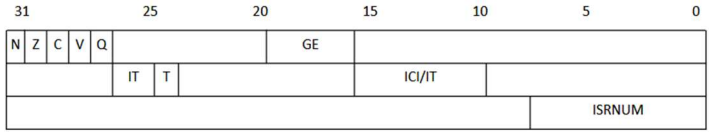
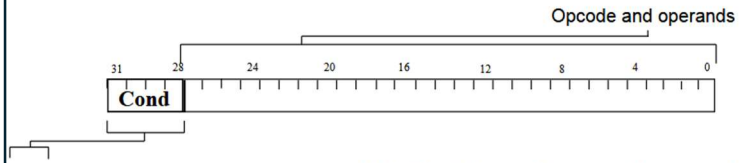
ARM Assembly Instructions



	Logical Instruction	Arithmetic Instruction
Flag		
Negative (N='1')	No meaning	Bit 31 of the result has been set Indicates a negative number in signed operations
Zero (Z='1')	Result is all zeroes	Result of operation was zero
Carry (C='1')	After Shift operation '1' was left in carry flag	Result was greater than 32 bits
oVerflow (V='1')	No meaning	Result was greater than 31 bits Indicates a possible corruption of the sign bit in signed numbers



0000 = **EQ** - Z set (equal)
0001 = **NE** - Z clear (not equal)
0010 = **HS** / CS - C set (unsigned higher or same)
0011 = **LO** / CC - C clear (unsigned lower)
0100 = **MI** -N set (negative)
0101 = **PL** - N clear (positive or zero)
0110 = **VS** - V set (overflow)
0111 = **VC** - V clear (no overflow)
1000 = **HI** - C set and Z clear (unsigned higher)
1001 = **LS** - C clear or Z (set unsigned lower or same)
1010 = **GE** - N set and V set, or N clear and V clear (>or =)
1011 = **LT** - N set and V clear, or N clear and V set (>)
1100 = **GT** - Z clear, and either N set and V set, or N clear and V set (>)
1101 = **LE** - Z set, or N set and V clear,or N clear and V set (<, or =)
1110 = **AL** - always
1111 = **NV** - reserved.

- `CMP` (compare) substracts `operand2` from `Rd` and updates the flags.
- `CMN` (compare negative) adds `operand2` to `Rd` and updates the flags.
  - The execution of the arithmetic operations at the base of the `CMP` and `CMN` is not modifying the content of the operands.
- `TST` (test) computes the logical AND between `operand2` and `Rd`; then updates all the flags except `V`.
- `TEQ` (test equivalence) computes the logical EOR between `operand2` and `Rd`; then updates all the flags except `V`.
- `MRS <Rn>, <Sreg>` copies a special register into a register.
- `MSR <Sreg>, <Rn>` copies a general purpose register into a special register.
- `Sreg` can be `APSR`, `EPSR`, `IPSR`, and `PSR`.
- `MRS r0, APSR` reads the flags and copies them to the uppermost nibble of `r0`.

- `ADD <Rd>,<Rn>,<op2>` `Rd = Rn + op2`
- `ADC <Rd>,<Rn>,<op2>`

  `Rd = Rn + op2 + C`
- `ADDW` is like `ADD`, but it takes only a 12-bit value and it can not update flags.
- With `ADC` it is possible to add 64-bit values:

```
ADDS r4,r0,r2
ADC r5,r1,r3
```

- `SUB <Rd>,<Rn>,<op2>` `Rd = Rn - op2`
- `SBC <Rd>,<Rn>,<op2>`

  `Rd = Rn - op2 + C - 1`
- `SUBW` is like `SUB`, but it takes only a 12-bit value and it can not update flags.
- With `SBC` it is possible to subtract 64-bit values

```
SUBS r4,r0,r2
SBC r5,r1,r3
```

- `RSB <Rd>,<Rn>,<op2>`

  `Rd = op2 - Rn + C – 1`
- Advantages:
  - either one or the other operand can be shifted before the subtraction

```
SUB r0, r1, r2, LSL #2   ;r0 = r1 - r2*4
RSB r0, r2, r1, LSL #2   ;r0 = r1*4 - r2
```

  - a register can be subtracted from a constant.

