An Introduction to the GNU Assembler

The GNU Assembler, part of the GNU Compiler Tools software suite, is the assembler used in the Digital Systems Laboratory to convert ARM assembly language source code into binary object files. This assembler is extensively documented in the *GNU Assembler Manual* (which can be found on your CD-ROM in the *gnutools/doc* directory). This Introduction is a summary of that manual, specifically for the Laboratory.

Example and Template Files

The *examples* directory and its subdirectories on your CD-ROM contain many examples of assembly language programs that you can study. And you are encouraged to do so! Please see the end of this document for more details.

One of the subdirectories in *examples* is *templates*. You are strongly encouraged to use the files provided there as a starting point in writing your computer programs. In particular, the file *template.s* should be used for all of your ARM assembly language programming. This file, stripped of most of its comments, appears below:

```
        .text                   ; Executable code follows

_start: .global _start          ; "_start" is required by the linker
        .global main            ; "main" is our main program

        b       main            ; Start running the main program

main:                           ; Entry to the function "main"

; Insert your code here

        mov     pc,lr           ; Return to the caller

        .end
```

Invoking the Assembler

You can assemble the contents of any ARM assembly language source file by executing the **`arm-elf-as`** program. This can be invoked as:

```
arm-elf-as -marm7tdmi --gdwarf2 -o filename.o filename.s
```

Naturally, replace *`filename`* with whatever is appropriate for your project. The command-line option **`-marm7tdmi`** instructs the GNU Assembler that your target CPU is the ARM7TDMI (ARMv4T architecture). The option **`--gdwarf2`** requests the assembler to include information in the output file that is helpful for debugging—it does not, incidentally, alter your program in any way. Chapters 1 and 2, as well as section 8.4, of the *GNU Assembler Reference* list other command line options.

For larger projects, you should modularise your code into multiple source files. Each source file (which has the extension *.s*) is then assembled using a command line similar to the one shown.

Once you have assembled all of your source files into binary object files (with the extension *.o*), you use the GNU Linker to create the final executable (extension *.elf*). This is done by executing:

```
arm-elf-ld -o filename.elf filename.o
```

Once again, replace *`filename`* with whatever is appropriate for your project. If you are using multiple source files (and hence multiple object files), replace *`filename.o`* with a list of all object files in your project.

Typing these commands again and again becomes rather tedious (even though the Unix shell allows you to use the Up Arrow key to retrieve previous command lines). For this reason, a template Makefile has been supplied that you can modify to suit your own purposes (it can be found as *examples/templates/Makefile.template-asm* on the CD-ROM). Once you have modified this file (and renamed it to *Makefile*), all you need to do to recreate the executable is type:

```
make
```

Assembly Language Syntax

The GNU Assembler is actually an assembler that can target many different CPU architectures, not just the ARM. For this reason, its syntax is slightly different from other ARM assemblers; the GNU Assembler uses the same syntax for all of the 45-odd CPU architectures that it supports.

Assembly language source files consist of a sequence of statements, one per line. Each statement has the following format, each part of which is optional:

```
label:  instruction     ; comment
```

A *label* allows you to identify the location of the program counter (ie, its address) at that point in the program. This label can then be used, for example, as a target for branch instructions or for load and store instructions. A label can be any valid symbol, followed by a colon "**:**". A valid symbol, in turn, is one that only uses the alphabetic characters **A** to **Z** and **a** to **z** (case does matter, ie, is significant), the digits **0** to **9**, as well as "**_**", "**.**" and "**$**". Note, however, that you cannot start a symbol with a digit. (For more information, please see sections 3.4 and 5.3 of the *GNU Assembler Reference*).

A *comment* is anything that follows a semicolon "**;**".[1] Everything after a semicolon—except when it appears in a string—is ignored to the end of that line. C-style comments (using "**/***" and "***/**") are also allowed.

The *instruction* field is the real meat of your program: it is any valid ARM assembly language instruction that you care to use. It also includes the so-called pseudo-operations or *assembler directives:* "instructions" that tell the assembler itself to do something. These directives are discussed in greater detail below.

Assembler Directives

All assembler directives have names that begin with a full-stop "**.**". These are discussed in detail in Chapter 7 of the *GNU Assembler Reference;* the list of directives presented here (in alphabetical order) are the more useful ones that you may need to use in your assembly language programs.

