The ARM Assembly Language – A Short Introduction

© R. Nigel Horspool and Micaela Serra, 2010
Department of Computer Science
University of Victoria

1. Preliminaries

This document provides only a barest-bones description of those ARM7TDMI assembler instructions and their addressing modes that are likely to be needed in an introductory course on assembly language and computer architecture. Because the ARM instruction set is common across the range of ARM processors, assembly code that runs on an ARM7 processor also runs on other ARM family processors.

The ARM architecture is based on RISC (Reduced Instruction Set Computer) ideas. The main consequence is that all computations have registers as their operands. That is, an ARM program must load values into registers, it manipulates those registers, and then it may store results back to memory. The architecture implements binary integers in 32 bit sizes in both signed and unsigned formats. It also implements 64 bit floating-point numbers. Addresses are 32 bits wide. We use the following names for the different size storage units: *byte* for 8-bit numbers, and *word* for 32-bit numbers.

Supervisor mode instructions, floating point instructions and other highly specialized instructions are not described in this document. Also note that GNU assembler syntax is used in this document – not the assembly language syntax used in the tools developed and distributed by ARM Ltd. The GNU syntax has changed between different releases of the Gnu assembler. We are assuming the assembler syntax supported by release 2.16.91, distributed as the program `arm-none-eabi-as.exe` with the 2006q1-6 release of the CodeSourcery toolchain (see the reference section for the CodeSourcery URL).

The ARM7TDMI processor has two operating states: the *ARM state*, where the processor executes 32-bit, word-aligned ARM instructions, and the *Thumb* state, where the processor executes 16-bit, halfword-aligned Thumb instructions. This introduction only considers the ARM state.

The normal twos-complement representation is used for binary numbers. There are two possible orderings for the bytes within a word, and the ARM supports both of them. With the big-endian format, the first byte in a word holds the most significant 8 bits, and so on in order for another three bytes. With little endian format, the ordering of the bytes in memory is reversed. (However, the ordering of the bits within a byte is the same for both big-endian and little-endian formats.) The ARM supports both formats, using a mode bit to select the desired format. If the byte ordering has any effect on any examples in this document then little-endian format (the same as used by Intel architecture processors) has been used.

The bibliography at the end of this document lists books which will provide a more detailed description of the ARM instruction set, its architecture, and assembly language programming for the ARM.

2. A sample ARM program

To demonstrate some of the major items in ARM programming and to give some feel for the Assembly language, it is useful to start with a short program. The code simply adds the elements of an array called "Vec" using a loop with counter from "Size" (which is 3 in this case).

```
1           .text
2           .global _start
3     _start:
4           LDR       R1,=Size   @ R1 is assigned the address of Size
5           LDR       R1,[R1]    @ R1 = Size
6           LDR       R2,=Vec    @ R2 is assigned the address of Vec
7           MOV       R0,#0      @ R0 = 0 for sum of elements
8     LOOP: LDR       R3,[R2]    @ R3 = Vec[i]
9           ADD       R0,R0,R3   @ R0 = R0 + Vec[i]
10          ADD       R2,R2,#4   @ R2 moves to the next element of Vec
11          SUB       R1,R1,#1   @ the counter for Size is decreased
12          CMP       R1,#0      @ if counter = 0, end of Vec
13          BNE       LOOP       @ else repeat the loop
14          LDR       R1,=Tot    @ R1 is assigned the address of Tot
15          STR       R0,[R1]    @ Tot = R0
16          SWI       0x11       @ end execution
17          .data
18          .align               @ Align on a word boundary
19    Size: .word     3
20    Vec:  .word     3, -1, 2
21    Tot:  .skip     4
22          .end
```

The line numbers on the left are shown simply here for referencing. Lines 1-3, 17-22 contain *assembler directives* which give the Assembler (or the Linker/Loader) some useful information in order to process the source code (see section 5). Line 1 indicates the start of the program portion containing executable instructions, while line 17 denotes the portion of the program containing data allocation directives. Line 3 denotes the starting label for execution, while line 2 makes that label into a global variable passed to the Linker/Loader. Lines 19-21 show the allocation of integer variables. In line 19, a 32-bit integer named "`Size`" is declared and initialized to 3. In line 20, three consecutive 32-bit integers are allocated, forming an array named "`Vec`", where the elements are initialized to `Vec`[0]=3, `Vec[`1]=−1 and `Vec`[2]=2. In line 21, "`Tot`" is declared as an uninitialized storage area of 4 bytes. Tot can be used as an integer variable by the program. Line 22 denotes the end of the source code, while line 18 forces alignment on a word boundary for the following data allocation (this directive is often not necessary).

The program starts by loading the address of variable "`Size`", which contains the size of the array, and then loading its contents into register R1. R1 is subsequently used as a decrementing counter for the loop. Similarly in line 6, the address of the array "`Vec`" is assigned to R2, which is then used as a pointer to step through the elements of the array. In line 7, register R0 is initialized to 0. (The details for Load and Store instructions and the available addressing modes are in section 4.5, while data movements and arithmetic instructions are described in section 4.3 and 4.1.) In line 8, the pointer to an element of the array is used as index to load the contents of that element into a register, R3, in order to add it to the cumulative total, as done in line 9. Using arithmetic on the address contained in R2 (a pointer), line 10 increases the index into the array to advance to the next element. In line 11, R1 is decremented (the counter based on the size of the array) and in line 12 it is compared to "0" to check for the end of the iteration. (The comparison instructions and the logical instructions are described in sections 4.7and 4.2.) A branch instruction in line 13 checks whether R1 is or is not equal to 0. If R1 is not equal to 0, control is returned to line 8 and the loop is executed once more. Otherwise, lines 14-15 are executed, where the address of "`Tot`" is assigned to

R1 and the accumulated sum of the elements from R0 is then stored into `Tot`. (Branching is explained in section 4.8.)

3. Registers

The ARM7TDMI has 16 general purpose user registers; the registers are named *R0, R1, ... R15*. The case of the letter is ignored by the assembler, so the names *r0, r1, ... r15* are equally acceptable. Each of these registers may be used to hold a 32-bit integer or a 32-bit address. Three of the registers have special uses, these are *R12, R13, R14* and *R15*. To emphasize their special uses, they have alternative names which are as follows:

`FP`	Frame Pointer	R12 is used in the subroutine linkage conventions to hold the address of a subroutine's *frame* on the stack. The frame contains storage for local variables and for registers which need to be saved into memory.
`SP`	Stack Pointer	R13 is by convention used as the Stack Pointer
`LR`	Link Register	R14 is used for subroutine linkage, because it receives the return address when the BL instruction is executed. At other times it can be used as a general-purpose register.
`PC`	Program Counter	R15 is the Program Counter.

These names can also be written in assemby language in lower-case as *fp, sp, lr* and *pc*.

There is one *Current Program Status Register (CPSR)* and its bit fields are shown in Figure 1. The *CPSR* is used to hold information about the result computed by a recently executed instruction; to control the enabling and disabling of interrupts; and to set the processor operating mode.

