

Basic Offset

LDR/STR Rt, [Rn]

Pre-index Offset

LDR/STR Rt, [Rn, #offset]

Post-index Offset

LDR/STR Rt, [Rn, #offset]!

PC-relative

LDR/STR Rt, [Rn], #offset

Pseudo-instruction

LDR Rd, LABEL

LDR Rd, =value or =LABEL

# Eg 7: Memory Addressing

## ➤ The various addressing modes of LDR instruction are

- Offset addressing: **LDR R2,[R1]; R4,[R3],#4; R5,[R1,#4]; R6,[R4,#4]!** @ loads **memory contents**
- PC-relative addressing: **LDR R1, NUM1** @ loads the **memory contents** referenced by the label
- Pseudo-instruction: **LDR R3, =NUM1** @ loads the **memory address** associated with the label

## ➤ Updated registers:

	Main Memory			Registers	
	0x00000100	LDR R1, NUM1		PC	
	0x00000104	LDR R2, [R1]		:	
	0x00000108	LDR R3, =NUM1		R1	<b>0x100</b>
	0x0000010C	LDR R4, [R3], #4		R2	<b>LDR R1, NUM1</b>
	0x00000110	LDR R5, [R1, #4]		R3	<b>0x00008004</b>
	0x00000114	LDR R6, [R4, #4]!		R4	<b>0x104</b>
	0x00000118	STR R7, [R3]		R5	<b>LDR R2, [R1]</b>
	:	:		R6	<b>LDR R2, [R1]</b>
<b>NUM1:</b>	0x00008000	0x100		R7	0x7
	0x00008004	<b>0x7</b>		:	

# Eg 8: Memory Addressing Application

➤ a. Using the assembly instructions you have just learned, find the sum of **(A+5) + (B+10) + C** & store the result in memory location ANS. Memory locations A & B hold the value of 2 numbers provided by peripheral devices in another subsystem, while C is a time-varying signal & its most updated value will only be available to you just before you assemble your code. *[Unless specified otherwise, you may assume all variables are .word-declared by default.]*

➤ b. What if the numbers 5 & 10 above are stored in the memory instead? At memory address of A +4 & +8 respectively. How would you modify the program to retrieve them & store the new sum? *[Hint: pre- or post-index, or both?]*

		Memory
A:	0x..40	A's value
	0x..44	5
	0x..48	10

## Eg 8a: We want $ANS \leftarrow (A+5) + (B+10) + C$ , one possible solution...

.EQU C, xxx	@	.equ is always located near the top of the code, so we can conveniently key in the last-minute entry 'xxx'
LDR R1, A	@	PC-Relative load R1 with the value of A
LDR R2, B	@	another PC-Relative load as above
LDR R3, =C	@	we want the assembler to substitute C's value & we want it <b>literally</b> in R3, hence use Pseudo-Instruction
ADD R1, #5	@	R1 updated
ADD R2, #10	@	R2 updated
ADD R4, R1, R2	@	sum the updated R1 & R2, & store the result in R4
ADD R0, R3, R4	@	sum the previous result R4 with R3 & store the result to the default "output" register, R0
@ STR R0, ANS	@	at this point we might be prone to issue a PC-Relative STR; but unfortunately, ARMv7E-M does not support it, unlike the more advanced versions. Instead, we'll have to substitute it with the following two instructions:
LDR R5, =ANS	@	load R5 with the address of ANS (since we <b>literally</b> want the address, use Pseudo-Instruction, with the equal sign)
STR R0, [R5] .LCOMM ANS 4	@	then load the final result in R0 to the memory location pointed by R5, which holds the address of ANS.

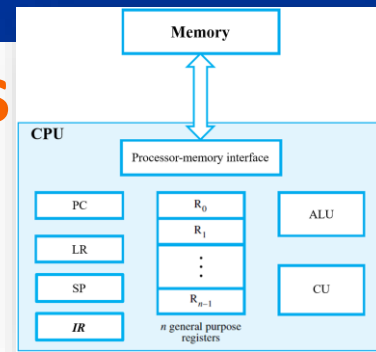
## Eg 8b

**Eg 8b. We want  $ANS \leftarrow (A+5^\#) + (B+10^\#) + C$ ; *#in memory*. One possibility...**

.EQU C, xxx	@	.equ is always located near the top of the code, so we can conveniently key in the last-minute entry 'xxx'
LDR R1, =A	@	Pseudo-Instruction load R1 with the address of A & use it as a pointer
LDR R2, B	@	PC-Relative load R2 with B's value
LDR R3, =C	@	we want the assembler to substitute C's value & we want it <b>literally</b> in R3, hence use Pseudo-Instruction
LDR R4, [R1], #4	@	Post-Index load R4 with A's value, pointer R1 incremented by 4
LDR R5, [R1]	@	Load R5 with value in address of A+4 (e.g. 5)
ADD R4, R5	@	$R4 = A + \text{value in address of } A+4 \text{ (e.g. 5)}$
LDR R6, [R1, #4]!	@	Pre-index load R6 with value pointed by R1(A+4) +4, (e.g. 10)
ADD R6, R2	@	$R6 = B + \text{value in address } A+8 \text{ (e.g. 10)}$
ADD R4, R6	@	$R4 = R4 + R6$
ADD R0, R3, R4	@	sum the previous result R4 with R3 & store the result to the default "output" register, R0
LDR R5, =ANS	@	load R5 with the address of ANS (since we <b>literally</b> want the address, use Pseudo-Instruction, with the equal sign)
STR R0, [R5] .LCOMM ANS 4	@	then load the final result in R0 to the memory location pointed by R5, which holds the address of ANS

# 3. ARMv7E-M Ctrl & Arithmetic Instructions

## 3.1 Move Instructions



### ➤ MOV

➤ Assembly language format:

**MOV{S}      *Rd, Op2***  
**MOV{S}      *Rd, #imm16***



e.g.

ADD R0, R1, **#0xFF**  
ADD R0, R1, **R2**  
ADD R0, R1, **R2, LSL #0x4**

*Operand2*

### ➤ Examples:

MOV      *Rd, Rm*  
performs  
 $Rd \leftarrow Rm$

MOVS    *Rd, #value*  
performs  
 $Rd \leftarrow \text{value}$






N & Z are updated accordingly; C may be updated based on the result of *Op2* (e.g. LSL #0x4); V is not affected

### LDR or STR vs MOV:

**LDR or STR** is for transfers to/from a **register** from/to **memory** (hence the [ ]) respectively, while **MOV** is to transfer to a **register** (*Rd*) an immediate constant value or from another **register**

