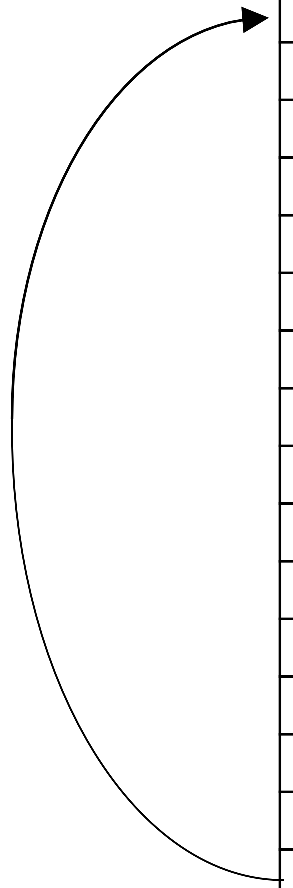
Unsigned and signed integer numbers

Carry



Binary	Unsigned	Signed
0000	**0**	**0**
0001	**1**	**1**
0010	**2**	**2**
0011	**3**	**3**
0100	**4**	**4**
0101	**5**	**5**
0110	**6**	**6**
0111	**7**	**7**
1000	**8**	**-8**
1001	**9**	**-7**
1010	**10**	**-6**
1011	**11**	**-5**
1100	**12**	**-4**
1101	**13**	**-3**
1110	**14**	**-2**
1111	**15**	**-1**

Subtraction sets C flag opposite of carry (ARM specialty)!

- if (carry = 0) then C=1

- if (carry = 1) then C=0

overflow

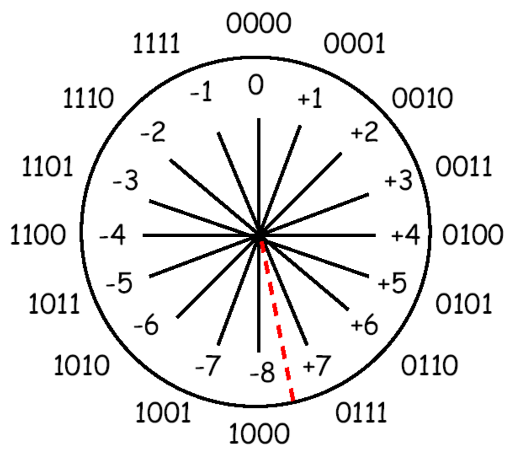
$$\updownarrow\; V = A_{n-1}B_{n-1}\overline{S}_{n-1} \vee \overline{A}_{n-1}\overline{B}_{n-1}S_{n-1}$$

Unsigned and signed integer numbers

Binary	Unsigned	Signed
1000	**8**	**-8**
1001	**9**	**-7**
1010	**10**	**-6**
1011	**11**	**-5**
1100	**12**	**-4**
1101	**13**	**-3**
1110	**14**	**-2**
1111	**15**	**-1**
0000	**0**	**0**
0001	**1**	**1**
0010	**2**	**2**
0011	**3**	**3**
0100	**4**	**4**
0101	**5**	**5**
0110	**6**	**6**
0111	**7**	**7**

C ↓

V



Overflow

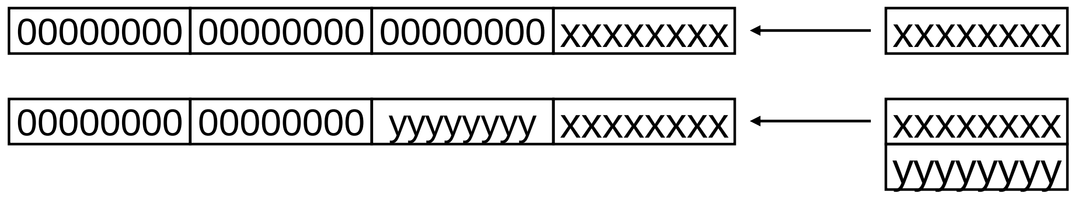
$$V = A_{n-1}B_{n-1}\overline{S}_{n-1} \vee \overline{A}_{n-1}\overline{B}_{n-1}S_{n-1}$$

Signed/unsigned extension to 32 bits

• when reading 8 and 16–bit operands from memory – they need to be extended by sign or zeros to full length (registers are 32bit long). Only load and store instructions are accessing operands in memory.