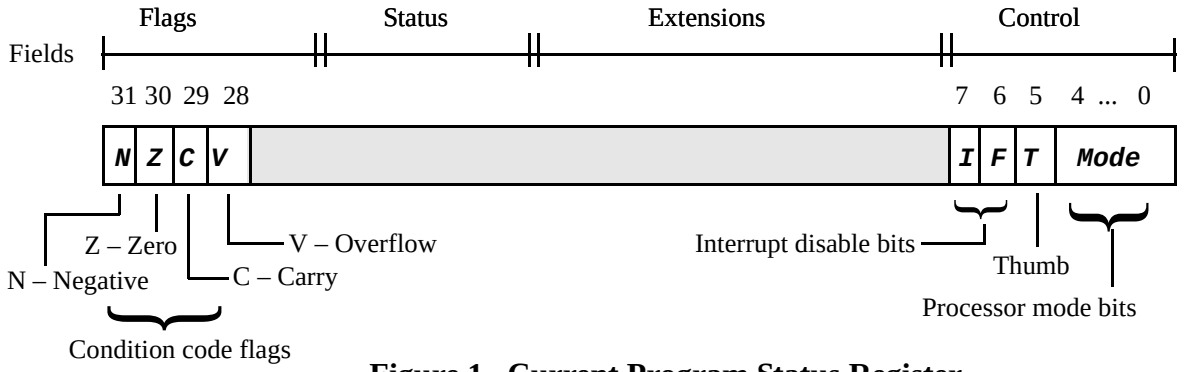

Figure 1. Current Program Status Register

4. The Instruction Set

The most useful and commonly used instructions are covered in this first section. A more complete list of ARM instructions appears in the Appendix.

The tables that provide instruction summaries, below, have three columns. The first column provides the assembly language mnenomic as it must appear in an assembly language file (a file whose name has the '.s' suffix) for pro-

cessing by an ARM assembler program. The mnemonic includes an optional character enclosed in braces. For example, the entry '`ADD{S}`' represents two different mnemonics, namely '`ADD`' and '`ADDS`'. All instruction mnemonics are accepted by the assembler either in capital or small letters – thus '`ADD`' and '`add`' are equivalent. The second column of the table indicates the operands that may be provided to an instruction. The notations *Rm*, *Rn*, *Rs* and *Rd* each represent one of the 16 general purpose registers, and <*Oprnd2*> represents one of several possibilities. These possibilities are listed in Table 6 and explained in Section 4.4.1. Example instructions in this section use only two of the possibilities, namely a register or an immediate constant. The third column of the table summarizes the effect of the instruction, using a notation similar to a programming language.

4.1 Integer Arithmetic Instructions

The operations of addition, subtraction and multiplication are available in a direct form, plus some variations including the Carry bit. The usual integer arithmetic instructions are listed in Table 1 and additional ones are in Table 2.

Table 1. Integer Arithmetic Instructions

`ADD{S}`	*Rd,Rn,<Oprnd2>*	Rd = Rn + Oprnd2 {CPSR}
`SUB{S}`	*Rd,Rn,<Oprnd2>*	Rd = Rn – Oprnd2 {CPSR}
`MUL{S}`	*Rd,Rm,Rn*	Rd = Rm * Rn {CPSR}

The notation `ADD{S}` is used to represent two mnemonics: `ADD` and `ADDS`. Both instructions perform the same operation, that of adding *Rn* to *Oprnd2* and copying the result into *Rd*. However, the `ADDS` version has the additional effect of testing the result stored in *Rd* and setting the four condition code bits in the CPSR accordingly. The notation {CPSR} in the third column indicates that effect. Exactly how the condition code bits are set and subsequently used is explained in Sections 4.7 and 4.8. The second operand of each instruction, denoted as <*Oprnd2*> can be a register, an immediate constant, or a register with an additional shift/rotate addressing mode, as listed in Table 6.

In the multiplication instruction, the destination operand, *Rd*, must be different from either of the two other operands, and all operands must be registers. No division instruction is provided on the ARM.

Examples:

```
ADD   r1,r2,r5        @ r1=r2+r5
SUB   r2,r2,r5        @ r2=r2-r5
ADDS  r0,r2,#23       @ r0=r2+23 and CPSR is set
SUB   r4,r5,#0x17     @ r4=r5-23 (23 = 17_16)
SUBS  r5,r5,r2        @ r5=r5-r3 and CPSR is set
MUL   r1,r2,r5        @ r1=r2*r5
MULS  r2,r5,r5        @ r2=r5*r5 and CPSR is set
```

The other arithmetic instructions follow a similar format. The `ADC` instruction operates similarly to `ADD` while also adding the current *Carry* bit from the *CPSR* to the result. The `SBC` instruction operates similarly to `SUB` while subtracting the negated current *Carry* bit from the *CPSR* to the result. `RSB` and `RSC` perform reverse subtractions, while `MLA` combines multiplication and addition and is the only instruction with 4 operands, all of which must be in registers. `CLZ` gives the count of leading zeros present in a register value, and `SWP` swaps the contents of two registers.

Table 2. Additional Integer Arithmetic Instructions

ADC{S}	*Rd,Rn,<Oprnd2>*	Rd = Rn + Oprnd2 + Carry {CPSR}
SBC{S}	*Rd,Rn,<Oprnd2>*	Rd = Rn - Oprnd2 - NOT Carry{CPSR}
RSB{S}	*Rd,Rn,<Oprnd2>*	Rd = Oprnd2 - Rn {CPSR}
RSC{S}	*Rd,Rn,<Oprnd2>*	Rd = Oprnd2 - Rn - NOT Carry {CPSR}
MLA{S}	*Rd,Rm,Rs,Rn*	Rd = Rm * Rs + Rn {CPSR}
CLZ	*Rd,Rm*	Rd = number of leading zeroes in Rm
SWP	*Rd,Rm*	temp = Rn; Rn = Rm; Rd = temp

Examples:

```
ADC   r1,r2,r5        @ r1=r2+r5+Carry
RSB   r1,r2,r5        @ r1=r5-r2
MLA   r1,r2,r3,r5     @ r1=r2*r3+r5
```

4.2 Logical Instructions

The logical instructions are shown in Table 3. The second operand, denoted as <*Oprnd2*> can be a register, an immediate constant, or a register with any additional shift/rotate addressing mode, as listed in Table 6. If the optional code letter 'S' is included, the condition code bits in the *CPSR* are set according to the result stored in register *Rd*.

The operation written as AND denotes a *bitwise logical and* of the bit patterns in the two operands. Bit position *i* of the destination register *Rd* is equal to 1 only if bit position *i* of *Rn* and bit position *i* of *Oprnd2* are both 1; otherwise that bit position holds 0. The operation written as *OR* denotes a *bitwise logical or*; here the result in bit position *i* of *Rd* is 1 if either *Rn* or *Oprnd2* hold a 1 in bit position *i*, otherwise the result is 0. The operation written as *EOR* is short for exclusive or; the result in bit position *i* of *Rd* is 1 if one, but not both, of bit positions *i* in *Rn* and *Oprnd2* hold 1, otherwise the result is 0. The logical complement of each bit is not in this list and given later (see the MVN instruction). Finally, the operation written as *BIC* operation is short for *bit clear.* The *BIC* instruction produces a result equivalent to copying *Rn* to *Rd* and then, for every bit position *i* in *Oprnd2* which holds a 1, clearing the corresponding bit position in *Rd* to 0 (As the explanation in the table show, BIC is equivalent to performing an AND between Rn and an inverted version of *Oprnd2* where all the bits are flipped – zeros become ones and ones become zeros.) The Assembler may convert BIC into AND where appropriate.