- multiplication with 32-bit result

  `MUL <Rd>, <Rn>, <Rm>`
- unsigned multiplication with 64-bit result

  `UMULL <Rd1>, <Rd2>, <Rn>, <Rm>`
- signed multiplication with 64-bit result

  `SMULL <Rd1>, <Rd2>, <Rn>, <Rm>`
- Note 1: there is no distinction between signed and unsigned multiplication with 32-bit result.
- Note 2: all operands must be registers.
- `MLA <Rd>, <Rn>, <Rm>, <Ra>`

  `Rd = Rn * Rm + Ra`
- `MLS <Rd>, <Rn>, <Rm>, <Ra>`

  `Rd = Rn * Rm – Ra`
- `UMLAL <Rd1>, <Rd2>, <Rn>, <Rm>`

  `Rd1,Rd2 = Rn * Rm + Rd1,Rd2`
- `SMLAL <Rd1>, <Rd2>, <Rn>, <Rm>`

  same as `UMLAL`, but with signed values.

- unsigned division

  `UDIV <Rd>, <Rn>, <Rm>`
- signed division

  `SDIV <Rd>, <Rn>, <Rm>`
- If `Rn` is not exactly divisible by `Rm`, the result is rounded toward zero.
- `UDIV` and `SDIV` do not change the flags (the suffix 'S' can not be added).

SHIFT

- `LSL <Rd>, <Rn>, <op2>`
- `LSR <Rd>, <Rn>, <op2>`
- `ASR <Rd>, <Rn>, <op2>`

- **ROTATE**
- `ROR <Rd>, <Rn>, <op2>`
- `RRX <Rd>, <Rn>`

- After a sum, C = 1 if the result size is 33 bit.

```
  1010 0001 0001 0001 0001 0001 0001 0001 +
  1011 0101 0101 0101 0101 0101 0101 0101 =
C = 1 0101 0110 0110 0110 0110 0110 0110 0110
```

- After a subtraction, C is inverted. Here C = 1:

```
  1101 0000 0000 0000 0000 0000 0000 0000 -
  0101 0000 0000 0000 0000 0000 0000 0000 =
  1000 0000 0000 0000 0000 0000 0000 0000
```

- After a move or logical instruction, C is the result of an inline barrel shifter operation.

- It corresponds to the first bit of the result.
- If N =1, a 2's complement number is negative.
- Example: -4 + (-3) = -7 -> N = 1

```
  1111 1111 1111 1111 1111 1111 1111 1100 +
  1111 1111 1111 1111 1111 1111 1111 1101 =
N = 1111 1111 1111 1111 1111 1111 1111 1001
```

- Why N = 1 in the following sum?

```
  0111 0001 0001 0001 0001 0001 0001 0001 +
  0011 0101 0101 0101 0101 0101 0101 0101 =
N = 1010 0110 0110 0110 0110 0110 0110 0110
```

- It is set if, in a sum of values with the same sign, there is a change in the MSB

```
This is a carry
  1010 0001 0001 0001 0001 0001 0001 0001 +
  1011 0101 0101 0101 0101 0101 0101 0101 =
(V = 1) 1  0101 0110 0110 0110 0110 0110 0110 0110
```

- V = 0 if both conditions occur: the result is right

```
  1110 0001 0001 0001 0001 0001 0001 0001 +
  1011 0101 0101 0101 0101 0101 0101 0101 =
(V = 0)  1 1001 0110 0110 0110 0110 0110 0110 0110
```

- It is set if the result is zero.

```
  1111 1111 1111 1111 1111 1111 1111 1101 +
  0000 0000 0000 0000 0000 0000 0000 0011 =
Z = 0000 0000 0000 0000 0000 0000 0000 0000
```