.align

Insert from zero to three bytes of 0x00's so that the next location will be on a 4-byte (word) boundary. Remember, in particularly, that the ARM microcontroller must always access words (32-bit quantities) on a word boundary. As an example, the following three lines will insert eight bytes into the object file output, assuming that the first line is already on a word boundary:

```
.byte   0x55        ; inserts the byte 0x55
.align              ; inserts three alignment bytes: 0x00 0x00 0x00
.word   0xAA55EE11  ; inserts the bytes 0x11 0xEE 0x55 0xAA (LSB order)
```

[1] At least when targeting the ARM microcontroller, although this so-called *comment character* may be different for other architectures.

By the way, please note that the **0x** prefix indicates the number is in hexadecimal. See the section on Expressions, later in this document, for more details.

This assembler directive has optional arguments that are not documented here; if you need to use them, you are encouraged to use the **.balign** directive instead. See section 7.3 of the *GNU Assembler Reference* for more information.

.ascii "*string*" …

Insert the string literal into the object file exactly as specified, with no trailing NUL character. More than one string may be specified if they are separated by commas. As an example, the following line inserts three bytes into the object file output:

```
.ascii  "JNZ"          ; inserts the bytes 0x4A 0x4E 0x5A
```

.asciz "*string*" …

Insert the string literal into the object file, followed by a NUL character (a 0x00 byte). As with **.ascii**, more than one string may be specified if separated by commas. As an example, the following line inserts four bytes (not three) into the object file output:

```
.asciz  "JNZ"          ; inserts the bytes 0x4A 0x4E 0x5A 0x00
```

See also the **.ascii** directive for one that does not insert the terminating NUL character.

.byte *expression* …

Insert the (8-bit) byte value of the expression into the object file. More than one expression may appear, if separated by commas. As an example, the following lines insert 5 bytes into the object file output:

```
.byte   64, 'A'           ; inserts the bytes 0x40 0x41
.byte   0x42              ; inserts the byte 0x42
.byte   0b1000011, 0104   ; inserts the bytes 0x43 0x44
```

See also the **.hword** and **.word** assembler directives.

.data

Switch the destination of following statements into the data section of the final executable. All executable programs have at least two sections called **.text** and **.data**. Directives having those names allow you to switch back and forth between the two sections.[2]

Technically speaking, only executable code is meant to appear in a **.text** section (although read-only constants are fine as well) and only read/write data is meant to appear in a **.data** section; this, however, is not enforced by the GNU Assembler. Chapter 4 of the *GNU Assembler Reference* deals with this topic in greater depth.

.end

Mark the end of this source code file; *everything* after this directive is ignored by the assembler. Quite optional but highly recommended.

---

[2] You might find the following analogy helpful… Imagine that the executable file is a factory containing at least two output bins. Bytes (the goods being produced) come down a single conveyer belt and are dropped into a particular bin (a section). The **.text** and **.data** directives, then, are like workers that move the end of the conveyer belt back and forth between the two bins labelled **.text** and **.data**.

.equ *symbol, expression*

Set the value of *symbol* to *expression*. This assembler directive can also be specified as **.set** or as "**=**". The following three lines are exactly identical, and set the value of adams to 42:

```
          .equ    adams, (5 * 8) + 2
          .set    adams, 0x2A
adams     =       0b00101010
```

.extern *symbol*

Specify that *symbol* is defined in some other source code file (ie, module). This directive is optional as the assembler treats any symbols that are undefined as external. It is, nevertheless, recommended as part of documenting a source code file.

.global *symbol*

Specify that *symbol* is to be made globally visible to all other modules (source code files) that are part of the executable, and visible to the GNU Linker. The symbol **_start**, which is required by the GNU Linker to specify the first instruction to be executed in a program, must always be a global one (and only present in one module, of course).

.hword *expression* ...
.2byte *expression* ...

Insert the (16-bit) half-word value of the expression into the object file. More than one expression may appear, if separated by commas. The directive **.2byte** can be used as a synonym. As an example, the following lines insert 8 bytes into the object file output:

```
.hword  0xAA55, 12345   ; inserts the bytes 0x55 0xAA 0x39 0x30
.2byte  0x55AA, -1      ; inserts the bytes 0xAA 0x55 0xFF 0xFF
                        ; Least Significant Byte (LSB) ordering assumed
```

Remember that the ARM microcontroller must access all half-word quantities on 16-bit boundaries. In other words, it cannot access a half-word memory location if that location is at an odd address. See the **.align** directive for a solution.