Table 3. Logical Instructions

AND{S}	*Rd,Rn,<Oprnd2>*	Rd = Rn AND Oprnd2 {CPSR}
EOR{S}	*Rd,Rn,<Oprnd2>*	Rd = Rn EXOR Oprnd2 {CPSR}
ORR{S}	*Rd,Rn,<Oprnd2>*	Rd = Rn OR Oprnd2 {CPSR}
BIC{S}	*Rd,Rn,<Oprnd2>*	Rd = Rn AND NOT Oprnd2 {CPSR}
NOP	No Operation	R0 = R0

Examples:

```
@ Let r1 = FA23045B16 before each instruction is executed below:

AND   r1,r1,#0xFF      @ mask off the most significant 24 bits
                       @ Afterwards, r1 = 0000005B

EOR   r1,r1,#0xFF000000     @ complement the upper 8 bits
                            @ Afterwards, r1 = 0523045B

@ testing for even numbers
ANDS  r1,r1,#1         @ Afterwards, r1 = 0 and r0 is therefore even

@ one can build constants too big for the 8-bit immediate field, as in:
MOV   r1,#0xFF
ORR   r1,r1,#0xFF00    @ Afterwards, r1 = 0xFFFF
```

4.3 Data Movement Instructions

The instructions `MOV` and `MVN` are used to move data between registers or to assign a constant. The destination operand, *Rd*, must be a register. The second operand, denoted as *<Oprnd2>* can be a register, an immediate constant, or a register with an additional shift/rotate addressing mode, as listed in Table 6. In this section only examples with registers and immediate constants are presented, while the shifted operands are discussed in section 4.4. If the immediate constant in the second operand is not rotated, its value is restricted to the 8 bit range: 0 to 255. If the optional code letter 'S' is included, the condition code bits in the *CPSR* are set according to the result after execution.

Table 4. Data Movement Instructions

`MOV{S}`	*Rd,<Oprnd2>*	Rd = Oprnd2 {CPSR}
`MVN{S}`	*Rd,<Oprnd2>*	Rd = NOT Oprnd2 {CPSR}

Examples:

```
MOV   r1,r2       @ r1 = r2
MVN   r2,r5       @ r2 = -r5+1 (the logical NOT r5)
MOV   r0,#23      @ r0 = 23
MOV   r4,#0x17    @ r4 = 1716 or 2310
MOVS  r3,r5       @ r3 = r5 and CPSR is set
```

4.4 Shift and Rotation

There are no explicit shift and rotate instructions, but the addressing mode for *<Oprnd2>* performs all these operations as implemented by the shifter within the ALU. As shown in Figure 2, describing the input and output operands to the ALU, the second operand *Rm* can be pre-processed through the shifter before being passed to the ALU. There are five shift operations which can be applied as part of the addressing mode of the second operand, and they are listed in Table 5.

The ASR performs an *arithmetic* shift right, which implies that the shift right preserves the sign by duplicating the sign bit in the leftmost position. Its arithmetic effect is to perform *signed* division by 2^n, where 'n' is the number of bit positions for the shift. LSR performs a logical shift right and the leftmost bits arefilled by a '0'. It can be seen to

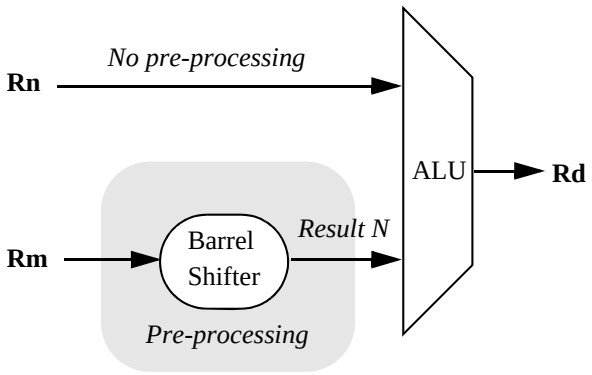

Figure 2. ALU Operations – The Barrel Shifter

Table 5. Shift and Rotate Modifiers

LSL	logical shift left by *n* bit positions
LSR	(unsigned) logical shift right by *n* bit positions
ASR	(signed) arithmetic shift right by *n* bit positions
ROR	(unsigned) rotate right by *n* bit positions
RRX	(unsigned) shift right by 1 and C bit moved to 31st bit position

perform *unsigned* division by 2^n, where 'n' is the number of bit positions for the shift. LSL performs a logical shift left and the rightmost bits are filled by a '0'; note that LSL can also be seen to perform an arithmetic shift left as the effect is the same, and no explicit arithmetic shift left is provided. (However ASL is accepted and aliases to LSL for most assemblers). Thus a shift left by 'n' bit positions performs multiplication by 2^n. Only a rotate right operation is provided, ROR, because rotation right by (*32* - *n*) places is equivalent to a rotate left by *n* places. The RRX instruction, rotate right extended, only moves the bits by 1 position right, with bit 0 going into the Carry bit, while bit 31 is assigned the previous Carry bit value. Figure 3 shows the behaviour of all shifts and rotations schematically for clarity.

Combining a MOV instruction with the appropriate addressing mode with shift or rotate for the second operand achieves the same functionality as an explicit shift or rotate instruction.

Examples:

```
MOV   r1,r1,LSL #2     @ r1=r1 shifted left by 2 bit positions,
                       @  that is, r1 = r1 << 2 = r1*4

MOV   r1,r1,ASR #3     @ r1=r1 shifted right by 3 bit positions,
                       @  that is, r1 = r1 >> 3 = r1/8
```

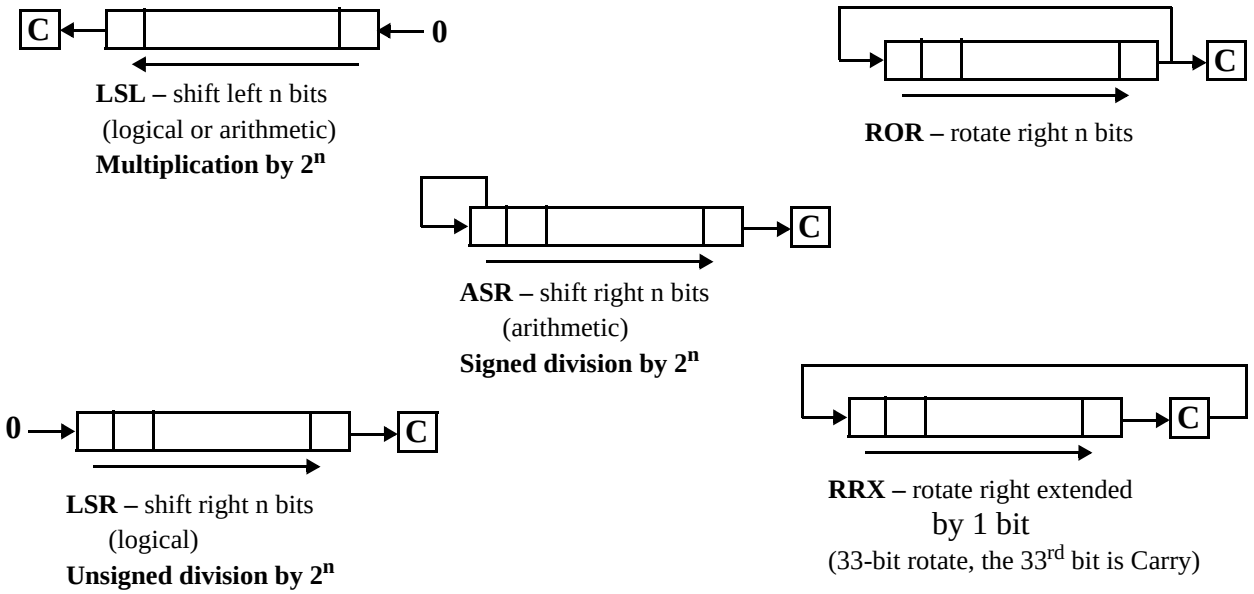

Figure 3. The Five Shift Operations

4.4.1 Addressing modes with shifts and rotations

Table 6 for <*Oprnd2*> shows the possible addressing modes available for data processing operations, including the shift pre-processing described above. Some combinations of these addressing modes and instructions are shown in the examples. The constant allowed in the <*imm_5*> operand is at most 5 bits, which provides a range of 0 to 31.

