an immediate value; the first must be a register. It is probably easiest just to bring all values into registers uniformly, at least to start with. However, the SPARC does support register+offset addresses in `ld` and `st` instructions, and you should use these when possible to reference variables in the frame.

Use a simple register allocation scheme, as implemented in `sparc.c`. If you run out of registers, just issue an error and die.

For PCAT procedures, use the following conventions for frame layout. Local variables start at `%fp-4` and are allocated in successive decreasing addresses. The usual 17 free words are left above `%fp`. Every procedure (except the main program!) has a static link, which is passed as a "hidden" first argument, and lives at `%fp+68`. The "real" arguments live at `%fp+72`, `%fp+74`, etc. The argument build area is as large as needed to hold the maximum number of arguments passed, but always at least 6 words big.

All variables, including those of the main program, are stack-allocated. You'll need to use a symbol table to keep track of variable addresses. Just record the offset (positive or negative) of each variable from the frame pointer. You can also use the symbol table to record the the values of the initial constants `TRUE` and `FALSE`.

It isn't necessary to worry about types; you can continue to treat all values as 4-byte words. Remember that address calculations on the SPARC are in terms of bytes, not words; e.g., array offsets need to be multiplied by 4.

The assembler doesn't support nested procedure definitions; all procedures become top-level. Since multiple procedures (at different scopes) may have the same name in a PCAT source program, you must disambiguate names, e.g., by appending a unique integer to each name. Put these numbers in the symbol table too.

Assembler Tricks

Use the special `gnu` SPARC assembler found in `/pkgs/gnu/bin/as`. (This is *not* the same as `/usr/ccs/bin/as`, which is the assembler you get via `gcc x.s`.)

Within a procedure body or variable initializer, code can be emitted one line at a time as it is generated; there is no need to store it up. Unfortunately, if we emit code for declarations in order, code for variable initializers will be interspersed with code for nested procedure bodies. This problem is most easily solved in the special `gnu` SPARC assembler, because it allows you to switch output among multiple numbered text segments. The pseudo-op `.text`n directs the following code into the next free location in text segment n. After the entire input file has been read, the assembler concatenates all the code segments in numeric order. You can use the current scope level number as segment number. Similar considerations apply to emitting string constants, which is most conveniently done when the string is encountered, say to text segment 0. Note that read-only constants should be placed in a `text` segment rather than a `data` segment.

Another handy assembler feature is the `=` pseudo-op, which allows you to use a symbolic constant *before* it is defined. In particular, you can use this feature to emit the `save` instruction for a procedure before you've calculated its frame size.

Machine Code Features and Tricks

Most processors have certain idiosyncratic instructions, and the SPARC is no exception. The most troublesome are the `mul` and `div` instructions. The product of two 32-bit numbers can take 64 bits to express; conversely, it makes sense to provide a 64 bit dividend (first argument) to a 32-bit division. The SPARC uses a special register called `%y` to hold the high-order 32 bits of the result of a multiply, and the high-order 32 bits of the dividend of a divide. You can just ignore this after the multiply, since, like most high-level languages, PCAT quietly throws away the high-order part of the result. But it is essential to clear the `%y` register before doing a divide, because the results of a previous multiply may be sitting around in it. This is done via an explicit `wr` instruction, which is documented as requiring up to three extra cycles to complete. Hence the complete sequence for a `DIV` is:

`wr %g0,0,%y; nop; nop; nop; udiv` *r1*`,`*r2*`,`*r3*

Incidentally, SPARC chips are permitted (but not required) to put the remainder of a division into apparently do not do this, however; hence the need for a messy code sequence to implement MOD.

There is no end to the number of tricks that can be used to speed up or compress machine code. Here is one used in my `gen`. When checking an array index i against the bounds of a 0-based array of size s, one must make sure that $i \geq 0$ and $i < s$. Assuming that i is a signed integer, this can be done using a single *unsigned* integer comparison:

`cmp` *i*`,`*s*`; bgeu` *bad*

This is because a negative value for i corresponds to a very large unsigned value.

Submitting the Program

Prepare a `makefile` that builds your generator and produces an executable called `gen`. Submit your program by `mail`ing a `shar` "bundle" containing *all* the relevant files to `cs302acc@cs.pdx.edu`.