# 3.1 Move Instructions

## ➤ MOV vs LDR/STR : *When to use what?*

Common Scenarios:	Preferred:	Bad Ideas:	Wrong Ideas:
1. I need a known <b>constant</b> (e.g. 314) to be in a <b>register</b> (e.g. R1)	MOV R1, #314 <i>or</i> LDR R1, =314 <i>or</i> .equ Pi, 314 LDR R1, =Pi	LDR R1, CONST CONST: .word 314	LDR R1, #314 
2. There is a <b>constant</b> in the <b>memory</b> (e.g. CONST) that I can use & I need it to be in R1	LDR R1, CONST CONST: .word 314	LDR R0, CONST MOV R1, R0 CONST: .word 314	MOV R1, CONST CONST: .word 314 
3. There is a <b>result</b> in another <b>register</b> (e.g. R0) that I can use & I need it to be in R2	MOV R2, R0	STR R0, ANS LDR R2, ANS	LDR R2, R0 
4. There is a <b>result</b> in the <b>memory</b> (e.g. ANS) that I can use & I need it to be in R2	LDR R2, ANS	LDR R0, ANS MOV R1, R0	MOV R2, ANS STR ANS, R2 
5. I need to set up a <b>pointer</b> (e.g. with R3)	LDR R3, =NUM1	LDR R0, =NUM1 MOV R3, R0	MOV R3, =NUM1 <i>or</i> STR =NUM1, R3 

## 3.2 Arithmetic Instructions: ADD, SUB

### ➤ ADD & SUB (Add & Subtract)

### ➤ Assembly language format:

**ADD{S}/SUB{S} {Rd,} Rn, Op2** →

or

**ADD{S}/SUB{S} {Rd,} Rn, #imm12**

If *Rd* is omitted, destination register is *Rn*

e.g.

ADD R0, R1,

#0xFF

ADD R0, R1,

R2

ADD R0, R1,

R2, LSL #0x4

*Operand2*

### ➤ Examples:

ADDS R0, R2, R4

while

SUB R3, #6

performs

performs

$R0 \leftarrow R2 + R4$

$R3 \leftarrow R3 - 6$

& NZCV are updated accordingly  
based on the sum in R0

# 3.3 Arithmetic Instructions: MUL, MLA



➤ **MUL & MLA** (Multiply & Multiply with Accumulate, **32-bit result**)

➤ Assembly language format:

**MUL{S} {Rd,} Rn, Rm**      If *Rd* is omitted, destination register is *Rn*

**MLA{S} Rd, Rn, Rm, Ra**

Example:

MUL    R0, R1, R2      *while*  
performs  
 $R0 \leftarrow R1 \times R2$

MLA    R0, R4, R5, R6  
performs  
 $R0 \leftarrow (R4 \times R5) + R6$

- **MUL**: only the low-order 32 bits of the 64-bit product are written to the destination R0. If the operands are signed, the product will be signed also. The two's-complement value in R0 is correct if the product fits into 32 bits
- **MLA**: only the low-order 32 bits of the 64-bit result are written to the destination R0
- These 32 bits do not depend on whether signed or unsigned calculations are performed, i.e. the input operands' signedness is ignored



# 3.3 MUL, MLA Related: -L, MLS, S/UDIV

- To get **64-bit products**, use the **long** versions that come with the **L** suffix, in either unsigned (**U**) or signed (**S**) variants, e.g. **UMULL**, **UMLAL**, **SMULL**, & **SMLAL**
- They all **treat** both their input operands as unsigned/signed integers, **regardless** of the input operands' actual binary representations (i.e. signedness)
- The counterpart of MLA: Multiply with Subtract (**MLS**) instruction  
e.g. `MLS R0, R4, R5, R6` performs  $R0 \leftarrow R6 - (R4 \times R5)$
- **Division** can either be SDIV or UDIV (Signed/Unsigned Divide)
- Assembly language format:  
**SDIV {Rd,} Rn, Rm**    or    **UDIV {Rd,} Rn, Rm**    @  $Rd \leftarrow Rn / Rm$
- divides a 32-bit signed/unsigned integer register value (dividend, Rn) by a 32-bit signed/unsigned integer register value (divisor, Rm), & writes the result to the destination register, Rd or Rn if Rd is omitted
- The condition code flags are not affected, hence **no S** suffix option

*Did you know we can also do multiplication/division without using these instructions?*

## 3.4 Compare Instructions

### ➤ **CMP & CMN** (Compare & Compare Negative)

### ➤ Assembly language format:

**CMP**    *Rn, Op2*

**CMN**    *Rn, Op2*

If **S** suffix option is **not available** to the instruction before a conditional branch, it is very useful to **CMP/CMN** !

### ➤ CMP performs:                      *while*                      CMN performs:

**Rn - Op2**

**Rn + Op2**

& NZCV are updated accordingly based on the result

### ➤ CMP & CMN are similar to SUBS & ADDS respectively, but do you know there is a **big difference**?

## 3.5 Branch Instructions: B, BL, BLX, BX

➤ Assembly language format:

**B{cond}**      *label*

**BL{cond}**      *label*

**BLX{cond}**      *Rm*

**BX{cond}**      *Rm*

**B**: if-else/switch-case; for/while/do-while loop; goto  
**BL** or **BLX**: from a Caller to a Callee function  
e.g. to jump to a Subroutine/Function from Main  
**BX**: from a Callee back to the Caller function  
e.g. to go back to Main, from a Subroutine/Function

where:

- **L**: branch with **link**, i.e. writes the address of the next instruction (in **PC**) to **LR**
- **X**: branch **indirect**, via register **Rm** that indicates the **address** to branch to
- *label*: a PC-relative expression indicating the **address** to branch to
- *cond*: an optional **condition code suffix** for conditional execution
- **B{cond}** is the only conditional instruction that can be either inside or outside an IT block. All other branch instructions must be **unconditional outside** an IT block (& must be conditional inside the IT block). (*More on IT block later ...*)

## 3.5 Branch Instructions: B

- **B** Branch (immediate)
- Assembly language format:

**B{cond}      label**

performs: branch to location indicated by *label*, when condition(s) specified by **condition code suffix {cond}** is(are) met

$PC \leftarrow label$

- Example:

```
CMP    R0, R1
BEQ    IFEQUAL
```

@ Some instructions for R0, R1 not equal  
:

```
B      SKIPEQUAL
```

IFEQUAL: @ Some instructions for R0, R1 equal  
:

SKIPEQUAL: @ Continue onto the rest of the program

**If** (cond)  
**Then**  
 {instructions for Then}  
**Else**  
 {instructions for Else}  
 :  
 {followed by common instructions}

- **B** branches to instructions at label IFEQUAL or SKIPEQUAL when R0 & R1 are equal, or R0 & R1 are not equal, respectively

## 3.5 Branch Instructions: BL, BLX

- **BL** Branch with **Link** (immediate)

- Assembly language format:

**BL**{*cond*} *label*

performs: branch to location indicated by *label* &  
write the address of the next instruction to **LR**,  
usually when calling a subroutine/function

$LR \leftarrow PC; PC \leftarrow label$

**LR** is referred to as **R14**,  
**PC** is referred to as **R15**  
in some literature

- **BLX** Branch **Indirect** with **Link** (Register)