Examples:

```
MOV   r2,r5,LSL #2          @ r2 = r5 << 2 (=r5*4)
ADD   r2,r5,ASR #3          @ r2 = r2+(r5 >> 3) (= r2+r5/8)
MOV   r2,r5,LSL r3          @ r2 = r5 shifted left by the number of
                            @   positions given by the contents of r3
MOV   r2,r5,ROR r4          @ r2 = r5 rotated right by the number of
                            @   positions given by the contents of r4
```

4.5 Load and Store instructions

In order to transfer data between registers and memory, the Load and Store instructions must be used. There are three types of Load/Store instructions: single-register transfers, multiple-register transfer, and swap. Only the first kind are presented at this point in Table 7.

The destination operand, *Rd*, must be a register. There are several possibilities for the addressing mode used in the second operand, denoted by <*a_mode2*>. These modes are the index addressing mode, the PC-relative, and the preindex and postindex autoincrementing modes.

Table 6. Addressing modes for <Oprnd2>

Immediate value	*#<imm_8>*
Register	*Rm*
Logical shift left by immediate	*Rm*, LSL *#<imm_5>*
Logical shift left by register	*Rm*, LSL *Rs*
Logical shift right by immediate	*Rm*, LSR *#<imm_5>*
Logical shift right by register	*Rm*, LSR *Rs*
Arithmetic shift right by immediate	*Rm*, ASR *#<imm_5>*
Arithmetic shift right by register	*Rm*, ASR *Rs*
Rotate right by immediate	*Rm*, ROR *#<imm_5>*
Rotate right by register	*Rm*, ROR *Rs*
Rotate right extended	*Rm*, RRX

Table 7. Load and Store Instructions

LDR	*Rd,<a_mode2>*	Rd = 32-bit contents of memory at the effective address computed from *<a_mode2>*
LDRB	*Rd,<a_mode2>*	rightmost byte of Rd = 8-bit contents of memory at the effective address computed from *<a_mode2>*; the rest of Rd is filled with 0
STR	*Rd,<a_mode2>*	32-bit location in memory at the effective address computed from *<a_mode2>* = Rd
STRB	*Rd,<a_mode2>*	8-bit location in memory at the effective address computed from *<a_mode2>* = Rd

Examples using only index addressing modes:

```
LDR   r1,[r2]    @ loads into r1 the 32-bit word contents of memory
                 @ at the address pointed to by r2

LDRB  r1,[r3]    @ loads into r1 the 8-bit byte contents of memory
                 @ at the address pointed to by r3

STR   r1,[r2]    @ stores the 32-bit word from r1 into memory
                 @ at the address pointed to by r2

STRB  r1,[r3]    @ stores the 8-bit rightmost byte from r1 into memory
                 @ at the address pointed to by r3
```

Operands of load and store instructions should be aligned. For instructions that load and store words (4-byte objects), the memory address must be divisible by four. The operand of an instruction that accesses a single byte in memory has no alignment restriction.

The GNU assembler provides a mechanism for loading the address of a memory location into a register by preceding its label in the second operand with an '=' sign. The label for the memory location would normally be declared in the data area of the program.

```
LDR   r1,=<label>     @ r1=address of location declared by <label>
```

The addressing mode generated for the =<label> operand is the PC-relative. Once the address of a memory location has been loaded into a register, another LDR can be issued with an index addressing mode for the second operand, in order to assign the contents of that memory location to a register.

The ADR and ADRL pseudo-instructions can also be used to assign the address of a label to a register. Both ADR and ADRL accept a PC-relative expression, which is a label within the same file and same code area, and calculate the offset required to reach that location. The requirement that the label be within the same code area means that the ADR and ADRL pseudoinstructions cannot be used for accessing variables whose storage is declared in the data area. To access a variable, the LDR instruction should be used.

Example:

Assume that a variable labelled 'NUM' has been allocated as a 32-bit memory word to contain an integer. The following two LDR instructions are needed to load the value contained in 'NUM' into a register.

```
LDR   r1,=NUM         @ r1=address of variable NUM
LDR   r2,[r1]         @ r2=content of variable NUM
```

Pre-indexing. The index addressing mode shown so far is simply a special case of a pre-indexed or post-indexed mode with a zero offset. The complete list of all addressing modes available for the second operand in *<a_mode2>* is given in Table 8. The general case can be seen as a type of indexing mode for the calculation of an *effective addres*s, denoted by *EA*, of the location in memory which needs to be reached. The more complex cases include index registers, offsets, shifts and autoincrements, and they are briefly explained here. For a complete reference it is important to consult an ARM programming text.

In the simplest case, only a register number is given with nothing added, as in *[Rn]*. The EA is given directly by the content of the register, thus *EA = Rn.*

```
LDR   r1,[r2]    @ loads into r1 the 32-bit word content of memory
                 @ at the address pointed to by r2
```

In all cases for *pre-indexing* addressing modes, the EA is computed using all the items contained within the square brackets before any access to memory for execution of the instruction. The leftmost register is often called the base register. The two initial possibilities include having an immediate constant, as in *[Rn,#<imm>],* or having another register, as in *[Rn,Rm],* to be added to the content of the base register. The EA is computed to be the sum of the content of the base register plus the *<imm>* constant, or plus the content of the additional register *Rm,* respectively.

```
LDR   r1,[r2,#4]      @ loads into r1 the 32-bit word content of memory
                      @ at the address at r2+4
                      @  EA = r2 + 4

STR   r1,[r2,r3]      @ stores the content of r1 into memory
                      @ at the address at r2+r3
                      @  EA = r2 + r3
```

The possibility of having an extra offset contained in a register, as in *[Rn,Rm]*, also includes all the cases where shifts and rotations can be added to the addressing mode. The calculation of the EA is done accordingly, as the sum of the contents of the two registers including any shift or rotation applied to the second register.

```
LDR   r1,[r2,r4,LSL #2]     @ loads into r1 the 32-bit word content of
                            @   memory at the address at r2+r4*4
```

In all the cases above, the EA is calculated using all items within square brackets before accessing memory - thus the name *pre-indexing*. After execution of the instruction, the base register, *Rn*, is not changed. If *Rn* needs to be updated, then *pre-indexing with writeback* should be used. The format is the same as above, with an extra '!' added after the right closing square bracket, for all the cases except the one with zero offset - that is, the cases for *[Rn]* and *[Rn]!* are the same and the second one is not used.

```
LDR   r1,[r2,#4] !          @ loads into r1 the 32-bit word content of
                            @   memory at the address at r2+4
                            @ after execution, r2 = r2 + 4

STR   r1,[r2,r3]  !         @ stores the content of r1 into
                            @   memory at the address at r2+r3
                            @ after execution, r2 = r2 +r3

LDR   r1,[r2,r4,LSL #2]     @ loads into r1 the 32-bit word content of
                            @   memory at the address at r2+r4*4
                            @ after execution, r2 = r2 + r4*4
```

Post-indexing. The post-indexed modes listed for <*a_mode2*> include the additional items to be used in an address computation outside the square brackets enclosing the index register. This is to indicate that the EA to be computed to access a memory operand is simply given by the content of the base or index register *Rn*, as in the initial simplest use of *[Rn]*. The rest is to be added to the register after execution of the load or store instructions – thus the name post-indexing – and the index register is always updated.

```
LDR   r1,[r2]    @ loads into r1 the 32-bit word content of memory
                 @   at the address pointed to by r2

LDR   r1,[r2],#4 @ loads into r1 the 32-bit word content of
                 @   memory at the address pointed to by r2
                 @ after execution, r2 is updated to be r2+4

STR   r1,[r2],r3 @ stores the content of r1 into memory at the
                 @   address pointed to by r2
                 @ after execution, r2 is updated to be r2+r3

LDR   r1,[r2],r4,LSL #2 @ loads into r1 the 32-bit word content of
                        @   memory at the address pointed to by r2
                        @ after execution, r2 = r2 + r4*4
```

Both pre-indexing and post-indexing addressing modes permit very efficient code to be generated for loops that step through arrays.

