Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science

6.035, Fall 2000 Handout 22 – Running Under IRIX Tuesday, November 2

IRIX sucks.[1]

This handout describes what you need to do to run your Decaf/Espresso programs natively on an SGI machine running IRIX 6.5.[2] The first part of this document gives an overview of UNIX toolchains, and the IRIX toolchain in particular. The second part describes what your compiler must do in order to generate assembly code appropriate for the rest of the IRIX toolchain. The third part lists the commands you need to build a native IRIX program from your Decaf/Espresso code.

1 UNIX Toolchains

The toolchain is the collection of software used to build *executables* and *libraries*. The compiler is only the first piece of the toolchain. For the most part, 6.035 ignores everything after the compiler phase, since the rest of the stages have more to do with operating system design than compilers, but you need to have a basic understanding of the rest of the toolchain in order to build working Decaf/Espresso executables. You should already understand the job of the compiler: it takes input in the form of some high level language, and outputs assembly code for some architecture.

The next stage of the toolchain is the *assembler*. The assembler's job is to turn the textual assembly language output of the compiler into binary machine code. The assembler's output, called an *object* file, is the actual 1's and 0's that the CPU executes, along with relocation information used in the next phase.

If the end goal is to produce a library (explained later), the next phase is some archival utility that does little more than package a bunch of objects together. On most systems this is done with `ar` and `ranlib`.

If the end goal is an executable, then the last stage of the chain is the *linker*, which combines all the necessary objects for a program (most programs come from more than one source file, and hence more than one object file), fixes the addresses using relocation information in the object files, and appends the appropriate headers. The output of the linker is the final executable, which is a binary program that runs under the operating system.

2 Keys to a successful compiler

2.1 Calling Conventions

You can get away with calling convention murder when you run your programs under SPIM, but you'll lose under IRIX. The current version of IRIX (6.5) supports both the n32 and o32 conventions.

[1] For the canonical list of operating systems that suck, see `http://linuxmafia.com/cabal/os-suck.html`.

[2] IRIX is the SGI operating system for its MIPS-based machines, including Athena Indy's and O2's. It sucks, even when compared against other sucky operating systems. The toolchain is especially sucky.

IRIX is moving in the n32 direction, and n32 is now the default, but the toolchain still supports o32. When we started supporting native mode o32 was more popular, so we still default to that, but you can use the n32 calling convention and libraries if you prefer.

Let's review the key features of o32:[3]

- At function call points, `$sp` must be 8-byte aligned.
- The first four arguments to a function are passed in `$a0` through `$a3`.
- The remaining arguments are passed on the stack. At the call point, the first argument is lowest in memory (top of stack), and the rest of the arguments follow. The first four stack slots are for the first four arguments, even though they are passed in registers. The value you put in those slots doesn't matter, but the slot must be there.
- Regardless of the number of arguments, you must allocate at least 16 bytes of argument space on the stack (4 slots).
- A function's return value is placed in `$v0` upon return to the caller.
- A function returns to the caller by jumping to the address that was in `$ra` upon entry to the function.
- Across calls, `$s0` through `$s7`, `$fp`, and `$gp` are all preserved. Functions are allowed to clobber `$t0` through `$t9` and `$a0` through `$a3`.

The n32 convention is similar, but it passes the first *8* arguments in registers, and does not require the caller to allocate stack space for the arguments passed in registers. It also requires that `$sp` be 16-byte aligned. Because of these differences, o32 and n32 code is incompatible, and all the objects linked together in an executable must use the same convention. Bug your TA if you want to know more about compiling for n32.

2.2 Passing things to external functions, and passing them back

When calling external functions, your compiler needs to generate code that passes reasonable argument values. Integer arguments are straightforward. Strings should be passed by placing the address of the string in the argument slot; you can do this using the `la` instruction:

```
    .data
    .align 2
my_string:
    .asciiz "Hello, world\n"
    .text

    la a0, my_string
    callout "printf", 1
```

[3] I'm ignoring lots of details here, especially related to floating point and passing larger data objects. For an excellent reference on the MIPS calling convention, and MIPS programming in general, see Dominic Sweetman's *See MIPS Run*.

Arrays are similarly passed using the base address of the array. Espresso arrays store the length in the first slot, so callouts should pass the address of the second slot to be compatible with C-style arrays.

You can callout to `malloc()` to get more memory from the operating system; the callout will leave a pointer to the memory in `$v0`.

Lastly, you can cheat to save pointers that external functions return by storing them in integer variables. This works because on 32-bit MIPS platforms, integers and addresses are both 32 bits.[4] For example, `fopen()` returns a pointer to a C `FILE` structure, which you can then pass to `fprintf()` when you want to write to the file. You can do file I/O in Decaf/Espresso by doing something like this:

```
{
    int file;

    file = callout("fopen", "filename", "r");
    callout("fprintf", file, "text to put in file\n");
}
```

2.3 Useful assembler directives

Handout 18 describes some of the assembler directives you need when writing assembly files. When writing assembly for the native toolchain, you might also want to add some additional directives. These directives help debuggers understand your stack frames so that you can generate stack traces and the like.