- Assembly language format:

**BLX**{*cond*} *Rm*

performs: branch to location indicated by *Rm* &  
write the address of the next instruction to **LR**,  
usually when calling a subroutine/function

$LR \leftarrow PC; PC \leftarrow Rm$

## 3.5 Branch Instructions: BX

➤ **BX** Branch **I**ndirect (Register)

➤ Assembly language format:

**BX**{*cond*} *Rm*

performs: branch to location indicated by *Rm*

$PC \leftarrow Rm$

Example: BX LR

➤ Branch Instructions Summary Example:

Main program

```
...
; R0 = X, R1 = Y, R2 = Z
BL    function 1
```

Subroutine/Function

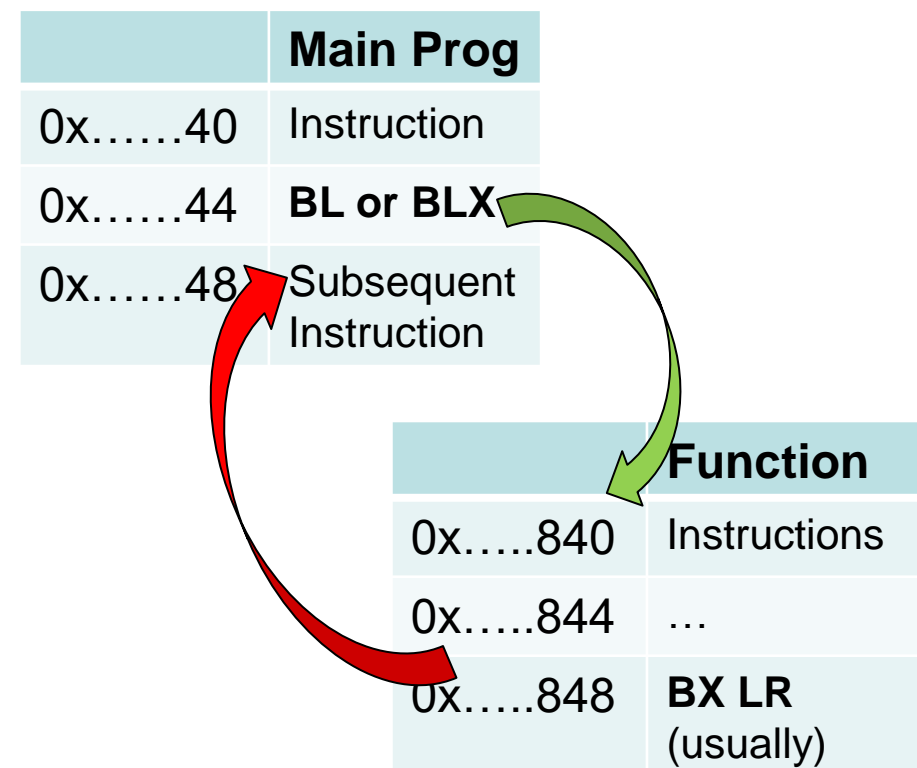
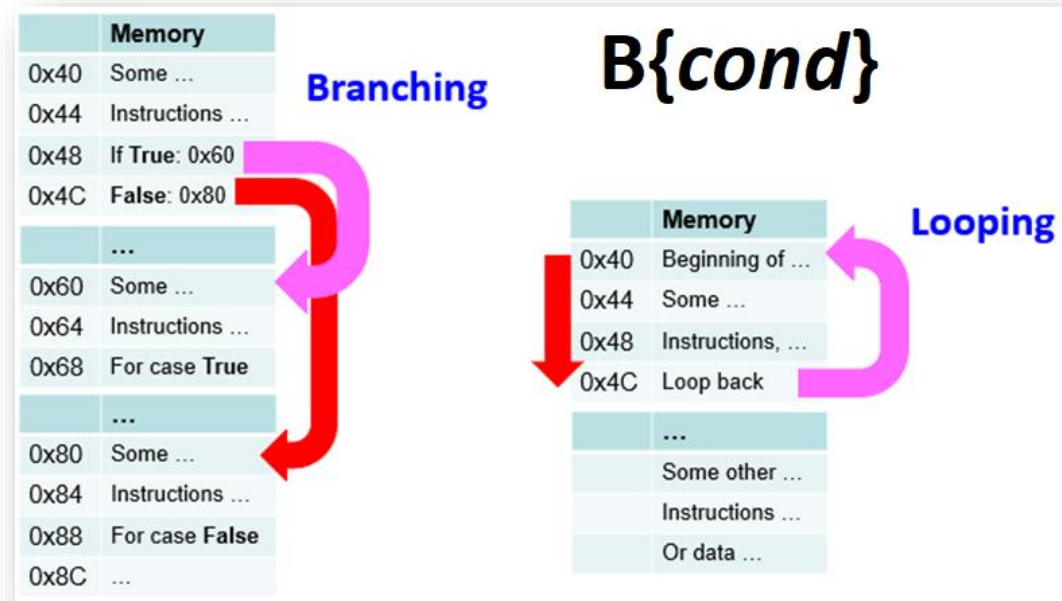
function 1

```
PUSH    {R0-R2} ; Store R0, R1, R2 to stack
... ; Executing task (R0, R1 and R2
    ; could be changed)
POP     {R0-R2} ; restore R0, R1, R2
BX      LR    ; Return
```

```
; Back to main program
; R0 = X, R1 = Y, R2 = Z
... ; next instructions
```

# 3.5 Branch Instructions: Summary

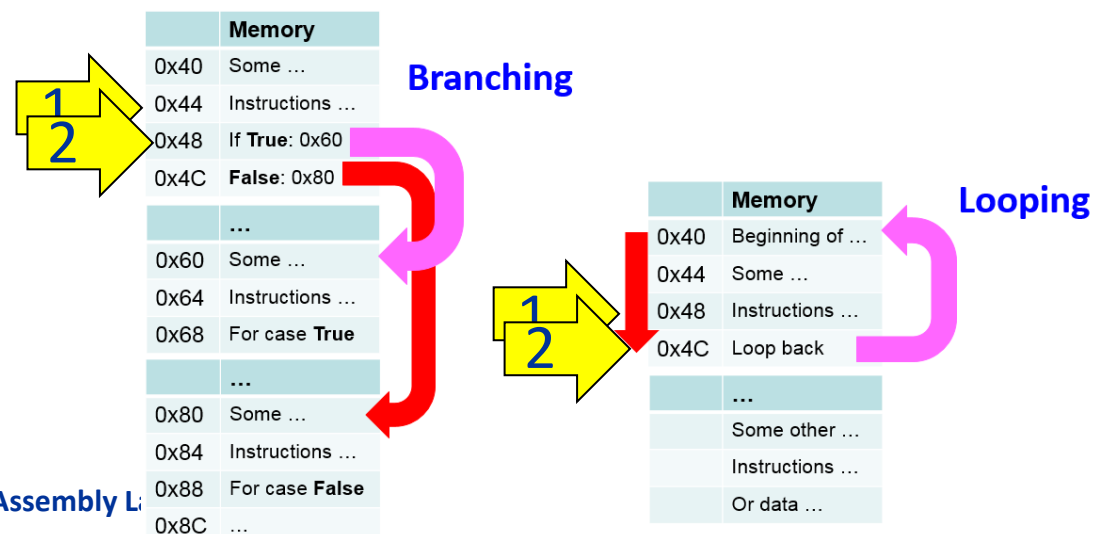
- *When to use what?* Usually, in situations similar to:
- ✓ If-Else/Switch-Case branch; For/While loop: **B{cond}**
  - ✓ To jump to a Subroutine/Function from Main: **BL** or **BLX**
  - ✓ To go back to Main, from a Function: **BX**



