GNU Assembler Programming Tips

18–349 Embedded Real-Time Systems

1 Introduction

This document contains a collection of tips that may prove useful when writing assembly code for 18–349 labs.

2 Defining Strings in Assembly

The GNU assembler (gas) recognizes three assembler directives for defining strings. "`.string`" and "`.asciz`" both assemble string literals with null terminators (the same as C strings), whereas "`.ascii`" assembles a string literal with no null terminator. For defining a simple null-terminated string, it is usually simple enough to use the `.string` directive as follows:

```
hello_str:
        .string "Hello world!\n"
```

Certain situations (e.g,. the `write` syscall) require strings with an explicitly defined length instead of using a null terminator. One typical way of defining strings of this fashion in C would be:

```
void bar(const char *str, unsigned int len);

void foo(void) {
        const char hello_str[]       = "Hello world!\n";
        const unsigned int hello_len = sizeof(hello_str) - 1;

        bar(hello_str, hello_len);
}
```

The resulting assembly code produced by GCC is:[1]

```
        .file   "foo.c"
        .text

        .global foo
foo:
        ldr     r0, phello_str
        mov     r1, #13
        b       bar
phello_str:
        .word   hello_str

        .section        .rodata
hello_str:
        .ascii  "Hello world!\n"
```

[1]Modified to remove unnecessary directives and to add sensible labels.

This assembly code produced by GCC is cumbersome for two reasons. First, the `ldr` instruction must reference a pointer to `hello_str` which GCC stores in a literal pool immediately following the code for function `foo`. Second, GCC hard codes the length of the string as an integer in the `mov` instruction—if one wanted to update the string in this assembly by hand, the string length would have to be modified as well. Let's consider the second issue first.

2.1 Determining the Length of Strings in Assembly

The "`.`" symbol refers to the current assembling address, and the "`.set`" directive assigns a value to a symbol. Combined, one may determine the size of a section of assembly code by subtracting the current address from a label and assigning the result to a symbol. For example, the following code sets the symbol `hello_size` to the size of the string `hello_str`:

```
hello_str:
        .ascii  "Hello world!\n"
        .set    hello_size, .-hello_str
```

With this code, modifications to the `hello_str` string will automatically change the value of `hello_size` accordingly.

3 Loading Constants & Labels

The ARM ISA provides two general methods for loading 32 bit constants in a register. First, the `mov` and `mvn` instructions may be used to load an 8 bit constant shifted by an even number of bits. Second, the `ldr` instruction may be used to load an arbitrary 32 bit constant that is stored in a nearby literal pool. As illustrated in Section 2, GCC emits assembly code that uses both methods to load 32 bit constants.

3.1 The `ldr` Pseudo Opcode

To facilitate the loading of an arbitrary 32 bit constant, gas supports a special syntax of the `ldr` instruction as a pseudo opcode:

```
        ldr     reg, =constant
```

The above instruction loads an immediate 32 bit value in a register, and translates to a real `ldr` instruction, or a `mov`/`mvn` instruction.

For example, the instruction:

```
        ldr     r0, =42
```

translate to:

```
        mov     r0, #42
```

while the instruction:

```
        ldr     r0, =0xdeadbeef
```

translate to:

```
        ldr     r0, literal
        ...
literal:
        .word 0xdeadbeef
```

In addition to immediate values, the `ldr` pseudo opcode may be used to load the address of a label. For example, the assembly emitted by GCC in Section 2:

```
foo:
        ldr     r0, phello_str
        ...
phello_str:
        .word   hello_str
```

could be rewritten as:

```
foo:
        ldr     r0, =hello_str
        ...
```

which produces the same machine code as a result.

3.2 The `adr` Pseudo Opcode

Using the `ldr` pseudo opcode to load the address of a label still requires an entry in the literal pool to store the address. As an alternative, gas provides the `adr` pseudo opcode to load the address of a label by translating the instruction to a pc-relative `add` or `sub` instruction.

For example, the instructions:

```
        adr     r0, beef
        b       elsewhere
beef:
        .word   0xdeadbeef
```

translate to:

```
        add     r0, pc, #0
        b       elsewhere
beef:
        .word   0xdeadbeef
```

The `adr` pseudo opcode is restricted to using labels that are defined in the same assembly source file and section as the `adr` instruction itself. To load a label in a different file or section, the `ldr` pseudo opcode must be used instead.

4 Strings in Assembly Revisited

Using the `.set` directive along with the `ldr` & `adr` pseudo opcodes allows us to rewrite the GCC generated assembly of Section 2 more succinctly.

If `hello_str` must reside in the `.rodata` section, then the code must be rewritten using the `ldr` pseudo opcode as follows:

```
        .file   "foo.c"
        .text

        .global foo
foo:
        ldr     r0, =hello_str
        mov     r1, #hello_size
        b       bar

        .section        .rodata
hello_str:
        .ascii  "Hello world!\n"
        .set    hello_size, .-hello_str
```

If, however, `hello_str` may be moved to the `.text` section, then the code may be written using the `adr` pseudo opcode as follows:

```
        .file   "foo.c"
        .text

        .global foo
foo:
        adr     r0, hello_str
        mov     r1, #hello_size
        b       bar

hello_str:
        .ascii  "Hello world!\n"
        .set    hello_size, .-hello_str
```

Since placing `hello_str` in the `.text` section simplifies the assembly and shortens the resulting machine code, it is valid argument for placing "data" in the `.text` section of a trivial assembly program. For non-trivial programs (those written in C or that span multiple source files), separating strings into the `.rodata` section is generally preferred.