`.ent/.end` (`java6035.tools.ASM.d{Ent,End}`)
Marks the start/end of a function. This is useful for debuggers. A trivial function might look something like this:

```
    .text
    .ent foo
foo:
    addu $v0, $a1, $a2
    j $ra
    .end foo
```

`.frame` (`java6035.tools.ASM.dFrame`)
Indicates the location and size of the stack frame, and the location of the return address. The syntax is:

```
    .frame framereg, framesize, returnreg
```

[4] Once you leave this class, don't ever do this. It's only suggested here because Decaf/Espresso has no other way of storing a pointer. Mixing pointer and integer types in C is famous for causing problems when porting to 64-bit platforms like the Alpha.

where *framereg* holds the register used to access items on the frame (either `$fp` or `$sp`), *framesize* is the size of the frame, which should be whatever you subtract from `$sp` in the prologue, and *returnreg* is the register that holds the return address. This may be `$ra` if your function never modifies it, otherwise put `$0` if it is saved somewhere in the frame.

`.mask` (`java6035.tools.ASM.dMask`)
Indicates where registers are saved in the stack frame. To allow debuggers to figure out what's going on, they need to be able to find saved values on the stack frame. This directive is part of the solution, but your compiler must save things on the frame in a particular way as well. The syntax is:

```
.mask regmask, regoffs
```

where *regmask* holds a saved register mask and *regoffs* indicates the register save offset from *framereg*. The mask is a bitmap of which registers are saved (bit 1 set = `$1`, bit 31 set = `$ra`, etc.). For example, a function that saved `$ra`, `$sp`, and `$s0` would set framereg to `0xc0010000`. This relies on saved registers being saved in order in the stack frame with no gaps, which may not be what your compiler does. Typically, if your framereg is `$fp`, then your offset is going to be a negative number, since `$fp` is at the top of the save area.

2.4 A word on $gp

The `$gp` register is used to help speed access to small global variables by putting them in a special segment, defining a label `_gp` to point in the middle of the segment, loading that address into `$gp`, and then doing loads and stores off `$gp`. This is more efficient because if a load address is too large to fit in the operand field of the `lw` instruction, then the assembler turns it into two instructions to do the load, which is slower.

Unfortunately, using `$gp` requires very careful cooperation between all parts of the toolchain, and honestly, it isn't worth the trouble. You're best off just avoiding the whole mess, and telling the MIPS assembler to not use `$gp` by passing it the `-G 0` option (see below). It gets worse, though, because if you share globals between objects then you run into trouble if some objects are using `$gp` and some aren't. The upshot is that you should assume any callout to an external function will clobber `$gp`, and your generated code should take care to save and restore it after each callout.

3 Using the IRIX 6.5 toolchain

3.1 Simple programs

We'll start with the simple case of just compiling your standalone application for use under IRIX. You start by compiling your Decaf/Espresso program into assembly, setting the *format* parameter of the `MIPSGenerator` object to `NATIVE_FORMAT`. You'll want to rename your assembly listing file to *progname*`.s`, because the toolchain is just that stupid. Then, do

```
athena% cc -o32 -o progname progname.s
```

to assemble and link your program. It may seem a little weird to use the compiler to do this, but `cc` is really a wrapper program that runs lots of other stages for you, not just the actual compiler. If you're curious, you can use the `-v` option to show all the commands that `cc` invokes. The `-o32` flag is important, otherwise it will link with n32 libraries and you will lose in all sorts of interesting ways.

If your program does any callouts, you may need to specify additional libraries. `cc` will automatically link in functions in `libc`, which includes `printf()`, `malloc()`, and most other common functions. If you link in functions from other libraries, you can use the `-l` option to find the library. If it's a standard system library, the toolchain will automatically grab the o32 version. If it is your own library, then you will also need to specify an additional library search path using `-L`. As with most UNIX commands, `man` is your fine source of further documentation.

3.2 More complicated stuff

You may want to run the compilation stages manually, for more flexibility. To assemble your listing into an object:

```
athena% as -o32 -G 0 filename.esp.lst
```

which generates `filename.esp.o`, a native object. At this point you can use the `nm` command to look at all of the symbols the object defines or imports. Once you have the object, link it into an executable with:

```
athena% ld -o32 -G 0 /usr/lib/crt1.o filename.esp.o -lc /usr/lib/crtn.o
```

`crt{1,n}.o` are the runtime libraries that the operating system provides, and `-lc` says to link in the standard C library. The runtime objects handle setting up your stack, interfacing with shared libraries, and other miscellaneous things. `crt1.o` calls your `main` label just like any other procedure, so your main routine should return just like any other procedure when your program is finished.

You can specify multiple objects on the `ld` command line between the run time libraries. You can write your own C routines that your Decaf/Espresso program calls by putting them in a file and doing:

```
athena% cc -o32 -c filename.c
```

while will compile and assemble the program into an object. If you want to just get the compiler's assembly output, use `-S` instead of `-c`. From there, you can link in the object just like any other object.

3.3 Calling Decaf/Espresso methods from C

So far we've examined how to call external functions from a Decaf/Espresso program. The reverse is also possible, with a bit of compiler hackery. The calling convention is the same both ways, so nothing stops a C function from calling a function in your program. Remember that Espresso functions take an implicit *this* pointer as the first argument, so you'll need to pass in an extra first argument. If the Espresso function refers to any of its fields, or calls a member function of its own

class, then the first argument will need to point to the correct place in memory to find the object's method table. Since there's no way to extract that address in the current language specification, you lose.

The language specification also requires a `Program.main()` method. If you disable this check in your compiler, you can declare `main()` in another object and have control begin there. This will work fine for Decaf compilers, but Espresso compilers will need to figure out how to do the implicit instantiation of the `Program` class and how to get the *this* pointer to `Program.main()`.