16	23	29	19	11
✓	✓	✓		

Instruction Set

Data Transfer	Data Processing	Program control
MOV R1, R0	ADD R0, R1, R3	B Back
STR R0, [R2, #4]	SUB R1, R2, #3	BNE Loop
LDR R1, [R2]	EOR R3, R3, R2	BL Routine

Data Transfer

Register Data Transfer : source operand to destination register

- MOV : source register direct or immediate addressing.

MOV R1, R0	copies R0 to R1
MOVS R0, #0	moves 0 into R0, sets Z flag

S ↳ allows setting values

- MVN (move complement) : source is bitwise inverted

MVN R1, R0	copies R0 (not) into R1
MVNS R0, #0	moves 32 bit value of -1 into R0, sets N flag

$$\text{Memory Data Transfer : memory } \overset{\text{LDR}}{\text{to}} \text{ / } \overset{\text{STR}}{\text{from}} \text{ register}$$

- LDR : memory at address is reffed using various indirect register addressing modes

LDR R1, [R0] copy 32-bit value pointed by R0 to R1
LDRB R1, [R0] copys 8-bit value pointed at by R0 to R1
↳ gets 0 extended. (LSB, little endian)

- STR : register to memory. accessed by indirect addressing modes

STR R1, [R0] copy R1 (32-bits) to address pointed at by R0

STRB R2, [R0, #1]! copy byte in R2 to only 1 address : R0 = R0+1 (post index)

program example : copying block of memory

- block copy : replicates a contiguous segment of memory from one location to another

move 5 words → 32 bit

```
MOV R0, 0x110 ;
MOV R1, 0x210 ;
LDR R2, [R0], #4   ← 2 cyc.
STR R2, [R1], #4   ← 2 cyc
```

LOOP x5

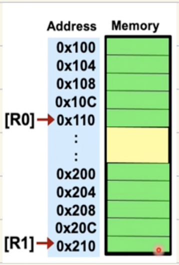
} 4

x5

20 cycles

$$\overset{\text{R0}}{\text{memory}} \rightarrow \overset{\text{R1}}{\text{memory}}$$

LDR ↳ reg ↗ STR



using just 1 pointer

saves 1 cycle

```
MOV R0, #0x100

LDR R2, [R0], #4
STR R2, [R0, #0xFC]
```

offset : (0x200 - 0x100) + 4 compensate.

max immediate offset range = ± 4096

4×2^{10} = 4kilobytes

Data Processing

↳ modifying data through arithmetic, logic, shift operations.

· ADD : ADD a, b, c

$$\begin{tikzcd} a, b & & c \\ \text{reg} \arrow[u] & & \text{reg / immediate value} \arrow[u] \end{tikzcd}$$

ADD S
↳ sets flags.

```
(+ve) 0000  0001   (1)
(+ve) 0111  1111   (127)        V
-----------------------------
      1000  0000   (-128)
```

· overflow : added signed numbers have the same sign, result has the opposite sign

```
                       sign.  unsigned
(+ve)  0000 0001       (1)     (1)        Z
(-ve)  1111 1111       (-1)    (255)      C → "unsigned carry"
       ------------------------------
(+ve)  0000 0000       (0)     (0)
```

· processor doesn't know if it is signed or unsigned, so it sets all possible flags.
∴ if its signed you look at V
unsigned C

· SUB , RSB ← reverse subtract

1 clk cycle ↗

SUB(S) R2, R0, #4 $\quad R2 = \overset{\text{"minuend"}}{R0} - \overset{\text{"subtrahend"}}{4}$

RSB(S) R2, R0, #4 $\quad R2 = 4 - R0$

$A - B = A + \underset{\text{2's complement of B}}{(-B)}$

setting of flags :

· carry : cleared if subtraction borrows
else, sets.

↓ unsigned value of minuend < unsigned value of subtra

unsigned underflow → unsigned(A) < unsigned(B) C=0

unsigned (A) ≥ unsigned(B) C=1

· Overflow : $-2^{31} < (A-B) \geq +2^{31}$: V=1

program example : convert negative to positive
↓
int array of 10 -ve values

```
               MOV   R0, #0x100    ; set-up source pointer
Loop           LDR   R1, [R0]        }
9 times        MVN   R1, R1          }  60 cycles
               ADD   R1, R1, #1      }
               STR   R1, [R0], #4    }
```

↓ better

```
               MOV   R0, #0x100 ;
LOOP           LDR   R1, [R0]        }
               RSB   R1, R1, #0      }  50 cycles
               STR   R1, [R0], #4    }
```

carry based arithmetic instruction

↑
for when you want to add 64 bit numbers.

