Modern Assembly Language Programming with the ARM processor

Chapter 2: GNU Assembler Syntax

Elements of Assembly

- labels
- assembler directives
- instructions and pseudoinstructions
- operands
- comments

```
        .data
msg:    .asciz  "Hello World\n" @ Define a null-terminated string

        .text
        .globl  main
        /* This is the beginning of the main() function.
           It will print "Hello World" and then return.
        */
main:   stmfd   sp!,{lr}        @ push return address onto stack
        ldr     r0, =msg        @ load pointer to format string
        bl      printf          @ printf("Hello World\n");
        mov     r0, #0          @ move return code into r0
        ldmfd   sp!,{lr}        @ pop return address from stack
        mov     pc, lr          @ return from main
```

Assembly Listing

```
ARM GAS  hello.S                   page 1

Input
 Line Address     Code
    1                        .data
    2    0000 48656C6C  msg:   .asciz "Hello World\n" @ Define null-terminated string
    2         6F20576F
    2         726C640A
    2         00
    3
    4                        .text
    5                        .globl main
    6    0000 00402DE9  main:  stmfd  sp!,{lr}    @ push return address onto stack
    7    0004 0C009FE5         ldr    r0, =msg    @ load pointer to format string
    8    0008 FEFFFFEB         bl     printf      @ printf("Hello World\n");
    9    000c 0000A0E3         mov    r0, #0      @ move return code into r0
   10    0010 0040BDE8         ldmfd  sp!,{lr}    @ pop return address from stack
   11    0014 0EF0A0E1         mov    pc, lr      @ return from main

DEFINED SYMBOLS
             hello.S:16     .text:00000000 $a
                            .data:00000000 $d
                             .bss:00000000 $d
                  .ARM.attributes:00000016 $d

UNDEFINED SYMBOLS
printf
```

Directives

- All assembler directives begin with a period (‘.’)
- The rest of the name is composed letters, usually in lower case

Statically Allocated Integer Variables

`.byte expressions`

.byte expects zero or more expressions, separated by commas. Each expression is assembled into the next byte. If no expressions are given, then the address counter is not advanced (no bytes are reserved).

`.hword expressions`
`.short expressions`

This expects zero or more expressions, and emits a 16 bit number for each. For the ARM, these are synonymous.

`.word expressions`
`.long expressions`

This directive expects zero or more expressions, separated by commas. The size of the number emitted, and its byte order, depend on what target computer the assembly is for. For the ARM, it will emit 4 bytes for each expression given. On the ARM, `.long` and `.word` are the synonymous.

Statically Allocated Character Strings

`.ascii "string"`

`.ascii` expects zero or more string literals separated by commas. It assembles each string (with no automatic trailing zero byte) into consecutive addresses.

`.asciz "string"`
`.string "string"`

`.asciz` is just like `.ascii`, but each string is followed by a zero byte. The “z” in `.asciz` stands for zero. `.string` is an alias for `.asciz`.

Statically Allocated Floating Point Numbers

`.float flonums`
`.single flonums`

This directive assembles zero or more flonums, separated by commas. The exact kind of floating point numbers emitted depends on how as is configured. On the ARM, they are standard 4-bye single precision. `.float` and `.single` are synonyms.

`.double flonums`

`.double` expects zero or more flonums, separated by commas. It assembles floating point numbers. The exact kind of floating point numbers emitted depends on how as is configured. On the ARM, they are standard 8-byte double precision.

Skipping, Filling, and Aligning

`.skip size, fill`
`.space size, fill`

This directive emits size bytes, each of value fill. Both size and fill are absolute expressions. If the comma and fill are omitted, fill is assumed to be zero. `.space` and `.skip` are equivalent.

`.align abs-expr, abs-expr, abs-expr`

Pad the location counter (in the current subsection) to a particular storage boundary.
For the ARM processor, the first expression specifies the number of low-order zero bits the location counter must have after advancement.
The second expression gives the fill value to be stored in the padding bytes. It (and the comma) may be omitted. If it is omitted.
The third expression is is also optional. If it is present, it is the maximum number of bytes that should be skipped by this alignment directive.

Aligning (Part 2)

`.balign[wl] abs-expr, abs-expr, abs-expr`

Pad the location counter (in the current subsection) to a particular storage boundary.
The first expression is the alignment request in bytes.
The second expression gives the fill value to be stored in the padding bytes. It (and the comma) may be omitted. If it is omitted.
The third expression is is also optional. If it is present, it is the maximum number of bytes that should be skipped by this alignment directive.