# 4. Conditional Execution

- Flow of a program can be altered by a **two-step** process:
- Step 1: Perform either **comparison/test** or **arithmetic/logic/move instructions** (with **suffix S** specified) to cause the condition flags (N,Z,C,V) in APSR\* to be updated accordingly
- Step 2: Use **Condition Code Suffixes** in **branch instruction/IF-THEN (IT) instruction block** to perform conditional execution of subsequent instructions

\*Application Program Status Register





# 4.1 Condition Code Suffixes

**Table A.2** Condition Code Suffixes

Suffix	Flags	Meaning
EQ	$Z = 1$	Equal
NE	$Z = 0$	Not equal
CS or HS	$C = 1$	Higher or same, unsigned $\geq$
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 $\leq$
GE	$N = V$	Greater than or equal, signed $\geq$
LT	$N \neq V$	Less than, signed $<$
GT	$Z = 0$ and $N = V$	Greater than, signed $>$
LE	$Z = 1$ or $N \neq V$	Less than or equal, signed $\leq$
AL	Can have any value	Always; default when no suffix is specified

## 4.2 Conditional Execution: Branch\_1

...Recall for **B**, Branch (immediate)

➤ Assembly language format:

**B{cond}      label**

performs: branch to location indicated by *label*, when condition...

*Actually performs:*

branch to location indicated by *label*

if and only if the condition flags satisfy {cond}

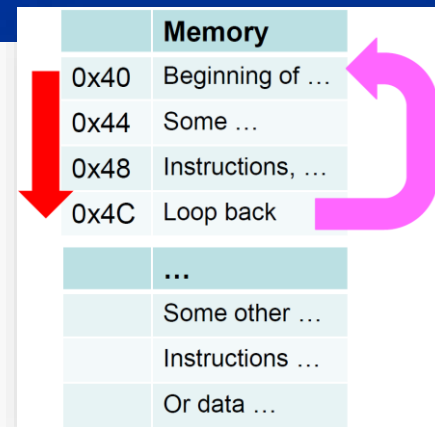
$PC \leftarrow label$  (if and only if the **flags satisfy {cond}**) !

➤ Example:

**BEQ      LOCATION**

branches to label LOCATION if **Z=1**

➤ A **loop** can thus be implemented if LOCATION refers to the **first line of the instruction block** that is to be repeated, & for as long as Z=1 when the instruction block ends, or



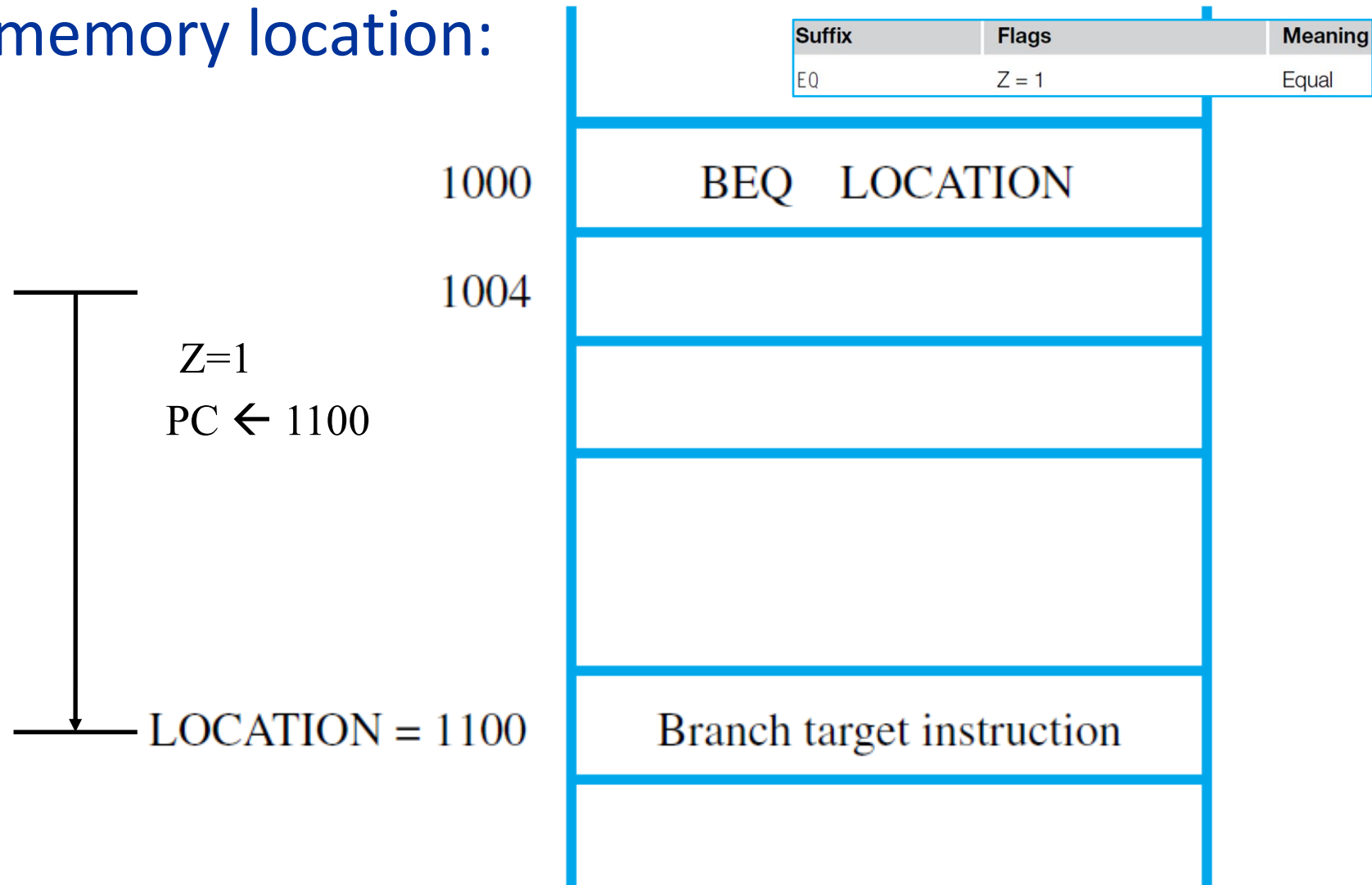
A diagram showing a memory layout with addresses 0x40, 0x44, 0x48, and 0x4C. A red arrow points down from 0x40 to 0x4C, and a pink arrow points from 0x4C back to 0x40, indicating a loop back.

