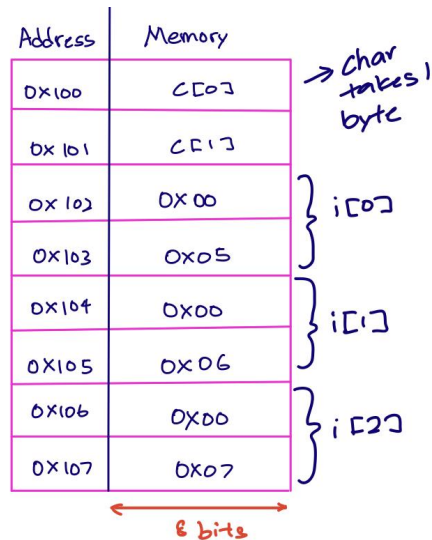


# Array Representation

```
main() {  
    char c[2]; // Instantiating char array C with 2 index  
    C[0] = 'A'; // Assigning 'A' to 1st index  
    C[1] = 'B'; // Assigning 'B' to 2nd index  
    Short int i[3]; // Instantiating short int array i with 3 index  
    i[0] = 5; // Assigning 5 to 1st index  
    i[1] = 6; // Assigning 6 to 2nd index  
    i[2] = 7; // Assigning 5 to 3rd index  
}
```



Short int takes 2 bytes

## Basic Execution Cycle

// R1: Destination, R0: Source

 MOV R1, R0 // Copy R0 content into R1; register to register; R0 value remains the same

 LDR R1, [R0] // Copy memory content of the address in R0 into R1; memory to register


 MOV R1, #3 // Copy immediate value #3 into R1

// R1: Source, R0: Destination


 STR R1, [R0] // Copy R1 register content to memory pointed to by address in R0

## Basic Execution Cycle


// Register Indirect with Offset

 LDR R1, [R0, #4] // Copy memory content of where address in R0 with an offset of 4 is pointing to and put it inside R1 (e.g. R0 = 0x100 -> 0x104); R0 value does not change


// Register Indirect with Index Register

 LDR R1, [R0, R2] // Copy memory content of where address in R0 + R2 is pointing to and put it inside R1 (e.g. R0 = 0x100, R2 = 0x004, [R0, R2] = 0x104); R0 & R2 value does not change

## Register Indirect with Offset

  
LDR R1, [R0, #4] // Copy memory content of where address in R0 with an offset of 4 is pointing to and put it inside R1 (e.g. R0 = 0x100 -> 0x104); R0 value does not change

## Register Indirect with Index Register

  
LDR R1, [R0, R2] // Copy memory content of where address in R0 + R2 is pointing to and put it inside R1 (e.g. R0 = 0x100, R2 = 0x004, [R0, R2] = 0x104); R0 & R2 value does not change

## Accessing Array Elements

  
MOV R2, #0x100 // Instantiating base address into R2

  
MOV R1, #7 // Instantiating value 7 into R1

  
STR R1, [R2, #0] // Store value in R1 (#7) into R2 base address (#0x100)

  
STR R1, [R2, #16] // Store value in R1 (#7) into R2 base address with offset 16 (#0x116)


## Cleaning Array Elements

  
MOV R2, #0x100 // Instantiating base address into R2

  
MOV R0, #0 // Instantiating value 0 into R0


  
MOV R1, #0 // Instantiating value 0 into R1

⋮  
⋮


  
STR R0, [R2, R1] // Store value in R0 (#0) into R2 + R1 ( $\#0x100 + \#0 = \#0x100$ )  
// #0 copied into #0x100 - 0x103

  
STR R1, R1, #4 // Increments value/index of R1 by 4, loops till it is finished

## Offset with Auto-Indexing (Pre-indexing)

  
LDR R1, [R0, #4]! // Update R0 content (0x100) with offset #4 (becomes #0x104), take memory content at address 0x104 and put into R1; value in R0 changes (increment by 4)

## Index Register with Auto-Index

  
LDR R1, [R0, R2]! // Update R0 content (0x100) with R2 ( $R1 + R2 = \#0x104$ ), take memory content at address 0x104 and put into R1; value in R0 changes (increment by 4)

Purpose of ! : set register to its new address value based on its last accessed address for the operation.