Controlling the Section

`.data subsection`

Tells the assembler to assemble the following statements onto the end of the data subsection numbered subsection (which is an absolute expression). If subsection is omitted, it defaults to zero.

`.text subsection`

Tells the assembler to assemble the following statements onto the end of the text subsection numbered subsection, which is an absolute expression. If subsection is omitted, subsection number zero is used.

`.bss subsection`

Tells the assembler to assemble the following statements onto the end of the bss subsection numbered subsection, which is an absolute expression. If subsection is omitted, subsection number zero is used.

Conditional Assembly

`.if absolute_expression`

> `.if` marks the beginning of a section of code which is only considered part of the source program being assembled if the argument (which must be an absolute expression) is non-zero. The end of the conditional section of code must be marked by the `.endif` directive. Optionally, you may include code for the alternative condition, flagged by the `.else` directive.

The following variants of `.if` are also supported:

`.ifdef symbol`

> Assembles the following section of code if the specified symbol has been defined.

`.ifndef symbol`

> Assembles the following section of code if the specified symbol has not been defined. Both spelling variants are equivalent.

Conditional Assembly (Part 2)

`.else`

`.else` is part of the support for conditional assembly; see section `.if` absolute expression. It marks the beginning of a section of code to be assembled if the condition for the preceding `.if` was false.

`.endif`

`.endif` is part of the as support for conditional assembly; it marks the end of a block of code that is only assembled conditionally. See section `.if` absolute expression.

Setting and Manipulating Symbols

```
.equ symbol, expression
.set symbol, expression
```

This directive sets the value of `symbol` to `expression`.

```
.equiv symbol, expression
```

The `.equiv` directive is like `.equ` and `.set`, except that the assembler will signal an error if symbol is already defined.

```
.global symbol
.globl symbol
```

This directive makes the symbol visible to the linker. If you define `symbol` in your partial program, its value is made available to other partial programs that are linked with it. Otherwise, `symbol` takes its attributes from a symbol of the same name from another file linked into the same program.

Macros (1)

`.include "file"`

This directive provides a way to include supporting files at specified points in your source program. The code from file is assembled as if it followed the point of the `.include`. When the end of the included file is reached, assembly of the original file continues. You can control the search paths used with the '`-I`' command line parameter. Quotation marks are required around `file`.

This is a good way to include files containing macros and other definitions. It is similar to including header files in C and C++.

Macros (2)

`.macro`

The commands `.macro` and `.endm` allow you to define macros that generate assembly output. For example, this definition specifies a macro `enum` that puts a sequence of numbers into memory, using a recursive macro call to itself:

```
        .macro  enum from=0, to=5
        .long   \from
        .if     \to-\from
        enum     "(\from+1)",\to
        .endif
        .endm
```

With that definition, ‘`enum 0,5`’ is equivalent to this assembly input:

```
        .long   0
        .long   1
        .long   2
        .long   3
        .long   4
        .long   5
```

Macros (3)

`.macro macname`
`.macro macname macargs ...`

Begin the definition of a macro called `macname`. If your macro definition requires arguments, specify their names after the macro name, separated by commas or spaces. You can supply a default value for any macro argument by following the name with ‘`=deflt`’. For example, these are all valid `.macro` statements:

`.macro comm`

Begin the definition of a macro called comm, which takes no arguments.

`.macro plus1 p, p1`
`.macro plus1 p p1`

Either of these two statements begins the definition of a macro called `plus1`, which takes two arguments; within the macro definition, write ‘`\p`’ or ‘`\p1`’ to evaluate the arguments.

Macros (4)

`.macro reserve_str p1=0 p2`

> Begin the definition of a macro called ’`reserve_str`’, with two arguments. The first argument has a default value, but not the second. After the definition is complete, you can call the macro either as ‘`reserve_str x,y`’ (with ‘ `\p1`’ evaluating to `x` and ‘`\p2`’ evaluating to `y`), or as ‘`reserve_str ,y`’ (with ‘`\p1`’ evaluating as the default, in this case ‘0’, and ‘`\p2`’ evaluating to `y`).

When you call a macro, you can specify the argument values either by position, or by keyword. For example, ‘`reserve_str 9,17`’ is equivalent to ‘`reserve_str p2=17, p1=9`’.

Macros (5)

`.endm`

Mark the end of a macro definition.

`.exitm`

Exit early from the current macro definition.

`\@`

The assembler maintains a counter of how many macros it has executed. You can copy that number to your output with ‘`\@`’, but only within a macro definition.