	Memory
0x40	Beginning of ...
0x44	Some ...
0x48	Instructions, ...
0x4C	Loop back
	...
	Some other ...
	Instructions ...
	Or data ...

Suffix	Flags	Meaning
EQ	Z = 1	Equal

## 4.2 Conditional Execution: Branch\_2

- Alternatively, it can also be used to **branch** to another memory location:



## 4.3 Conditional Execution: IT Block\_1

- IF-THEN (IT) block allows efficient branching whenever If-Then-Else conditions are needed:

**Table 4.32** Various Length of IT Instruction Block

	IT Block (each of <x>, <y> and <z> can either be T [true] or E [else])	Examples
Only one conditional instruction	IT <cond> instr1<cond>	IT EQ ADDEQ R0, R0, R1
Two conditional instructions	IT<x> <cond> instr1<cond> instr2<cond or ~(cond)>	ITE GE ADDGE R0, R0, R1 ADDLT R0, R0, R3
Three conditional instructions	IT<x><y> <cond> instr1<cond> instr2<cond or ~(cond)> instr3<cond or ~(cond)>	ITET GT ADDGT R0, R0, R1 ADDLE R0, R0, R3 ADDGT R2, R4, #1
Four conditional instructions	IT<x><y><z> <cond> instr1<cond> instr2<cond or ~(cond)> instr3<cond or ~(cond)> instr4<cond or ~(cond)>	ITETT NE ADDNE R0, R0, R1 ADDEQ R0, R0, R3 ADDNE R2, R4, #1 MOVNE R5, R3

The conditions can be the same or the **logical inverse** !

**Not more than 4 instructions!**

## 4.3 Conditional Execution: IT Block\_2

Here is an example of IT use:

```
if (R0 equal R1) then {  
    R3 = R4 + R5  
    R3 = R3/2  
} else {  
    R3 = R6 + R7  
    R3 = R3/2  
}
```

**Table A.2** Condition Code Suffixes

Suffix	Flags	Meaning
EQ	Z = 1	Equal
NE	Z = 0	Not equal
CS or HS	C = 1	Higher or same, unsigned $\geq$
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 $\leq$
GE	N = V	Greater than or equal, signed $\geq$
LT	N $\neq$ V	Less than, signed $<$
GT	Z = 0 and N = V	Greater than, signed $>$
LE	Z = 1 or N $\neq$ V	Less than or equal, signed $\leq$
AL	Can have any value	Always; default when no suffix is specified

This can be written as follows:

```
CMP R0, R1          ; Compare R0 and R1  
ITTEE EQ            ; If R0 equal R1, Then-Then-Else-Else  
ADDEQ R3, R4, R5    ; Add if equal  
ASREQ R3, R3, #1    ; Arithmetic shift right if equal  
ADDNE R3, R6, R7    ; Add if not equal  
ASRNE R3, R3, #1    ; Arithmetic shift right if not equal
```

## 4.3 Conditional Execution: IT Block\_3

**Table A.2** Condition Code Suffixes

Suffix	Flags	Meaning
EQ	Z = 1	Equal
NE	Z = 0	Not equal
CS or HS	C = 1	Higher or same, unsigned $\geq$
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 $\leq$
GE	N = V	Greater than or equal, signed $\geq$
LT	N $\neq$ V	Less than, signed $<$
GT	Z = 0 and N = V	Greater than, signed $>$
LE	Z = 1 or N $\neq$ V	Less than or equal, signed $\leq$
AL	Can have any value	Always; default when no suffix is specified

### IT Example 2: Compare and Update Value

```
CMP      R0, R1  @ Compare R0 and R1, setting flags
ITT      GT      @ IT - Skip next two instructions unless GT condition holds
CMPGT    R2, R3  @ If 'greater than', compare R2 and R3, setting flags
MOVGT    R4, R5  @ If still 'greater than', do R4 = R5
```



## 4.4 Conditional Execution: Program E.g.

- An asm program for adding a list of  $N$  numbers (at location labelled  $N$ ) stored in consecutive memory locations, starting from a location labelled  $NUM1$ :

	LDR	R1, N	Load count into R1.
	LDR	R2, =NUM1	Load address NUM1 into R2.
	MOV	R0, #0	Clear accumulator R0.
LOOP	LDR	R3, [R2], #4	Load next number into R3.
	ADD	R0, R3	Add number into R0.
	SUBS	R1, #1	Decrement loop counter R1.
	BGT	LOOP	Branch back if not done.
	LDR	R4, =SUM	Store sum.
	STR	R0, [R4]	
	:	:	

GT    Z = 0 and N = V    Greater than, signed >

PC-relative STR is not permitted in ARMv7E-M;  
it takes 2 instructions to complete the task.

# 5. ARMv7E-M Logic Instructions

## 5.1 Logic Instructions: AND, ORR, EOR

➤ **AND, ORR & EOR** (*bit-wise* logic AND, OR & Exclusive-OR)

➤ Assembly language format:

**op{S} {Rd,} Rn, Op2**

where op is one of the above

Recall: *Operand2*

ADD R0, R1, #0xFF
ADD R0, R1, R2
ADD R0, R1, R2, LSL #0x4

➤ Example:

**ANDS Rd, Rn, Rm**

performs the bit-wise logic AND of the operands in registers Rn & Rm, writes the result into register Rd. N & Z are updated accordingly, C may be updated based on the result of Op2 (e.g. LSL #0x4, see Shift), V is not updated in all logic instructions



## 5.2 Logic Instructions: MVN (NOT)

➤ **MVN** (Move NOT: *bit-wise* logic NOT)

➤ Assembly language format:

**MVN{S}     *Rd, Op2***

performs a bit-wise logic NOT operation on the value of *Op2*, & places the result into *Rd*.