ADC R2, R0, R1	R2 = R0 + R1 + C
SBC R2, R0, R1	R2 = R0 - R1 + NOT(C)
RSC R2, R0, R1	R2 = R1 - R0 + NOT(C)

↑
add S to set flags.

logical

bitwise boolean

- MVN R2, R2

$$\text{MVN} \curvearrowright \begin{array}{ll} \text{R2} & \boxed{0x00000000} \\ \text{R2} & \boxed{0xFFFFFFFF} \end{array}$$

- AND, ORR, EOR ← (S) → influences N, Z

set to 1 ← ORR R1, R1, #0X000000FF (OR w/ 0 → itself; w/ 1 → 1)
pass w/ 0
↑ mask using 1

$\text{R1}\ \boxed{0x12345678} \rightarrow \text{R1}\ \boxed{0x\underbrace{123456}_{\text{pass}}\underline{FF}}$ ↳ clear

pass w/ 1
set to 0 ← AND (can mask using 0)

pass w/ 0 ← EOR (xor → can use 1 to complement the bit) regardless of sign

multiply by 2^n, shift 0 n times

add S to set C

shift

LSL : logical shift left $0 \to LSB \dashrightarrow MSB \to C$ — MSB goes to carry

LSR : logical shift right $0 \to MSB \dashrightarrow LSB \to C$ — LSB ↗

divide by 2^n shift 0 n times

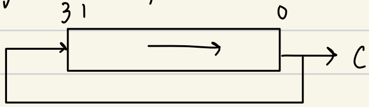
must be unsigned

ASR : arithmetic shift right ∴ unsigned division

assumes signed. repearts MSB : $MSB \to MSB \dashrightarrow LSB \to C$

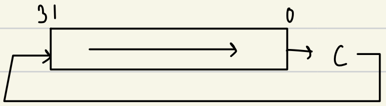
Rotate / Cyclic shift

ROR : rotate right
32 bit



LSB is put in C flag

RRX : rotate right extended
assumes 33 bit



0 replaces C flag, old C flag goes to MSB

— × —

MOV R0, R0, LSL #1	R0 = R0 << 1
ADD R2, R1, R0, LSR #2	R2 = R1 + (R0 >> 1) twice
ADDS R2, R1, R0, LSL R4	R2 = R1 + R0 << R4 set flags
ORRS R0, R0, R0, ROR #1	R0 = R0 | R0>>1 set flags (1 if LSB is 1)

allowed because shift centre (barrel shift) is on the way to the ALU (arithmetic logic unit)

program example: count no. of 1's in register R0

R0 = 0x22222222 (8 1's)

algo: rotate, add LSB to another reg, repeat 32 times

```
             MOV   R2, #0
             MOV   R3, #32            loop counter
LOOP         MOV   R0, R0, ROR #1     rotate right once
31           AND   R1, R0, #1         clear all except LSB (=mask LSB)
times        ADD   R2, R2, R1         add R1 (=LSB) to counter reg.
back         SUB   R3, R3, #1         decrement loop counter
```

↓ optimise

```
             MOV   R2, #0

LOOP         MOVS  R0, R0, LSR #1     sets carry = LSB
back if      ADC   R2, R2, #0         adds carry to R2
not zero
   ↑
```

loop ends as soon as 'R0' becomes all zero ∴ worst case: 32

disadvantage: R0 is completely lost

Program control

changing normal sequential execution flow
↓
modifying content of Program counter

direct ↙ ↘ branch

```
0x50             MOV PC, #0x060          B   CodeB
0x54  codeA      MOV R0, R1
  :                :
0x60  codeB      MOV R0, R2
```

- B : branch, allows jump to ↗ 'nother code
- BCC : conditional branch

CC ← to be filled

```
      BEQ  Skip
Skip  MOV  R1, R0
```

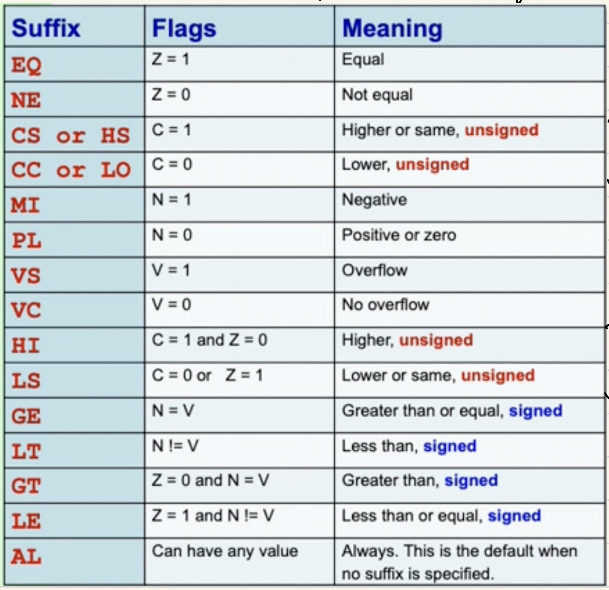
BEQ → branch eq to zero
↓
checks if result of code above = 0
check zero flag