AREA *sectionName* {,*attr*} {,*attr*}…

If *sectionName* starts with a number, it must be enclosed in bars
e.g. |1_DataArea|

|.text| is used by the C compiler

At least one AREA directive is mandatory

Example: AREA Example,CODE,READONLY

AREA	Defines a block of code or data
RN	Can be used to associate a register with a name
EQU	Equates a symbol to a numeric constant
ENTRY	Declares an entry point to your program
DCB, DCW, DCD	Allocates memory and specifies initial runtime contents
ALIGN	Aligns data or code to a particular memory boundary
SPACE	Reserves a zeroed block of memory of a particular size
LTORG	Assigns the starting point of a literal pool
END:	Designates the end of a source file

CODE: the section contains machine code

DATA: the section contains data

READONLY: the section can be placed in read-only memory

READWRITE: the section can be placed in read-write memory

ALIGN = *expr*: the section is aligned on a 2*expr*-byte boundary

- The ALIGN directive aligns the current location to a specified boundary by padding with zeros:

ALIGN {*expr*{, *offset*}}

- The current location is aligned to the next address of the form

n * *expr* + *offset*

- If *expr* is not specified, ALIGN sets the current location to the next word boundary.

{*label*} DC*xx* *expr*{,*expr*}…

- The available directives are:
  - DCB: define constant byte
  - DCW: define constant half-word
  - DCWU: define constant half-word unaligned
  - DCD: define constant word
  - DCDU: define constant word unaligned
- *expr* is:
  - a numeric expression in the proper range
  - a string (with DCB only)

- Besides loading values from memory, LDR can be used to load constants into registers:

LDR <Rd>, =<constant>

- If constant is among the valid values of MOV, then the instruction is replaced with:

MOV <Rd>, #<constant>

- Otherwise, a block of constant, called *literal pool*, is created and the instruction becomes:

LDR <Rd>, [PC, #<offset>]

- shift is an optional shift applied to Rm
  - ASR #n: arithmetic shift right
  - LSL #n: logical shift left
  - LSR #n : logical shift right
  - ROR #n : rotate right
  - RRX : rotate right 1 bit with extend
- The equivalent shift instruction is preferred:

LSL r0,r1,#3 corresponds to MOV r0,r1,LSL #3

MOV <Rd>, #<constant>

- The constant can be:
  - a 16-bit value (0-65535)
  - a value obtained by shifting left an 8-bit value
  - of the form 0x00XY00XY
  - of the form 0xXY00XY00
  - of the form 0xXYXYXYXY.
- MOVW is like MOV, but it takes only a 16-bit value.

- The MVN instruction moves a one's complement of the operand into a register.
- Same syntax as MOV, with one difference:
  - MVN does not accept a 16-bit value
- Example: MVN r0,#0 -> r0 = 0xFFFFFFFF
- MOVT moves a 16-bit value in the high halfword of a register:

MOVT <Rd>, #<constant>

- A register can be set to any 32-bit constant by using MOV and MOVT together:

MOV r0, #0x47D2
MOVT r0, #0xC901

The new value of r0 is 0xC90147D2.

Besides loading values from memory, LDR can be used to load constants into registers:

LDR <Rd>, =<constant>

If constant is among the valid values of MOV, then the instruction is replaced with:

MOV <Rd>, #<constant>

Otherwise, a block of constant, called *literal pool*, is created and the instruction becomes:

LDR <Rd>, [PC, #<offset>]

Two pseudo-instructions are available:

LDR <Rd>, =<label>
ADR <Rd>, <label>

LDR creates a constant in a literal pool and uses a PC relative load to get the data.

ADR adds or subtracts an offset to/from PC.

ADR does not increase the code size, but it can not create all offsets.

The ADR pseudo-instruction is replaced with

LDR <Rd>, [PC, #<offset>]

The offset is expressed with 12 bits.

If the offset is higher than 4095 bytes, ADRL must be used instead of ADR.

ADRL generates two operations and its offset can be up to 1 MB.

ADR and ADRL load addresses in the same section.

Load	Store	Size and type
LDR	STR	word (32 bits)
LDRB	STRB	byte (8 bits)
LDRH	STRH	halfword (16 bits)
LDRSB	–	signed byte
LDRSH	–	signed halfword
LDRD	STRD	two words
LDM	STM	multiple words

- The address is computed by summing the offset to the value in the base register Rn:

load/store <Rd>, [<Rn>,<offset>]{!}

- the offset is either a 12-bit constant or a register, which can be shifted left up to 3 positions.