• unsigned operands from memory are extended with zeros:

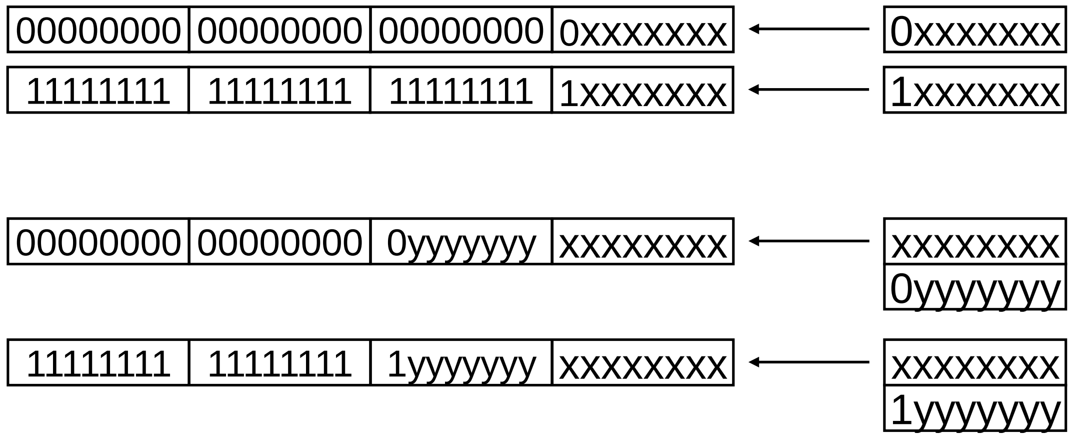
ldrb
ldrh


00000000 | 00000000 | 00000000 | xxxxxxxx
xxxxxxxx
00000000 | 00000000 | yyyyyyyy | xxxxxxxx
xxxxxxxx
yyyyyyyy

• signed operands from memory are extended with sign bit:

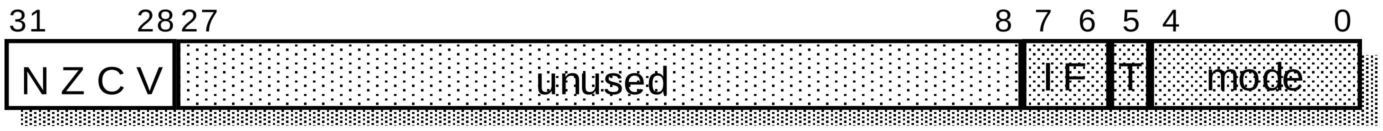
ldrsb

ldrsh


00000000 | 00000000 | 00000000 | 0xxxxxxx
0xxxxxxx
11111111 | 11111111 | 11111111 | 1xxxxxxx
1xxxxxxx
00000000 | 00000000 | 0yyyyyyy | xxxxxxxx
xxxxxxxx
0yyyyyyy
11111111 | 11111111 | 1yyyyyyy | xxxxxxxx
xxxxxxxx
1yyyyyyy

Flags

- flags are 4 bits in „status register“ CPSR , having values of :
- 1 – flag is SET.
- 0 – flag is NOT SET (unset).



Flags (can) be changed according to results of ALU operations:

N = 0: bit 31 of the result is 0, N=1: bit 31 of the result is 1 (*Negative*)

Z = 1: result is equal to 0, Z=0: result is not equal to 0 (*Zero*)

C: **+**: C = 1: result has carry, C = 0: result doesn‘t have carry (*Carry*)

-: C = **0**: result has carry, C = **1**: result doesn‘t have carry (*Carry*)

V = 1: result has overflow, V = 0: result doesn‘t have overflow(*oVerflow*)

Subtraction sets C flag opposite of carry (ARM specialty)!

- if (carry = 0) then C=1

- if (carry = 1) then C=0

If we want that ALU instruction changes flags, we have to add „s“ to coresponding instruction !!!

```
movs r1, #3           @ r1 ← 3
adds r2, r7, #0x20    @ r2 ← r7 + 32
subs r4, r5,#1        @ r4 ← r5 - 1
```

Comparisons

Compare instructions always change the state of flags (ALU group of instructions):

cmp (Compare): sets flags according to result of Rn - Op2

```
cmp R1, #10 @ R1-10
```

cmn (Compare negated): sets flags according to result of Rn + Op2

```
cmn R1, #10 @ R1+10
```

Instructions only influence the **state of flags**, **registers are not changed**. Because their only intention is to change the state of flags, we don‘t have to add letter **s** to their names.

Comparisons of unsigned numbers

Example: Compare two unsigned numbers:

- focus on the state of flags C and Z

```
mov r1,#11
cmp r1,#10 @ C=1, Z=0

mov r1,#10
cmp r1,#10 @ C=1, Z=1

mov r1,#9
cmp r1,#10 @ C=0, Z=0
```

Summary:

r1 > 10	C=1 in Z=0	Higher
r1 >= 10	C=1	Higher or Same
r1 = 10	Z=1	Equal
r1 < 10	C=0	Lower
r1 <= 10	C=0 ali Z=1	Lower or Same

Comparisons of signed numbers

Comparison uses subtraction/addition to compare numbers; the opeation is the same for unsigned and signed numbers. But to properly compare values of signed numbers, we have to watch another set of flags!

- focus on the state of flags V, Z and N

Example:

```
    mov r1,#0
    cmp r1,#-1 @ C=0, Z=0, V=0, N=0
```

Flags don‘t corespond to relation > for unsigned numbers (C=1 in Z=0)!

Condition for **> for signed** numbers is different from condition **> for unsigned** numbers.
Correct signed number condition for > is: Z = 0 and N = V .