See also the **.byte** and **.word** assembler directives.

.include "*filename*"

Insert the contents of *filename* into the current source file, as if that file had been typed into the current file directly. This is exactly the same as C's use of #include, and is for the same reason: it allows you to include header files full of various definitions. By the way, beware of having **.end** in the included file: the GNU Assembler will stop reading everything after that directive!

.ltorg

Insert the literal pool of constants at this point in the program. The literal pool is used by the ldr =, adr and adrl assembly language pseudo-instructions and is specific to the ARM. Using this assembler directive is almost always optional, as the GNU Assembler is smart enough to figure out when and where to put any literal pool.

.set *symbol, expression*

This is a synonym for the **.equ** assembler directive; it is a personal preference as to which you use (but be consistent!).

.skip *expression*

Skip *expression* bytes in the object file output. The bytes so skipped should be treated as unpredictable in value, although they are often initialised to zero. This directive is useful for declaring uninitialised variables of a certain size. As an example, the following three lines declare three variables, two that are pre-initialised, one (buffer) that is not:

```
head_ptr:  .word  0    ; Head pointer to within buffer (initially zero)
tail_ptr:  .word  0    ; Tail pointer to within buffer (initially zero)
buffer:    .skip  512  ; Buffer of 512 bytes, uninitialised
```

See also the **.ascii**, **.asciz**, **.byte**, **.hword** and **.word** directives for initialised storage.

.text

Switch the destination of following statements into the text section of the final executable. This is the section into which assembly language instructions should be placed. See the **.data** directive for a somewhat longer explanation.

.word *expression* ...
.4byte *expression* ...

Insert the (32-bit) word value of the expression into the object file. More than one expression may appear, if separated by commas. The directive **.4byte** can be used as a synonym. As an example, the following lines insert 8 bytes into the object file output:

```
.word   0xDEADBEEF  ; inserts the bytes 0xEF 0xBE 0xAD 0xDE
.4byte  -42         ; inserts the bytes 0xD6 0xFF 0xFF 0xFF
                    ; Least Significant Byte (LSB) ordering assumed
```

Remember that the ARM microcontroller must access all word quantities on 32-bit boundaries. In other words, it cannot access a word memory location if the lowest two bits of that address are non-zero. See the **.align** directive for a solution.

See also the **.byte** and **.hword** assembler directives. Those coming from a background in other processors need to remember that, in ARM nomenclature, a word is a 32-bit quantity, not a 16-bit one.

Expressions

Many ARM assembly language statements, as well as assembler directives, require an integer number of some sort as an operand. For example, `MOV R0,#1` requires the number "1". An expression may appear any place a number is expected.

At the most basic level, an expression can be a simple integer number. This number can be expressed in decimal (with the usual notation), in hexadecimal (with a **0x** or **0X** prefix), in octal (with a leading zero) or in binary (with a **0b** or **0B** prefix). It can also be expressed as a character constant, surrounded or preceded by single quotes. Thus, the following six lines all load register R0 with the same value, 74:

```
mov  r0,#74          ; decimal number 74
mov  r0,#0x4A        ; hexadecimal number 0x4A (0X4A and 0x4a are also OK)
mov  r0,#0112        ; octal number 0112
mov  r0,#0b1001010   ; binary number 0b1001010 (0B1001010 is also OK)
mov  r0,#'J'         ; character constant "J" (preferred syntax)
mov  r0,#'J          ; character constant "J" (alternative syntax)
```

Of course, a good programmer will define a symbol with the appropriate value and use that instead:

```
.set letter_J, 'J'    ; This is a contrived example!
mov  r0, #letter_J
```

Character constants can also contain certain backslash escape sequences, similar to the C programming language. See section 3.6.1 of the *GNU Assembler Reference* for more details.