For e.g. for the code on the left, the address accessed was 0x104, whereas the original address of R0 is 0x100, after the operation, ! updates to #0x104

## Clearing all Array Elements with Auto-indexing

  
MOV R2, #0x100 // Instantiating base address into R2

  
MOV R1, #0 // Instantiating value 0 into R1

  
STR R1, [R2] // Stores value of 0 from R1 to index R2

⋮  
⋮

  
STR R1, [R2, #4]! // Store value in R1 (#0) into R2 (using current effective address + 4),  
then put this effective address into R2



## Pre-index

  
LDR R1, [R0, #4]! // offset with auto-indexing ->  $R0 = R0 + 4$ ;  $R1 = \text{mem}[R0]$

  
LDR R1, [R0, R2]! // index with auto-indexing ->  $R0 = R0 + R2$ ;  $R1 = \text{mem}[R0]$

\* In pre-index, the indirect register is auto-indexed **before** being used to compute effective address

## Post-index

LDR R1, [R0], #4 // offset with auto-indexing ->  $R1 = \text{mem}[R0]$ ;  $R0 = R0 + 4$

LDR R1, [R0], R2 // index with auto-indexing ->  $R1 = \text{mem}[R0]$ ;  $R0 = R0 + R2$

\* In post-index, the indirect register is used to compute the effective address after it is auto-indexed

## Absolute Jump

Address	Absolute Jump
0x050	MOV PC, #0x060
0x054	CodeA    MOV R0, R1
:	:
0x060	CodeB    MOV R0, R2

MOV PC, #0x060 // skips you to address #0x060 to run code B

## Relative Jump

Address	Relative Jump
0x050	B CodeB
0x054	CodeA    MOV R0, R1
:	:
0x060	CodeB    MOV R0, R2

B CodeB // Branch instruction with appropriate signed offset; skips execution of CodeA segment

## Register Data Transfer

  
MOV R1, R0 // Copy R0 to R1

  
MOVS R0, #0 // Move 0 into R0 and set Z flag

  
MVN R1, R0 // R1 = NOT(R0)

  
MVN R1, #0 // sets R1 to -1 (NOT(0x00000000) = 0xFFFFFFFF = -1)

  
MVN R1, #1 // sets R1 to -2

  
LDR R1, [R0] // copy 32-bit (4 bytes) value pointed by R0 into R1

  
LDRB R1, [R0] //copy 8-bit (1 byte) value pointed by R0 into R1

  
STR R1, [R0] // copy R1 (4 bytes) starting at address pointed by R0

  
STRB R2, [R0, #1]! // Copy byte in R2 to only 1 address at [R0+1], then R0 = R0 + 1

## Arithmetic Instructions

ADDS R2 , R0 , #8    //  $R2 = R0 + \#8$

SUBS R2 , R0 , #4    //  $R2 = R0 - \#4$

RSBS R2 , R0 , #4    //  $R2 = \#4 - R0$  (reverse subtraction)

All the “s” suffixes has been added to all these instructions to influence the flags

## CPSR Condition Code Flags

CPSR bit(s)	Name	Description	NZVC		CPSR
31	N	Can set if last operation produced a <b>negative</b> result			
30	Z	Can set if last operation produced a <b>zero</b> result			
29	C	Can set if last operation produced a <b>carry out</b> in the most significant bit			
28	V	Can set if the last operation produced an <b>overflow</b> for a <b>signed</b> arithmetic operation			

## Carry-based Arithmetic Instructions

  
ADC R2 , R0, R1    //  $R2 = R0 + R1 + C$  (ADD with CARRY)

  
SBC R2 , R0, R1    //  $R2 = R0 - R1 + \text{NOT}(C)$  (SUB with CARRY)

  
RSC R2 , R0 , R1    //  $R2 = R1 - R0 + \text{NOT}(C)$  (RSB with CARRY)

## Different Bcc Conditions

Suffix	Flags	Meaning	
EQ	Z = 1	Equal	
NE	Z = 0	Not equal	
CS or HS	C = 1	Higher or same, <b>unsigned</b>	<b>Unsigned comparison</b>
CC or LO	C = 0	Lower, <b>unsigned</b>	
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, <b>unsigned</b>	<b>Unsigned comparison</b>
LS	C = 0 or Z = 1	Lower or same, <b>unsigned</b>	
GE	N = V	Greater than or equal, <b>signed</b>	<b>Signed comparison</b>
LT	N != V	Less than, <b>signed</b>	
GT	Z = 0 and N = V	Greater than, <b>signed</b>	
LE	Z = 1 and N != V	Less than or equal, <b>signed</b>	
AL	Can have any value	Always. This is the default when no suffix is specified.	