Conditions

Name	Condition	State of flags
EQ	Equal / equals zero	Z set
NE	Not equal	Z clear
CS	Carry set	C set
CC	Carry clear	C clear
MI	Minus / negative	N set
PL	Plus / positive or zero	N clear
VS	Overflow	V set
VC	No overflow	V clear
HS	Unsigned higher or same	C set
LO	Unsigned lower	C clear
HI	Unsigned higher	C set and Z clear
LS	Unsigned lower or same	C clear or Z set
GE	Signed greater than or equal	N equals V
LT	Signed less than	N is not equal to V
GT	Signed greater than	Z clear and N equals V
LE	Signed less than or equal	Z set or N is not equal to V

Branch instructions

Branch is a „GOTO label“ instruction – usually target line is denoted by label. In this case, address of the instruction in line with a label is moved to PC.

b (Branch)

```
loop:         …
              sub r1, r1, #1
              b loop @ GOTO loop
```

This is „eternal“ loop. r1 will be continually decremented, after value of 0, its value will change to 0xffffffff.

If we want only finite number of loop repetitions (until r1 is non-zero), we need instruction of type „**IF cond THEN GOTO label**“. We need conditional branch instruction.

Conditional branch instructions

In ARM assembler condition is always determined with the state of flags (remember previous table with all possible conditions).

Before using conditional branch instruction, we need to set the state of flags. This is commonly done with cmp instruction, or quite often also with ALU instructions.

Loop, that ends when r1 reaches value of 0 could be implemented in following way:

b (Branch)

```
loop:       …
            sub r1, r1, #1
            cmp r1, #0
            bne loop @ IF Z=0 THEN GOTO loop
```

Instruction b is combined with the proper condition name that determines, if the branch will take place. If condition is not met, branch will not be executed – next instruction that follows will be next.

Therefore Bxx instruction is denoted as **Conditional branch**.

Conditional branch instructions

Instruction cmp in previous example has set the Z flag for conditional branch instruction. We can set same flag already earlier – when decrementing value of r1. We just need to add letter s to subtraction instruction (subs):

b (Branch)

```
                mov r1, #10
    loop:       …
                subs r1, r1, #1 @ sets flags accroding to r1!
                bne loop        @ IF Z=0 THEN GOTO loop
                mov r2, #10
```

Loop will repeat 10 times. When r1 reaches 0, subs will set flag Z to 1 and conditional branch will not take place anymore.
Next instruction `mov r2, #10` will be executed. Pseudo code for conditional branch is:

```
IF condition THEN PC ← label
  ELSE PC ← PC+4
```

Conditional branch instructions

Branch	Interpretation	Normal uses
B	Unconditional	Always take this branch
BEQ	Equal	Comparison equal or zero result
BNE	Not equal	Comparison not equal or non-zero result
BPL	Plus	Result positive or zero
BMI	Minus	Result minus or negative
BCC BLO	Carry clear Lower	Arithmetic operation did not give carry-out Unsigned comparison gave lower
BCS BHS	Carry set Higher or same	Arithmetic operation gave carry-out Unsigned comparison gave higher or same
BVC	Overflow clear	Signed integer operation; no overflow occurred
BVS	Overflow set	Signed integer operation; overflow occurred
BGT	Greater than	Signed integer comparison gave greater than
BGE	Greater or equal	Signed integer comparison gave greater or equal
BLT	Less than	Signed integer comparison gave less than
BLE	Less or equal	Signed integer comparison gave less than or equal
BHI	Higher	Unsigned comparison gave higher
BLS	Lower or same	Unsigned comparison gave lower or same

Conditional execution of instructions

Conditional branch instructions are just specific case of common feature of conditional execution that can be applied to all ARM assembler instructions. If condition is appended to the instruction, it will only be executed, if condition is true.

```
         cmp r0, #5
         beq DONT
         add r1,r1,r0
         sub r1,r1,r2
DONT     ..
```

```
cmp    r0, #5
addne  r1,r1,r0
subne  r1,r1,r2
```

```
if (r1<10)
    then
        r4=r1+5
    else
        r4=r1+8
```

```
         cmp r1, 10
         blo LOWER
         add r4,r1,#8
         b CONT
LOWER    add r4,r1,#5
CONT     ...
```

```
cmp    r1,#10
addlo  r4,r1,#5
addhs  r4,r1,#8
```

Conditional execution of instructions

```
if ((r0==r1) AND (r2==r3)) then r4=r4+1
```

⬇

```
cmp   r0,r1     ; set Z, if r0=r1
cmpeq r2,r3     ; compare only if Z=1 and
                ;   set Z again, if r2=r3
addeq r4,r4,#1 ; add only if Z=1 (r0=r1 and r2=r3)
```

- majority of if-then-else clauses can be implemented by conditional execution!
- if-then-else clauses with complex conditions (composed with AND or OR) can be implemented more efficiently by conditional execution.
- use of conditional execution is usually more efficient for shorter sequences of instructions than using separate branches!
- Conditional execution takes at least 1 clock cycle regardless of condition being true or false!