Apart from simple integer numbers, expressions can also look like standard mathematical and logical expressions expressed in the C language. These are described in detail in Chapter 6 of the *GNU Assembler Reference*. Some examples are:

```
.set  ROM_size,  128 * 1024          ; 131072 bytes (128KB)
.set  start_ROM, 0xE0000000
.set  end_ROM,   start_ROM + ROMsize ; 0xE0020000
.set  bm1,       0b11001101          ; Binary bit-mask (hex 0xCD)

.set  val1, -2 * 4 + (45 / (5 << 2)) ; -6 (in two's complement)
.set  val2, ROM_size >> 10           ; 128 (131072 shifted right by 10)
.set  val3, bm1 | 0b11110000         ; bm1 OR 0b11110000 = 0b11111101
```

Expressions can, of course, contain previously-defined labels, as illustrated above. Such expressions can yield absolute values or relative values. *Absolute values* don't depend on their position in the final executable (ie, they are position-independent); they are a simple numeric constant, such as those in the examples above. *Relative values*, on the other hand, are relative to some address, such as the start of the **.data** section. These values are only fixed into place (ie, assigned a value) by the linker when the final executable is created, not by the assembler.

Relative values are mainly used for offset calculations, and can only use the "**+**" and "**-**" operators. For example:

```
    .text

code_start:
    XXX           ; Many assembly language instructions...
code_end:

    .set  instr_size,  4    ; instructions are all 4 bytes long on ARM
    .set  code_length, code_end - code_start
    .set  first_instr, code_start
    .set  instr_11th,  code_start + 10 * instr_size
    .set  instr_10th,  instr_11th - instr_size
```

The symbol `code_length` is set to an absolute number of bytes (the difference between `code_end` and `code_start`). The other symbols, `first_instr`, `instr_11th` and `instr_10th`, are all relative to the start of the **.text** section: their exact value is only set when the final executable is created.

In general, the following rules apply to relative expressions:

relative + *absolute* → *relative*
absolute + *relative* → *relative*
relative – *absolute* → *relative*
relative – *relative* → *absolute*

All other expressions involving relative values are not allowed (except, of course, for the simple case of a symbol by itself). Note also that relative symbols must be in the same section: you cannot find the difference between a label in the **.text** section and one in the **.data** section, for example.

You may find more information about relative expressions, if you need it, in section 4.1 of the *GNU Assembler Reference*.

Example Files

As already mentioned, the *examples* directory and its subdirectories on your CD-ROM contain many examples of assembly language programs. These example files illustrate various aspects of the GNU Assembler for the ARM microcontroller. You are encouraged to study these examples—at the very least, you should quickly look through them!

In particular, the *examples/intro* directory contains the following example files; these files should be studied in the order presented:

simple.s	A simple ARM assembly language program, with which to start
subr.s	Simple subroutines (function calls)
values.s	Load constant values into registers, with `ldr =`
pseudo.s	More information about pseudo-instructions for the ARM
jumptbl.s	Multi-way branches and pointers to functions
wordcopy.s	Copy an array of words, stored in the `.data` section
blockcopy.s	Copy an array en-masse, with stack pointer initialisation
copy.s	Copy a NUL-terminated string to a buffer (assembler module)
strcopy-a.s	String copy using multiple source files, using *copy.s*
strcopy-c.c	Mixing C and assembler for string copy, using *copy.s*

You can create the associated executable files by copying all files in that directory to a temporary one, then use **`make`**:

```
mkdir -p ~/intro              # Create a directory to hold the files
cd /mnt/cdrom/examples/intro  # Assumes CD-ROM is mounted on /mnt/cdrom
cp * ~/intro                  # Copy the files
cd ~/intro                    # Change to that directory
chmod 644 *                   # Make the files read/writable
make all                      # Create the executables
```

You can use the simulator provided by either **`arm-elf-gdb`** or **`arm-elf-insight`** to actually run the executable files; see *An Introduction to the GNU Debugger* for more details.

More Information

You can find more information about the GNU Assembler in the *GNU Assembler Reference*, also called *Using AS*. This 200-odd-page document is fairly comprehensive, although not particularly user-friendly.[3] It can be found in the *gnutools/doc* directory on your CD-ROM.

The definitive reference is the actual source code to the GNU Assembler. You can find this on the CD-ROM in the file *gnutools/src/binutils*-version.*tar.gz* for some version number. After unpacking this archive (and applying the appropriate patch file), try browsing the source code files in the *gas* subdirectory.

[3] If you think you can do better, you can always try! That is one of the advantages of having a program with free access to the source code…