$Rd \leftarrow (\sim Op2)$

➤ Example:

**MVNS     *Rd, Rn***

performs the bit-wise logic NOT on *Rn*, writes the result into register *Rd*. N & Z are updated accordingly, C may be updated based on the result of *Op2* (e.g. LSL #0x4), V is not updated in all logic instructions

## 5.3 Shift & Rotate Instructions\_1

➤	<b>LSL</b>	Logical Shift Left
	<b>LSR</b>	Logical Shift Right
	<b>ASR</b>	Arithmetic Shift Right
	<b>ROR</b>	Rotate Right
	<b>RRX</b>	Rotate Right with Extend

The range of  $n$  (shift length) for the various instructions:  
LSL: 0 to 31  
LSR, ASR: 1 to 32  
ROR: 1 to 31

### ➤ Assembly language format:

<b>op{S}</b>	<b><i>Rd, Rm, Rs</i></b>	} where $op$ is one of the top 4, $Rs$ & $n$ (shift length) limited to $\leq 31/32$ , &
<b>op{S}</b>	<b><i>Rd, Rm, #n</i></b>	
<b>RRX{S}</b>	<b><i>Rd, Rm</i></b>	

- $Rm$ : value to be shifted/rotated, which **remains unchanged** after operation
- $Rs$  or  $\#n$ : holds the shift length, in **number of bits**
- The **first/last bit** shifted/rotated out is written into the **C** flag if the **S** suffix is specified. N & Z are updated accordingly
- **RRX always rotates by 1 bit.** Update the N, Z & C flags if the **S** suffix is specified

# 5.3 Shift & Rotate Instructions\_2

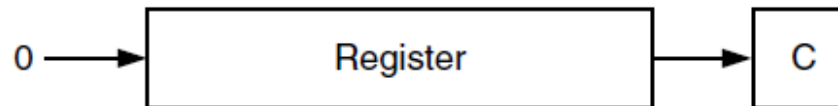
## Instructions:

Logical Shift Left (LSL)

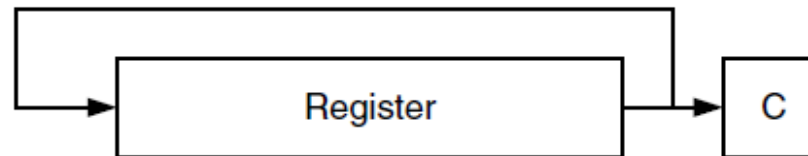
Carry flag



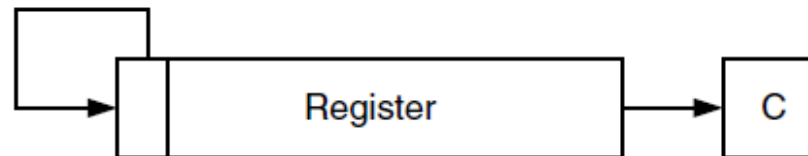
Logical Shift Right (LSR)



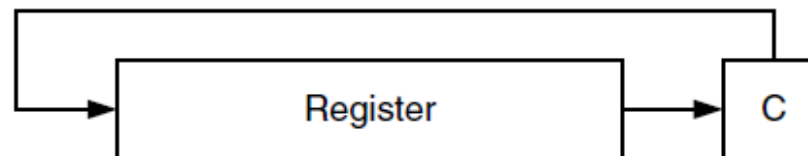
Rotate Right (ROR)



Arithmetic Shift Right (ASR)



Rotate Right eXtended (RRX)



## Effects:

Multiply by  $2^n$

Unsigned division by  $2^n$

32-bit rotate

Signed division by  $2^n$

33-bit rotate  
(33<sup>rd</sup> bit is carry flag)

**Note:** The **S suffix** should be specified in order to update the Carry flag, e.g. **LSLS R0, R1, #2**

#0xFF

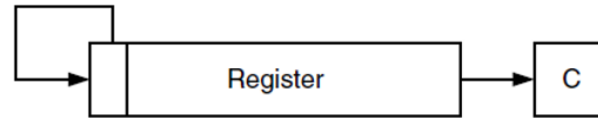
R2

R2, LSL #0x4

op{S} Rd, Rm, Rs  
 op{S} Rd, Rm, #n  
 RRX{S} Rd, Rm

## 5.3 Shift & Rotate Instructions: E.g.

Arithmetic Shift Right (ASR)

Signed division by  $2^n$ 

- Inline Barrel Shifter Example, recall Op2:

MOVS R0, R1, **ASR #2** @  $R0 \leftarrow R1 \gg 2$  (i.e. Op2)

If  $R1 = 0xA0000014$  ( $1010\dots\dots01\ 0100$ )<sub>2</sub>

$R1 \gg 2 = 0xE8000005$  ( $1110\ 10\dots\dots00\ 0101$ )<sub>2</sub>

which results in 0xE8000005 being written to R0

- The most significant bit (**sign** bit) of R1 is being copied in every shift, i.e. for **n** times
- Carry flag is finally cleared, due to the **0** in the original **n**<sup>th</sup> position (**n=2** in this ASR) of R1 before the shift

## 5.4 Test Instructions

- **TST & TEQ** (Test & Test Equivalence)
- Assembly language format:  
**TST**    *Rn, Op2*  
**TEQ**    *Rn, Op2*
- TST performs: **bit-wise** logic AND of the two operands
- TEQ performs: **bit-wise** logic Exclusive OR of the two operands
- Both update N & Z accordingly; C may be updated based on the result of Op2 (e.g. LSL #0x4); V is not affected
- TST & TEQ are similar to ANDS & EORS respectively, but they **discard** results

## 5.4 Test Instructions: Examples

### ➤ Checking a specific bit

TST R3, #1 @ bit-wise logic AND

sets Z = 1 if the least significant bit of R3 is 0

sets Z = 0 if the least significant bit of R3 is 1

(useful for checking status bits in I/O devices)

### ➤ Checking for a specific bit pattern

TEQ R2, #5 @ bit-wise logic Exclusive OR

sets Z = 1 if R2 equals 5

sets Z = 0 otherwise

(useful for testing the flags in peripheral control registers)

# 6. Stack & Subroutines/Functions

➤ **Stack:** an efficient **memory** usage model where data is saved/recalled in a Last-In-First-Out manner & address specified by **SP**