Table 8. Addressing modes for Load and Store

Index mode	*Syntax*	*Effective address calculated (EA)*	*Base register*
Pre-index			
Immediate offset	[*Rn*,#<*imm_5*>]	EA = Rn+<imm>	Rn unchanged
Zero offset	[*Rn*]	EA = Rn	Rn unchanged
Register offset	[*Rn*,*Rm*]	EA = Rn+Rm	Rn unchanged
Scaled register offset	[*Rn*,*Rm*, LSL #<*imm_5*>]	EA = Rn+Rm shifted	Rn unchanged
	[*Rn*,*Rm*, LSR #<*imm_5*>]		Rn unchanged
	[*Rn*,*Rm*, ASR #<*imm_5*>]		Rn unchanged
	[*Rn*,*Rm*,ROR #<*imm_5*>]		Rn unchanged
	[*Rn*,*Rm*, RRX]		Rn unchanged
Pre-index with writeback			
Immediate offset	[*Rn*,#<*imm_5*>]!	EA = Rn+<imm>	Rn updated
Register offset	[*Rn*,*Rm*]!	EA = Rn+Rm	Rn updated
Scaled register offset	[*Rn*,*Rm*, LSL #<*imm_5*>]!	EA = Rn+Rm shifted	Rn updated
	[*Rn*,*Rm*, LSR #<*imm_5*>]!	EA = Rn+Rm shifted	Rn updated
	[*Rn*,*Rm*, ASR #<*imm_5*>]!	EA = Rn+Rm shifted	Rn updated
	[*Rn*,*Rm*, ROR #<*imm_5*>]!	EA = Rn+Rm shifted	Rn updated
	[*Rn*,*Rm*, RRX]!	EA = Rn+Rm shifted	Rn updated
Post-index			
Immediate offset	[*Rn*], #<*imm_5*>	EA = Rn	Rn updated
Register offset	[*Rn*],*Rm*	EA = Rn	Rn updated
Zero offset	[*Rn*]	EA = Rn	Rn updated
Scaled register offset	[*Rn*],*Rm*,LSL #<*imm_5*>	EA = Rn	Rn updated
	[*Rn*],*Rm*, LSR #<*imm_5*>	EA = Rn	Rn updated
	[*Rn*],*Rm*, ASR #<*immv*>	EA = Rn	Rn updated
	[*Rn*],*Rm*, ROR #<*imm_5*>	EA = Rn	Rn updated
	[*Rn*],*Rm*, RRX	EA = Rn	Rn updated

4.6 Load and Store Multiple instructions

Load multiple and *store multiple* instructions transfer the content of multiple registers between memory and the processor in a single instruction. They are used mostly for saving and restoring context during subroutine calls. The transfer starts from a base register *Rd* containing a memory address - usually *Rd* is *R13*, the *stack pointer*, and the instructions are used as Push/Pop to/from the stack. The register list must be enclosed in braces; the list may be discontinuous and it will be sorted automatically. Multiple register instructions are executed more quickly than the equivalent set of single register ones.

Table 9. Load and Store Multiple Registers Instructions

`LDM`	*`<a_mode4> Rd{!}, <reglist>`*	Load a list of registers starting from [Rd]
`STM`	*`<a_mode4> Rd{!}, <reglist>`*	Store a list of registers starting from [Rd]

The memory area to be used to store/load the contents of the register from the list is addressed by a base register, *Rd*, just like the single register `LDR` and `STR` instructions. The base register *Rd* can be incremented or decremented automatically if the extra '!' is added.

The choices given in <a_mode4> decide two aspects of the execution: (1) whether the data structure in memory, where the registers are stored (/loaded), grows *upwards* by memory addresses or *downwards*; (2) whether the base address is going to be adjusted *before* or *after* the operation occurs. The important point is that, whatever the choices made, the two load and store instructions are matched in their behaviour to maintain the balance when used in the context of entry and exit from a function.

Examples:

```
STMIA  r9!,{r1-r3,r6} @ store r1-r3 and r6 starting from [r9], updating r9
          ... other instructions ...
LDMDB  r9!,{r1-r3,r6} @ load r1-r3 and r6 starting from [r9], updating r9
```

The first `STMIA` instruction stores the content of registers *R1, R2, R3, R6* (a total of 16 bytes) in memory at the starting address indicated by the content of *R9*. For example, if *R9* = 0x0000 9010, then the content of *R1* are copied at address 0000 9010, *R2* at address 0000 9014, *R3* at address 0000 9018, and *R6* at address 0000 901B. Then *R9* is incremented by 16 and at the end is *R9* = 0x0000 9020. This is because the choice was "`IA`", that is, *Increment After*, as referring to the base register *R9*. Similarly the `LDMDB` instruction retrieves the content of registers *R1, R2, R3, R6* (a total of 16 bytes) from memory starting at the address indicated by the content of *R9*. If this is executed as a pair after the previous `STMIA`, then *R9* is first decremented by 16 and becomes *R9* = 0x0000 9010, because of the "`DB`", *Decrement Before*. After that, *R1, R2, R3, R6* are copied as a sequence from that address.

The four choise for addressing modes are: "`IA, IB, DA, DB`". They represent the functionality of the instructions directly, but more commonly their counterparts for use on stacks are used.

All the addressing modes available are listed in Table 10. Here the correspondence is also given between the modes as used directly (as above), and the modes as used, much more commonly, for stack operations.

The addressing modes "`FD, ED, FA, EA`" denote the functionality of a stack with push and pop operations. The first letter, "`F`" or "`E`" denotes whether the stack is considered to be *Full* or *Empty* when accessed. A *Full* stack implies that the current address to its top points to the last element pushed on the stack (thus to the *full* top). An *Empty* stack implies that the current address to its top points to the next empty element to be used for a push on the stack (thus to the *empty* top). The second letter, "`D`" or "`A`" denotes whether the stack grows towards a *lower* memory address, as in *Descending*, or towards a *higher* memory address, as in *Ascending*.

Examples:

```
STMFD  r13!, {r1-r3}  @ Push onto a full descending stack
LDMFD  r13!, {r1-r3}  @ Pop from a full descending stack
```

The instruction `STMFD` decrements the base register, *R13* (the stack pointer) and changes its value (because of the "!"), and then stores registers R1-R3 in that space. The instruction `LDMFD` increments the base register,*R13* (the stack pointer), and changes its value (see "!") and loads registers R1-R3.

Examples:

```
STMFA r13!, {r0-r5}   @ Push onto a full ascending stack
LDMFA r13!, {r0-r5}   @ Pop from a full ascending stack
STMFD r13!, {r0-r5}   @ Push onto a full descending stack
LDMFD r13!, {r0-r5}   @ Pop from a full descending stack
STMEA r13!, {r0-r5}   @ Push onto an empty ascending stack
LDMEA r13!, {r0-r5}   @ Pop from an empty ascending stack
STMED r13!, {r0-r5}   @ Push onto an empty descending stack
LDMED r13!, {r0-r5}   @ Pop from an empty descending stack
```

The most important thing to remember is to use the same stack instruction syntax for matching pairs of `STM` and `LDM` instructions to achieve correctness. The system stack, with stack pointer denoted by register *R13* or *SP*, is usually *Full Descending*, and all examples here will use that syntax, highlighted in the examples above.