- AND <Rd>, <Rn>, op2 ;Rn AND op2
- BIC <Rd>, <Rn>, op2 ;Rn AND NOT op2
- ORR <Rd>, <Rn>, op2 ;Rn OR op2
- EOR <Rd>, <Rn>, op2 ;Rn XOR op2
- ORN <Rd>, <Rn>, op2 ;Rn OR NOT op2
- MVN <Rd>, <Rn> ; NOT Rn

- There are four instructions for unconditional branch:
  - branch `B <label>`
  - branch indirect `BX <Rn>`
  - branch and link `BL <label>`
  - branch indirect with link `BLX <Rn>`
- `BL` and `BLX` save the return address (i.e., the address of the next instruction) in `LR` (`r14`) and they are used to call subroutines.
- An infinite loop is added as last instruction:

```
stop B stop
```

or

```
     LDR r1, =stop
stop BX r1
```

Conditional branch: B?? and BX??

??	Flags	Meaning	??	Flags	Meaning
EQ	Z = 1	equal	NE	Z = 0	not equal
CS HS	C = 1	unsigned ≥	CC LO	C = 0	unsigned <
MI	N = 1	negative	PL	N = 0	positive or 0
VS	V = 1	overflow	VC	V = 0	no overflow
HI	C =1 & Z = 0	unsigned >	LS	C =0 & Z = 1	unsigned ≤
GE	N ≥ V	signed ≥	LT	N ≠ V	signed <
GT	Z =0 or N = V	signed >	LE	Z =1 or N ≠ V	signed ≤

Compare and branch if Zero:

```
CBZ <Rn>, <label>
```

jumps to `label` if `Rn` = 0

Compare and branch if Nonzero:

```
CBNZ <Rn>, <label>
```

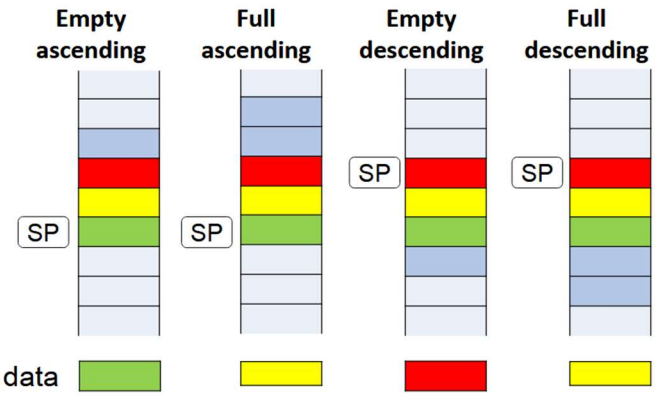
jumps to `label` if `Rn` ≠ 0

`Rn` must be among `r0-r7`.

Only forward branch is possible (4-130 byte).

- The IT (If-Then) block avoids branch penalty because there is no change to program flow:

```
ITxyz <cond>
instr1<cond> <operands>
instr2<cond OR not cond> <operands>
instr3<cond OR not cond> <operands>
instr4<cond OR not cond> <operands>
```




```
LDM{xx}/STM{xx} <Rn>{!}, <regList>
```

Stack type	PUSH	POP
Full descending	STMDB STMFD	LDM LDMIA LDMFD
Empty ascending	STM STMIA STMEA	LDMDB LDMEA

- `PUSH <regList>` is the same as `STMDB SP!, <regList>`
- `POP <regList>` is the same as `LDMIA SP!, <regList>`

While loop: implementation

```
1.        B test
   loop   ...      ; do something
   test   CMP r0, #N
          BNE loop
2. test   CMP r0, #N
          BE exit
          ...      ; do something
          B test
   exit
```

For loop: optimization

```
          MOV r0, N
loop      ...      ; do something
          SUBS r0, r0, #1
          BNE loop
exit
```

Do-While loop

The pseudocode

```
    do {
          ...      //do something
    } while (r0 != N);
```

can be implemented as:

```
loop      ...      ; do something
test      CMP r0, #N
          BNE loop
```