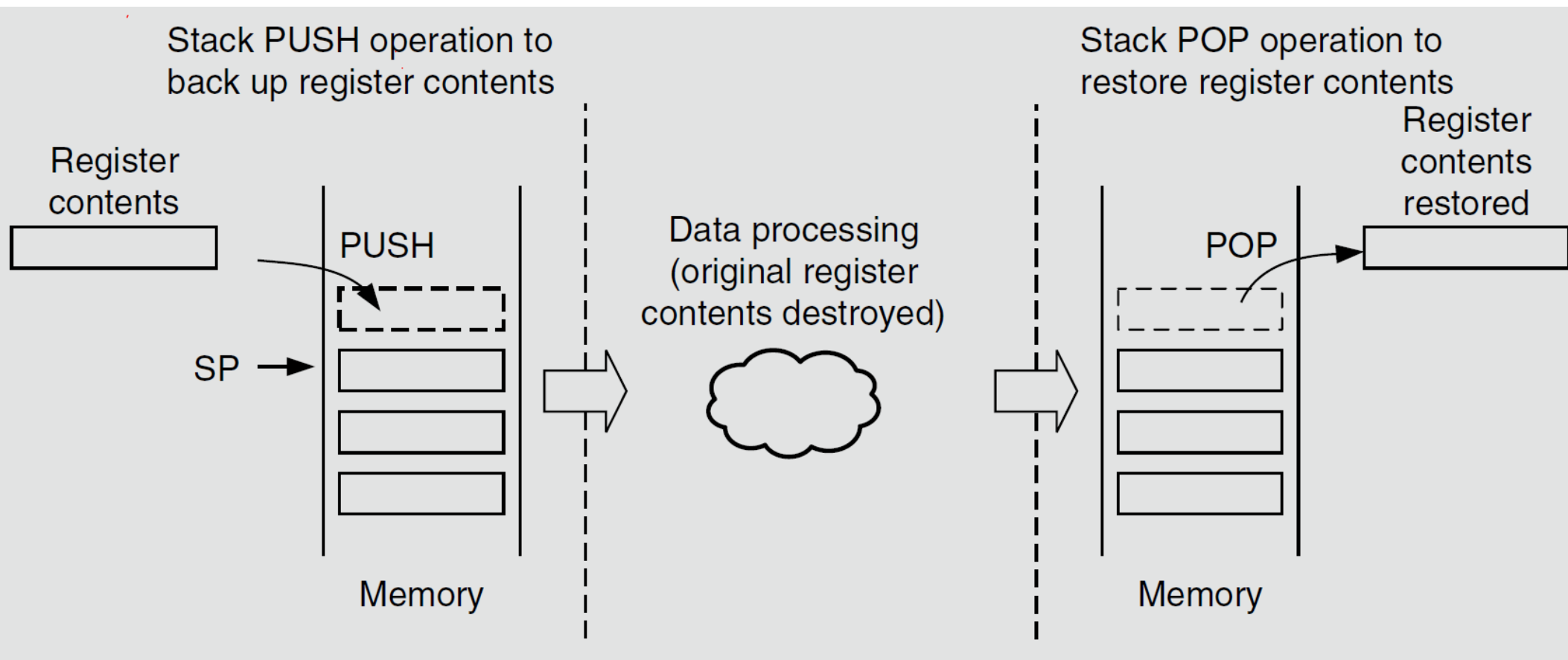
➤ **Stack Types:**

ARM supports **4** different stack implementations, categorised by two axes, namely Empty vs Full & Ascending vs Descending:

- **Empty** stack: SP points to the **next free** location on the stack, i.e. the location where the next item to be pushed onto the stack will be stored.
- In a **Full** stack, the stack pointer points to the **most recent item** in the stack, i.e. the location of the last item pushed onto the stack.
- An **Ascending** stack grows **upwards**: it **starts from a low memory** address &, as items are **pushed** onto it, progresses to higher memory addresses.
- A **Descending** stack grows **downwards**: it **starts from a high memory** address, & as items are **pushed** onto it, progresses to lower memory addresses.

# 6. Stack: Cortex-M4

- **Cortex-M4 Stack:** can be **software-controlled** or carried out **automatically** when enter/exit an exception/interrupt handler; **Full-Descending** stack
- **Common use:** to save Register contents before some data processing & then restore those contents from the stack after the processing task is done





## 6.1 Stack: Instructions

- Assembly language format:

**PUSH *reglist*** (Push registers onto stack)

**POP *reglist*** (Pop registers off stack)

They are **not**  
**OPTIONALS !**

where *reglist* is a non-empty list of register(s), enclosed in **braces**.

It can contain a register range, e.g. {R0-R7}

or more than one register or register range, which  
must be **comma separated**, e.g. {R0, R1-R3, R5-R7}

- **SP (R13)** is auto-decremented/incremented respectively

- E.g.: **PUSH {R0}**

performs

$R13 \leftarrow R13 - 4$

followed by

$Memory[R13] \leftarrow R0$

**POP {R0}**

performs

$R0 \leftarrow Memory[R13]$

followed by

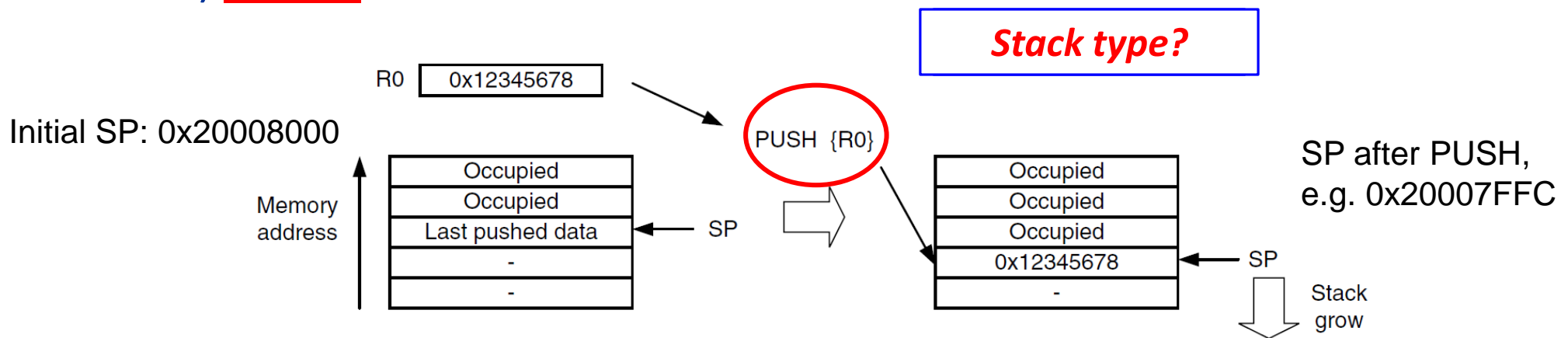
$R13 \leftarrow R13 + 4$

## 6.2 Stack: Practical Considerations

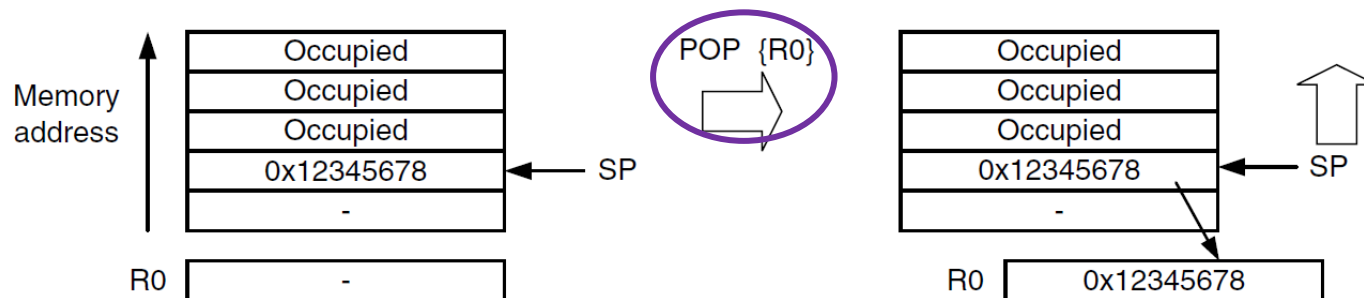
- **Avoid unnecessary PUSH & POP** – they are relatively expensive operations in terms of CPU cycles
- For the stack to work as expected, always perform PUSH & POP **in pairs**, and preferably with the same *reglist*
- **R7 &/or R11** are often used as the default Frame Pointers (which point to the current stack frame of a function) – avoid using them to prevent overwriting these return addresses accidentally
- The **highest numbered register** will be **PUSHed first & POPed last**, & vice versa, irrespective of the order of registers in the *reglist*