Bcc uses PC-relative addressing mode w/ range ± 32MB

value used is 8 bytes ahead of current execution cc : 15 possible conditions (using flags

B / BAL   doesn't check anything   branch always
BEQ . if Z = 1   branch equal
BVS : if V = 1   branch overflow set

↑
can manipulate by adding a op^n that sets flag using 's'
↓
careful about signed, unsigned



Suffix	Flags	Meaning
EQ	Z = 1	Equal
NE	Z = 0	Not equal
CS or HS	C = 1	Higher or same, unsigned
CC or LO	C = 0	Lower, unsigned
MI	N = 1	Negative
PL	N = 0	Positive or zero
VS	V = 1	Overflow
VC	V = 0	No overflow
HI	C = 1 and Z = 0	Higher, unsigned
LS	C = 0 or Z = 1	Lower or same, unsigned
GE	N = V	Greater than or equal, signed
LT	N != V	Less than, signed
GT	Z = 0 and N = V	Greater than, signed
LE	Z = 1 and N != V	Less than or equal, signed
AL	Can have any value	Always. This is the default when no suffix is specified.

} unsigned

} unsigned

} signed comparison

program example

```
MOV   R2, #0
MOV   R3, #32
           ↓ 31

Loop
LOOP   MOV   R0, R0, ROR #1
31     AND   R1, R0, #1
times  ADD   R2, R2, R1
back   SUBS  R3, R3, #1
       BNE   Loop      ← if 0     branch not equal
       BPL   Loop      ← if -ve   branch positive / zero
```

testing signed values

```
SUBS  R1, R1, R2       R1 = R1 - R2
BGE   R1≥R2            jump to R1≥R2 if result is +ve
  ⋮
R1≥R2
```

testing unsigned values

```
SUBS  R1, R1, R2       R1 = R1 - R2
BHS   R1≥ R2           jumpto R1≥R2 if R1 higher/equal
                                       to R2
R1≥R2
```

- setting flags w/o changing values
  ↓
  CMP not SUBS

```
CMP   R1, #4        test (R1-4) ; R1 is signed
BGE   R1 ≥ 4        if +ve, branch
```

R1 ≥ 4

finding c string terminator in memory pointed to by R0

```
Loop   LDRB   R1, [R0],#1     read mem byte using post-ind
       CMP    R1, #0          compare w/ 0  ∴ (R1 - 0)
       BEQ    Found           if value = 0
       B      Loop            branch back to loop
Found  :
```

- other conditional test instructions

CMN	R0, R1	R0+R1	compare -ve	N,Z,C,V
TST	R0, R1	R0 AND R1	test bits	N,Z,C
TEQ	R0, R1	R0 EOR R1	test equi	N,Z,C,

program example : find largest number (FindMax)

- find largest unsigned
- store in R3
- array of 10 int , starts at 0x100

- loop count reg R1
  temp store max reg R3
  temp current reg R2
  pointer reg R0

```
        MOV   R0, #0x100     pointer
        MOV   R1, #9         loop counter
        LDR   R3, [R0]       first number is max
Loop    ADD   R0, R0, #4
        LDR   R2,[R0]
        CMP   R2, R3          R2 - R3
                                          R2≤R3
        BLS   Skip            lower/same       C=0 or Z=1
        MOV   R3, R2

Skip SUBS  R1, R1, #1
     BNE   Loop
```

```
      MOV    R0, #0x100       pointer
      MOV    R1, #9           loop counter
      LDR    R3, [R0]         first number is max
Loop  LDR    R2, [R0, #4]!

      CMP    R2, R3           R2 - R3
                              lower/same  R2≤R3  C=0 or Z=1

      MOVHI  R3, R2           if R2 > R3

Skip  SUBS   R1, R1, #1
      BNE    Loop
```

knowledge ↓

Conditional Execution

instructions can be conditional

```
if (R0==1)         CMP    R0, #1    }        CMP     R0, #1
  R1=3             BNE    Else      }        MOVEQ   R1, #3    Z=1  equal
else               MOV    R1, #3    }  →     MOVNE   R1, #5    Z=0  not equal
  R1=5             B      Skip      }
             Else  MOV    R1, #5    }
             Skip  ⋮
```