

CS 322 Homework 4 – due 1:30 p.m., Saturday, Mar. 12, 2003

Generating SPARC Assembly Code for PCAT

Working individually or in teams of two, write a SPARC code generator for the IR used in assignment 3. In principle, your generator should work on all IR programs; however, in practice it suffices for it to work on IR programs generated from PCAT programs using the reference solution to assignment 3.

Don't worry about the runtime behavior of programs that attempt to allocate more heap space than is available at runtime, or that fail to set 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 IR programs having more than 19 temporaries with simultaneously overlapping live intervals; such programs may be generated from PCAT programs containing procedures with very large numbers of arguments, arrays or records with very large numbers of initializers, or expressions with very large numbers of terms.

The code generator should read the `.ir` file from standard input, The generator should produce assembly code on standard output, which is normally captured into an `.s` file. This file can then be assembled and linked via `gcc` with a “standard library” `/u/cs322-01/pcatlib.o` to make a SPARC executable. When executed, programs should behave as they did under the various interpreters in previous assignments.

Runtime errors such as array bounds violation should be handled by generating code to call the appropriate built-in runtime library function, e.g. `array_bounds`, as detailed below.

As usual, a “correct” generator is available in the jar file `sparcgen.jar`; it can be run on file `foo.pcat` by typing

```
java -classpath sparcgen.jar SparcGenDriver < foo.ir > foo.s
```

It is not necessary that the code you generate be identical to what my `sparcgen` generates, 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 an integer value (always 0, for PCAT programs) which the operating system treats as a completion status code.

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` (which takes no arguments), `bounds_error` (which takes no arguments, issues the message `Array bounds violation` to `stderr`, and exits), `nil_pointer` (which takes no arguments, issues the message `NIL pointer dereference` 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 `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

A large collection of support classes and functions are provided in file `ir.jar`; it contains classes for representing, lexing, parsing, and checking IR files, for calculating liveness of IR temporaries, and for emitting SPARC code. Source file `IR.java` documents the IR representation and `Sparc.java` documents the SPARC code emission methods.

Your code generator must be defined by a class `SparcGen` that compiles and links with `ir.jar` and the `SparcGenDriver` class provided; you must not change these classes. Thus, your `SparcGen` class must implement the method

```
static void genProgram(IR.Body body)
    throws SparcGen.SparcGenException
```

where `SparcGenException` is a subclass of `Exception` that you define in the `SparcGen` class, which is thrown in the event of an unrecoverable error while generating code.

Your generator class should be placed in a separate file `SparcGen.java`, which can be compiled using

```
javac -classpath .:ir.jar Sparc.java SparcGen.java
```

A skeleton for a working generator is available in `SparcGen0.java`; feel free to use this as the basis for your generator if you wish.

The register allocator in the skeleton is extremely simplistic, and will fail if more than 19 temporary registers are used in a procedure. You should implement a linear scan or graph-coloring register allocator instead; the linear scan approach is easiest. You can use the methods in class `Liveness.java` to calculate which temporaries are live after each instruction, and to calculate the corresponding live intervals. In particular, you can use the method

```
static Map calculateLiveIntervals(IR.Body body)
    throws Liveness.LivenessException
```

which calculates the live interval associated with each temporary in `body`. The returned `Map` has `IR.RegTemp` object as keys, and `Liveness.Interval` objects as values. The latter objects have two integer fields `start` and `end`, representing the interval by its first and last indices in `body.codeLines`. A `LivenessException` is raised if the the liveness calculation fails because the body's code "falls of the end" rather than executing a `return`. Starting from this map, you can build a register allocation `Map` whose keys are `IR.RegTemp` objects and whose values are `Sparc.Reg` objects. If you run out of registers, throw a `SparcGenError` exception.

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

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 (and then only if it fits into 13 bits); the first must be a register. 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.

Remember that address calculations on the SPARC are in terms of bytes, not words, so the implementation of the IR's `adda` instruction needs to scale its second argument by 4. The `alloc` system call also needs special treatment; see the skeleton code for details.

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 an environment to keep track of variable addresses. Just record the offset (positive or negative) of each variable from the frame pointer.

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 environment too. You can also use them to disambiguate labels (which must be unique across the entire assembler file).

Use the `%g` registers for very short-lived scratch registers. Remember that `%g0` is always 0.

Don't forget about delay slots! Use them where it is convenient, but just generate `nop` fillers otherwise.

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 SPARC sequence for an IR `sdiv` is:

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

Incidentally, SPARC chips are permitted (but not required) to put the remainder of a division into `%y` as well. Our processors apparently do not do this, however, which is why `mod` was not in the IR's instruction set.

Submitting the Program

Place your `SparcGen` class in the single file `SparcGen.java`, and mail it to `cs322-01@cs.pdx.edu`. You must mail this file as a plain text *attachment*; the contents of the message itself don't matter. We should then be able to compile your code by creating a fresh directory, saving your attachment, copying in the provided `ir.jar` and typing

```
javac -classpath .:ir.jar SparcGen.java
```

If we also copy in the provided file `SparcGenDriver.class`, we should then be able to execute your generator on file `foo.ir` by typing

```
java -classpath .:ir.jar SparcGenDriver < foo.ir
```

Note that we will be using automated mechanisms to read, compile, and test your programs, so adherence to this naming and mailing policy is important! You may lose points if you fail to submit your program in the correct way.

If you are working in a team of two, only one team member should submit a solution, which must have the names of both team members in a comment at the top; the other team member should send mail identifying him- or her-self as a team member.