Table 10. Addressing Mode 4 - Multiple Data Transfer

Block load			**Stack pop**	
IA	Increment After	⇔	FD	Full Descending
IB	Increment Before	⇔	ED	Empty Descending
DA	Decrement After	⇔	FA	Full Ascending
DB	Decrement Before	⇔	EA	Empty Ascending
Block store			**Stack push**	
IA	Increment After	⇔	EA	Empty Ascending
IB	Increment Before	⇔	FA	Full Ascending
DA	Decrement After	⇔	ED	Empty Descending
DB	Decrement Before	⇔	FD	Full Descending

Examples:

```
STMFD  sp!,{r1,r2}    @ copy/push R1 and R2 onto the stack
LDMFD  sp!,{r1,r2}    @ copy/pop R1 and R2 from the stack
```

Let R13 = sp = 0x 0000 8010 and let R1 = 0x 0000 0009 and R2 = 0x 0000 000A. Figure 4 shows the stack before and after the execution of the STMFD instruction. At the end, R1 and R2 are unchanged, while SP = 0x 0000 8008.. .

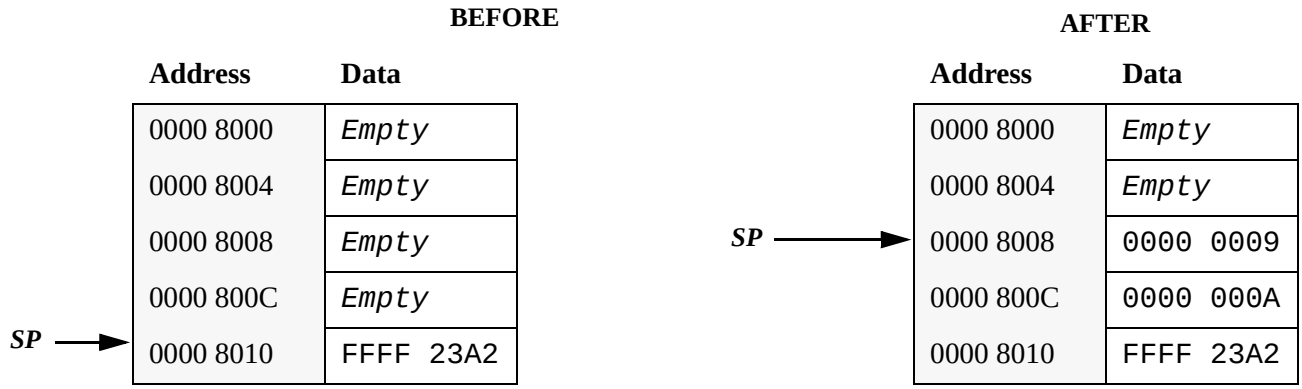

Figure 4. STMFD Instruction: `STMFD sp!,{r1,r2}`

Figure 5 shows the stack pointer positions before and after the execution of the LDMFD instruction. The multiple Load and Store instructions are best understood when used in the context of subroutine calls and parameters passing.

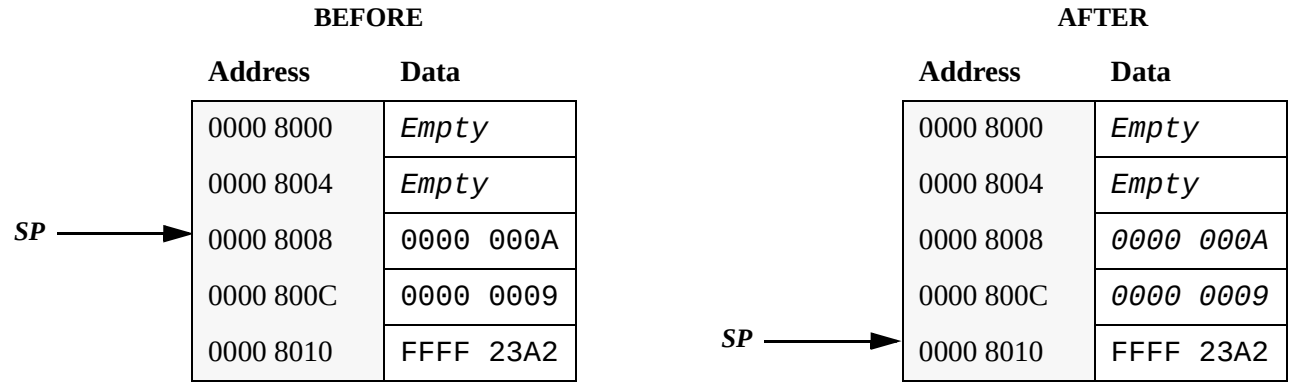

Figure 5. LDMFD Instruction: `LDMFD sp!,{r1,r2}`

4.7 Compare and Test Instructions

Comparisons instructions are used to compare or test the content of a register with a given 32-bit value. The *CPSR* is always affected, but no register is changed. The `CMP` instruction uses the same addressing modes as arithmetic instructions, thus the first operand is a register, *Rd*, and the second operand can be an immediate constant or a register with any additional shifts or rotations. The instructions `TST` and `TEQ` perform direct testing on a register by using a Boolean evaluation with the AND and Exclusive-OR operators respectively. All these instructions change the condition code bits in the *CPSR.*

Table 11. Compare and Test Instructions

CMP{S}	*Rd*,<*Oprnd2*>	Update {CPSR} after (Rn - <*Oprnd2*>) ;Rn unchanged
TST	*Rd*,<*Oprnd2*>	Update {CPSR} after (Rn AND <*Oprnd2*>) ;Rn unchanged
TEQ	*Rd*,<*Oprnd2*>	Update {CPSR} after (Rn EOR <*Oprnd2*>) ;Rn unchanged

4.8 Branch Instruction

The branch instructions are listed in Table 12. They are used to transfer control conditionally to another instruction anywhere in memory, changing the sequential flow of execution. The main branch instruction, 'B' is modified by the desired condition according to Table 13. That is to say, the notation 'B{cond}' is shorthand for a list of the 17 different mnemonics BEQ, BNE ... BAL. The BEQ mnemonic represents a branch instruction which is taken only when the CPSR has its Z bit equal to one. The other mnemonics must similarly be interpreted according to the details provided in Table 13. Note that writing the instruction without a condition, as in

```
B Label3
```

is equivalent to writing the BAL (branch always) mnemonic.

The instruction 'BL' is used for calls to subroutines and it is typically used without a condition (BL and BLAL are synonyms). It is important not to confuse the actions of 'BL' versus 'BAL'.

Table 12. Branch Instructions

B{*cond*}<*label*>	branch conditionally	if condition is satisfied then R15 = <*label*>
BL{*cond*}<*label*>	branch and link conditionally	if condition is satisfied then R14 = address following the BL R15 = <*label*>

The branching instruction tests the contents of the *CPSR* to determine whether control should be transferred. The condition code listed in *{cond}* specifies which bits in the *CPSR* need to be tested. The possible settings for the *cond* are listed in Table 13.

The BL instruction determines the address of the next instruction (by inspecting the PC register, r15) and stores that address in LR (r14). It then transfers control to the address of <*label*>. That label would normally be located at a subroutine entry point. When the subroutine has finished its computations, it can return control back to the caller by copying the return address from LR (r14) to PC (r15) using a MOV instruction, or as part of the LDM instruction.