## Logical Instructions

SET R0 = 0101 0101

AND R1 , R0 , #1111 0000 // Use 0 as a mask to clear bit -> R1 = 0101 0101

ORR R0 , R0 , #1111 0000 // Use 1 as a mask to set bit -> R0 = 1111 0101

EOR R2 , R0 , #1111 0000 // Use 1 as a mask to complement bit -> R2 = 1010 0101

**AND** truth table

A	B	Z = A . B
0	0 *	0
0	1	0
1	0 *	0
1	1	1

\* Binary **0 mask** is used to **clear** the bit

**OR** truth table

A	B	Z = A+B
0	0	0
0	1 *	1
1	0	1
1	1 *	1

\* Binary **1** mask is used to **set** the bit

**EX-OR** truth table

A	B	Z = A ⊕ B
0	0	0
0	1 *	1
1	0	1
1	1 *	0

\* Binary **1** mask is used to **complement** the bit



## Logical Shift Left



**SIGNED OR UNSIGNED MULTIPLY - Multiply source register \*  $2^n$  :  $n$  = num of bits shifted**

LSL R3, R3, #1 // Binary "0" is shifted into LSB of R3 from the right - Shift R3 left by 1 bit

MOV R0, R0, LSL #1 // Shift R0 left by 1 bit ->  $R0 = R0 \ll 1$  ( $R0 = R0 * 2^1$ )

ADDS R2, R1, <sup>+</sup>R0, LSL R4  
// Shift R0 by R4 bits before ADD with R1 and UPDATE N Z V C flags and result in R2 ->  $R2 = R2 + R0 \ll R4$

## Logical Shift Right



**UNSIGNED DIVIDE - Divide source register \*  $2^n$  :  $n$  = num of bits shifted**

LSR R3, R3, #1 // Binary "0" is shifted into MSB of R3 from the left - Shift R3 right by 1 bit

MOV R0, R0, LSR #1 // Shift R0 left by 1 bit ->  $R0 = R0 \gg 1$  ( $R0 = R0 / 2^1$ )

ADD R2, R1, <sup>+</sup>R0, LSR #2  
// ADD R1 with 2-bit right shifted R0. Put result in R2

## Arithmetic Shift Right



### **SIGNED DIVIDE**

`ASR{S}{cond} Rd, Rm, Rs`

`ASR{S}{cond} Rd, Rm, #sh`

- S - optional suffix
- Rd - destination register
- Rm - register holding the first operand. This operand is shifted to the right
- Rs - register holding a shift value to apply to the value in Rm. Only LSB is used
- sh - is a constant shift. Range of values permitted is 1 - 32

## Rotate Right



**ROR**

The bit shifted out of register is returned in at the leftmost end and is also placed into the c flag

Cyclical shift , as no bits in register is lost during the shifting operation

  
`ORRS R0 , R0 , R0 , ROR #1` // OR R0 with a 1-bit right rotated version of itself. Set c flag is  
bit rotated out is 1 ->  $R0 = R0 \mid R0 \gg 1$

## Rotate right extended



The C-flag is shifted into the register at the leftmost end, while the bit shifted out replaces the current C-flag

```
RRX{S}{cond} Rd, Rm
```

S - optional suffix

Rd - destination register

Rm - register holding the first operand. This operand is shifted right

## CMP Instructions

Use CMP instead of SUBS to compare values of 2 operands without affecting the operands and sets the CC flags according to the results

SUBS R1 , R1 , R2    // R1 = R1 - R2

BGE R1≥R2            // jump to R1≥R2 if the result is positive

R1≥R2

[Without affecting the operands]

CMP R1 , #4            // test (R1 - 4), where R1 is a signed no.

BGE R1≥4            // branch to R1 ≥ 4 if result is positive

R1≥4            :

## Other Conditional Test Instructions

**They do not need “s” suffix to influence the condition code flags**

CMN R0 , R1      // set (N,Z,C,V) based on R0 + R1 (Compare Negatives)

TST R0,R1        // set (N,Z,C) based on R0 AND R1 (Test Bits)

TEQ R0,R1        // set (N,Z,C) based on R0 EOR R1 (Test Equivalence)