## 6.3 Stack: Operation Details

- E.g. initial SP set to point at: 0x20008000 (always empty)
  - SP (R13) always points to the last data pushed into stack memory
  - When **PUSH**, SP **decrements** by 4 or multiples of 4 (if several registers are saved) **before** new data is inserted



- When **POP**, SP **increments** by 4 or multiples of 4 (if several registers are recalled) **after** data is copied (*What happen to stack's contents after POP?*)



## 6.4 Subroutines/Functions\_Method 1

- Subroutines (*software-controlled stack operation*):  
Manually PUSH **registers** that will be modified in the subroutine, then POP them at the end of the subroutine to restore their original contents
- Multiple-registers **PUSH** & **POP** are similar to multiple-word **STR** & **LDR** respectively (*refer to STM & LDM for details, not in syllabus*)

Main program

```
...  
; R0 = X, R1 = Y, R2 = Z  
BL    function 1
```

Subroutine/Function

function 1

```
PUSH    {R0-R2} ; Store R0, R1, R2 to stack  
... ; Executing task (R0, R1 and R2  
      ; could be changed)  
POP     {R0-R2} ; restore R0, R1, R2  
BX      LR      ; Return
```

```
; Back to main program  
; R0 = X, R1 = Y, R2 = Z  
... ; next instructions
```

## 6.4 Subroutines/Functions\_Method 2

### ➤ Subroutines (*software-controlled stack operation*):

Manually PUSH **registers** that will be modified & **LR** in the subroutine, then POP registers at the end to restore their original contents & **LR into PC**, thus combining POP & BX LR into one instruction.

Main program

```
...  
; R0 = X, R1 = Y, R2 = Z  
BL    function 1
```

Subroutine/Function

function 1

```
PUSH    {R0-R2, LR} ; Save registers  
                ; including link register
```

```
... ; Executing task (R0, R1 and R2  
    ; could be changed)
```

```
POP      {R0-R2, PC} ; Restore registers and  
                ; return
```

```
; Back to main program  
; R0 = X, R1 = Y, R2 = Z  
... ; next instructions
```

# Armv7E-M Instructions Summary\_1

CG2028 Armv7E-M Instruction Summary ©Dr Henry Tan

## Glossary:

1. LABEL
2. {optional}
3. Op2
4. #immediate
5. Pre- & Suffix

S-suffix updates **flags** found in Special Register APSR:

N Negative  
Z Zero  
C Carry  
V Overflow

## Data Declaration:

Constant, by substitution  
Constant, by memory allocation  
Static Variable

.equ symbol, expression  
LABEL: .word constant-value(s) or pointer  
.lcomm symbol no.-of-bytes-reserved

## Load/Store:

Basic Offset  
  
Pre-index Offset  
Post-index Offset  
PC-relative  
Pseudo-instruction

LDR/STR Rt, [Rn]  
LDR/STR Rt, [Rn, #offset]  
LDR/STR Rt, [Rn, #offset]!  
LDR/STR Rt, [Rn], #offset  
LDR Rd, LABEL  
LDR Rd, =value or =LABEL

## Move:

8-bit data, Register or Inline Barrel Shifter  
16-bit data

MOV{S} Rd, Op2  
MOV{S} Rd, #imm16

## Arithmetic:

Add/Subtract, 8-bit data, Reg or Inline  
Add/Subtract, 16-bit data  
32-bit Multiplication  
32-bit Mul with Accumulate  
32-bit Mul with Subtract  
64-bit Unsigned Multiplication  
64-bit Unsigned Mul with Accumulate  
64-bit Signed Multiplication  
64-bit Signed Mul with Accumulate  
32-bit Unsigned Division  
32-bit Signed Division  
Compare (similar to SUBS)  
Compare Negative (similar to ADDS)

ADD{S}/SUB{S} {Rd,} Rn, Op2  
ADD{S}/SUB{S} {Rd,} Rn, #imm12  
MUL{S} {Rd,} Rn, Rm  
MLA{S} Rd, Rn, Rm, Ra  
MLS{S} Rd, Rn, Rm, Ra  
UMULL{S} {Rd,} Rn, Rm  
UMLAL{S} Rd, Rn, Rm, Ra  
SMULL{S} {Rd,} Rn, Rm  
SMLAL{S} Rd, Rn, Rm, Ra  
UDIV {Rd,} Rn, Rm  
SDIV {Rd,} Rn, Rm  
CMP Rn, Op2  
CMN Rn, Op2

# Armv7E-M Instructions Summary\_2

## Branch:

Branch (immediate)  
Branch with Link (immediate)  
Branch Indirect with Link (Register)  
Branch Indirect (Register)  
If-Then (IT) Block

B{cond} LABEL  
BL{cond} LABEL  
BLX{cond} Rm  
BX{cond} Rm  
IT<x><y><z> <cond>  
instr1<cond>  
instr2<cond or ~(cond)>  
instr3<cond or ~(cond)>  
instr4<cond or ~(cond)>

## Logic (i.e. bit-wise):

And  
Or  
Xor  
Not (aka Move Not)  
Logical Shift Left  
Logical Shift Right  
Arithmetic Shift Right  
Rotate Right  
Rotate Right with Extend  
Inline Barrel Shifter: an option for *Op2*  
Test (similar to ANDS)  
Test Equivalence (similar to EORS)

AND{S} {Rd,} Rn, Op2  
ORR{S} {Rd,} Rn, Op2  
EOR{S} {Rd,} Rn, Op2  
MVN{S} Rd, Op2  
LSL{S} Rd, Rm, Rs or #n  
LSR{S} Rd, Rm, Rs or #n  
ASR{S} Rd, Rm, Rs or #n  
ROR{S} Rd, Rm, Rs or #n  
RRX{S} Rd, Rm  
e.g. MOV{S} Rd, Rm, ASR #n TST  
Rn, Op2  
TEQ Rn,

## Stack:

Onto Stack  
Off Stack

PUSH reglist  
POP reglist

Table A.2 Condition Code Suffixes

Suffix	Flags	Meaning
EQ	Z = 1	Equal
NE	Z = 0	Not equal
CS or HS	C = 1	Higher or same, unsigned $\geq$
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 $\leq$
GE	N = V	Greater than or equal, signed $\geq$
LT	N != V	Less than, signed $<$
GT	Z = 0 and N = V	Greater than, signed $>$
LE	Z = 1 or N != V	Less than or equal, signed $\leq$
AL	Can have any value	Always; default when no suffix is specified