The BEQ and BNE instructions test whether the result of an arithmetic or logical operation result is zero by looking at the *Z* bit in the *CPSR*. The BVS and BVC instructions test for arithmetic overflow. BVS checks the *V* flag in the *CPSR* which indicates whether a signed overflow occurs (e.g. the sum of two large positive integers produces a negative result). BVC checks the *C* flag in the *CPSR* indicating an unsigned overflow (e.g. 0xFFFFFFFF + 1 = 0). Similarly, BGE, BLT, BGT and BLE are normally used after comparisons between *signed* integers by looking at some logical combination of the *N, V* and *Z* flags. They test for greater than or equal (GE), less than (LT), greater then (GT), and less than or equal (LE). Following a comparison between two *unsigned* integers, for example two addresses, the BHI and BLS instructions test for higher than and for lower or equal, respectively. The precise *CPSR* combination of bits tested for each condition is shown for completeness in the rightmost column of Table 13.

Examples:

Let r1 = `0x 0000 00FF` and r2 = `0x 0000 0000`. After execution of:

```
CMP   r1,r2
```

each of the following branch instructions would have the effect shown.

`BEQ <label>`	– branch not taken	reason: (r1−r2) equal to zero: Z==1
`BNE <label>`	– branch taken	reason: (r1−r2) not equal to zero: Z==1
`BGE <label>`	– branch taken	reason: r1≥r2, i.e. (r1−r2) greater or equal to zero: N==V
`BGT <label>`	– branch taken	reason: r1>r2, i.e. (r1−r2) greater than zero: N==V && Z==0
`BLE <label>`	– branch not taken	reason: r1≤r2, i.e. (r1−r2) less or equal to zero: N != V OR Z == 1
`BLT <label>`	– branch not taken	reason: r1<r2, i.e. (r1−r2) less than zero: N != V
`BAL <label>`	– branch taken	reason: none
`BMI <label>`	– branch not taken	reason: (r1−r2) is negative: N==1
`BPL <label>`	– branch taken	reason: (r1−r2) is positive or zero: N==0
`BHI <label>`	– branch taken	reason: (unsigned r1) > (unsigned r2): C == 1 && Z == 0
`BLS <label>`	– branch not taken	reason: (unsigned r1) ≤ (unsigned r2): C == 0 || Z == 1

Table 13. Condition Code Mnemonics {cond}

`EQ`	Equal to zero	Z == 1	Check for zero
`NE`	Not equal to zero	Z == 0	
`CS or HS`	Carry Set, unsigned higher or same (after a compare)	C == 1	Check Carry
`CC or LO`	Carry clear, unsigned lower (after a compare)	C == 0	
`MI`	Minus (last result negative)	N == 1	Check positive or negative
`PL`	Plus (last result positive or zero)	N == 0	
`VS`	V set for signed overflow	V == 1	Check overflow
`VC`	V clear for no signed overflow	V == 0	
`HI`	Unsigned higher (after a compare)	C == 1 && Z == 0	Use for *unsigned* integers
`LS`	Unsigned lower or same (after a compare)	C == 0 || Z == 1	
`GE`	Signed greater or equal	N == V	Use for *signed* integers
`LT`	Signed less than	N != V	
`GT`	Signed greater than	N ==V && Z == 0	
`LE`	Signed less than or equal	N != V || Z == 1	
`AL`	Always	true	Unconditional

5. Assembler Directives

5.1 Program Layout in Memory

A program can be divided into areas of memory. The files assembled and linked will create an executable object which occupies two separate regions of memory – one to hold the instructions, usually called the *text* part of the program, and another one to hold the statically allocated variables and data that are defined in the file, usually called the *data* part of the program. When the program begins execution, it will be provided with an additional region of memory which holds the *run-stack* - the stack of frames which grows and contracts as functions are entered and exited. One can see the list of addresses of the regions of memory occupied by the program when running it in the simulator and examining the appropriate registers and memory contents.

5.2 GNU Assembly Language Directives

As shown in the small example in section 2, the Assembler and the Linker need some information extraneous to the executable instructions in order to function correctly. This is accomplished through the use of Assembler directives - instructions directed to the Assembler and not adding any binary executable code to the processor. They include the directive for the allocation of storage. Assembler directives are local to each Assembler implementation. The local implementation is based on the widely distributed set of GNU tools and follows its conventions. Not all the GNU directives are available and the explanation covers the ones most commonly used. The most useful directives are summarized in Table 14, with examples appearing below.

Table 14. Assembler Directives

`.text`	Begin a source code area containing instructions
`.data`	Start portion of the source code where data allocation is declared
`.end`	This denotes the end of the source code
`.global` *`label`*	Make *label* externally visible
`_start:`	Make this address the start address for the Linker
`.extern` *`label`*	declare *label* as being a storage location defined in another module
`.word` *`n`*	Reserve one word of storage (4 bytes), initialized to *n*
`.byte` *`n`*	Reserve one byte of storage, initialized to *n*
`.skip` *`k`*	Reserve *k* consecutive bytes of memory, uninitialized
`.ascii` *`"string"`*	Place an ASCII string in memory
`.asciz` *`"string"`*	Place a null-terminated ASCII string in memory
`.align [`*`n`*`]`	Align next item on an address divisible by 2^n. I.e., 0 → byte boundary, 1 → halfword boundary, 2 → word boundary. The default is a word boundary if *n* is omitted.
`.equ` *`name,value`*	Define symbolic label *name* to represent the constant *value*. Any expression which evaluates to a constant may be used.

Examples:

`.global _start`	*@ makes the label "`_start`" available globally to the Linker*
`.extern PrintInt`	*@informs the Linker that the source file includes a call to an external routine "`PrintInt`"*
`Size: .word`	*@declares storage for a 32-bit integer named "`Size`" and initialized to "3"*
`Vec: .word 3,-1,2`	*@declares storage for an array named "`Vec`" of three 32-bit integers, initialized as shown*

`Vec1: .word 3`
`.word -1`
`.word 2`
	@Vec1 is the *same as the declaration of "`Vec`" above*

`Bb: .byte 0xFF`	*@declares storage for one byte named "`Bb`" initialized to Hex "FF"*
`AR: .skip 40`	*@declares storage for 40 bytes named "`AR`", uninitialized (possibly 10 integers)*
`hello:.ascii "Welcome user"` `.byte 0x00`	*@declares a string of 12 bytes, named "`hello`", initialized plus a null byte to be appended to the string for termination*
`bye: .asciz "Good Bye"`	*@declares a string of 9 bytes, automatically including the null byte*
`.equ MAX,20`	*@the label "`MAX`" will be replaced in all occurrences by the constant 20*

6. References and Bibliography

- CodeSourcery Toolchain. URL: `http://www.codesourcery.com/gnu_toolchains/arm/`
- ARM Architecture Reference Manual, Second Edition, edited by David Seal. Addison-Wesley, 2001. ISBN 0 201 73719 1.
- Gnu Assembler User Guide, version 2.17. URL: `http://sourceware.org/binutils/docs-2.17/as/`

Appendix A: Instruction Set Sorted by Mnemonic (not complete)

Opcode	Operands	Description
ADC{S}	*Rd,Rn,<Oprnd2>*	Rd = Rn + Oprnd2 + Carry {CPSR}
ADD{S}	*Rd,Rn,<Oprnd2>*	Rd = Rn + Oprnd2 {CPSR}
AND{S}	*Rd,Rn,<Oprnd2>*	Rd = Rn AND Oprnd2 {CPSR}
B{*cond*}<*label*>	branch conditionally	if condition is satisfied then R15 = <*label*>
BIC{S}	*Rd,Rn,<Oprnd2>*	Rd = Rn AND NOT Oprnd2 {CPSR}
BL{*cond*}<*label*>	branch and link	if condition is satisfied then R14 = address following the BL; R15 = <*label*>
CLZ	*Rd,Rm*	Rd = number of leading zeroes in Rm
CMP{S}	*Rd,<Oprnd2>*	Update {CPSR} after (Rn - <*Oprnd2*>) ;Rn unchanged
EOR{S}	*Rd,Rn,<Oprnd2>*	Rd = Rn EXOR Oprnd2 {CPSR}
LDM<a_mode4>	*Rd{!},<reglist>*	Load list of registers from Rd
LDR	*Rd,<a_mode2>*	Rd = word of memory at address computed from <a_mode2>
LDRB	*Rd,<a_mode2>*	rightmost byte of Rd =byte of memory at the address computed from <a_mode2>; the rest of Rd is filled with 0
MLA{S}	*Rd,Rm,Rs,Rn*	Rd = Rm * Rs + Rn {CPSR}
MOV{S}	*Rd,<Oprnd2>*	Rd = Oprnd2 {CPSR}
MUL{S}	*Rd,Rm,Rn*	Rd = Rm * Rn {CPSR}
MVN{S}	*Rd,<Oprnd2>*	Rd = NOT Oprnd2 {CPSR}
NOP		R0 = R0
ORR{S}	*Rd,Rn,<Oprnd2>*	Rd = Rn OR Oprnd2 {CPSR}
RSB{S}	*Rd,Rn,<Oprnd2>*	Rd = Oprnd2 - Rn {CPSR}
RSC{S}	*Rd,Rn,<Oprnd2>*	Rd = Oprnd2 - Rn - NOT Carry {CPSR}
SBC{S}	*Rd,Rn,<Oprnd2>*	Rd = Rn - Oprnd2 - NOT Carry{CPSR}
STM<a_mode4>	*Rd{!},<reglist>*	Store list of registers to [Rd]
STR	*Rd,<a_mode2>*	word in memory at address computed from <a_mode2> = Rd
STRB	*Rd,<a_mode2>*	byte in memory at the address computed from <a_mode2> = Rd
SUB{S}	*Rd,Rn,<Oprnd2>*	Rd = Rn – Oprnd2 {CPSR}
SWI	<int_vector>	flow of control changed to interrupt vector address; supervisor mode
SWP	*Rd,Rm,[Rn]*	Rd = 32-bit content of memory at [Rn]; Rn = Rm
TEQ	*Rn,<Oprnd2>*	Update {CPSR} after (Rn EOR Oprnd2) ; Rn is unchanged
TST	*Rn,<Oprnd2>*	Update {CPSR} after (Rn AND Oprnd2) ; Rn is unchanged

Appendix B: Extract from ARM Reference card

Operation	Assembler	Action
Move	MOV{S} Rd, <Oprnd2>	Rd := Oprnd2 {CPSR}
Arithmetic	ADD{S} Rd, Rn, <Oprnd2>	Rd := Rn + Oprnd2 {CPSR}
	ADC{S} Rd, Rn, <Oprnd2>	Rd := Rn + Oprnd2 + Carry {CPSR}
	SUB{S} Rd, Rn, <Oprnd2>	Rd := Rn - Oprnd2 {CPSR}
	SBC{S} Rd, Rn, <Oprnd2>	Rd := Rn - Oprnd2 - NOTCarry {CPSR}
	RSB{S} Rd, Rn, <Oprnd2>	Rd := Oprnd2 - Rn {CPSR}
	RSC{S} Rd, Rn, <Oprnd2>	Rd := Oprnd2 - Rn - NOTCarry {CPSR}
	MUL{S} Rd, Rm, Rs	Rd := Rm * Rs {CPSR}
	MLA{S} Rd, Rm, Rs, Rn	Rd := Rm * Rs + Rn {CPSR}
	CLZ Rd, Rm	Rd := # leading zero in Rm
Logical	AND{S} Rd, Rn, <Oprnd2>	Rd := Rn AND Oprnd2 {CPSR}
	EOR{S} Rd, Rn, <Oprnd2>	Rd := Rn EXOR Oprnd2 {CPSR}
	ORR{S} Rd, Rn, <Oprnd2>	Rd := Rn OR Oprnd2 {CPSR}
	TST Rn, <Oprnd2>	Update CPSR on Rn AND Oprnd2
	TEQ Rn, <Oprnd2>	Update CPSR on Rn EOR Oprnd2
	BIC{S} Rd, Rn, <Oprnd2>	Rd := Rn AND NOT Oprnd2 {CPSR}
	NOP	R0 := R0
Compare	CMP{S} Rd, <Oprnd2>	Update CPSR on Rn - Oprnd2
Branch	B{cond} label	R15 := label
	BL{cond} label	R14 := R14-4; R15 := label
Swap	SWP Rd, Rm,[Rn]	temp := [Rn]; [Rn] := Rm; Rd := temp
Load	LDR Rd, <a_mode2>	Rd := address
	LDM <a_mode4L> Rd{!}, <reglist>	Load list of register from [Rd]
Store	STR Rd, <a_mode2>	[address]:= Rd
	STM <a_mode4S> Rd{!}, <reglist>	Store list of register to [Rd]
Software Interrupt	SWI <immed_24>	Software interrupt processor exception

Addressing Mode 2 - Data Transfer		
Pre-indexed	Immediate offset	[Rn, #+/-<immed_12>]{!}
	Zero offset	[Rn]
	Register offset	[Rn, +/-Rm]{!}
	Scaled register offset	[Rn, +/-Rm, LSL #<immed_5>]{!}
		[Rn, +/-Rm, LSR#<immed_5>]{!}
		[Rn, +/-Rm, ASR #<immed_5>]{!}
		[Rn, +/-Rm, ROR #<immed_5>]{!}
		[Rn, +/-Rm, RRX]{!}
Post-indexed	Immediate offset	[Rn], #+/-<immed_12>
	Register offset	[Rn], +/-Rm
	Zero offset	[Rn]
	Scaled register offset	[Rn], +/-Rm, LSL #<immed_5>
		[Rn], +/-Rm, LSR #<immed_5>
		[Rn], +/-Rm, ASR #<immed_5>
		[Rn], +/-Rm, ROR #<immed_5>
		[Rn], +/-Rm, RRX

Key to tables	
{cond}	See Condition Field
<Oprnd2>	See Operand 2
{S}	Updates CPSR if present
<immed>	Constant
<a_mode2>	See Addressing Mode 2
<a_mode4>	See Addressing Mode 4
<reglist>	List of registers with commas inside braces
{!}	Updates base register if present

Operand 2	
Immediate value	#<immed_8>
Logical shift left immediate	Rm, LSL #<immed_5>
Logical shift right immediate	Rm, LSR #<immed_5>
Arithmetic shift right immediate	Rm, ASR #<immed_5>
Rotate right immediate	Rm, ROR #<immed_5>
Register	Rm
Rotate right extended	Rm, RRX
Logical shift left register	Rm, LSL Rs
Logical shift right register	Rm, LSR Rs
Arithmetic shift right register	Rm, ASR Rs
Rotate right register	Rm, ROR Rs

Addressing Mode 4 - Multiple Data Transfer			
Block load		Stack pop	
IA	Increment After	FD	Full Descending
IB	Increment Before	ED	Empty Descending
DA	Decrement After	FA	Full Ascending
DB	Decrement Before	EA	Empty Ascending
Block store		Stack push	
IA	Increment After	EA	Empty Ascending
IB	Increment Before	FA	Full Ascending
DA	Decrement After	ED	Empty Descending
DB	Decrement Before	FD	Full Descending

Condition Field	
EQ	Equal
NE	Not equal
CS	Carry Set
CC	Carry clear
MI	Negative
PL	Positive or zero
VS	Overflow
VC	No overflow
HI	Unsigned higher
LS	Unsigned lower or same
GE	Signed greater or equal
LT	Signed less than
GT	Signed greater than
LE	Signed less than or equal
AL	Always