

CG2028
Computer Organization

Lecture 3: ARM Instruction Set & Assembly Language

Dr Henry Tan, ECE, NUS
E-mail: eletanh@nus.edu.sg



```
CMP R0, R1
ITTEE EQ
ADDEQ R3, R4, R5
ASREQ R3, R3, #1
ADDNE R3, R6, R7
ASRNE R3, R3, #1
```

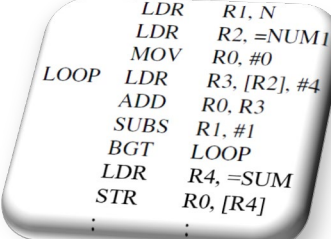
ARM Instruction Set & Asm

➤ Objectives:

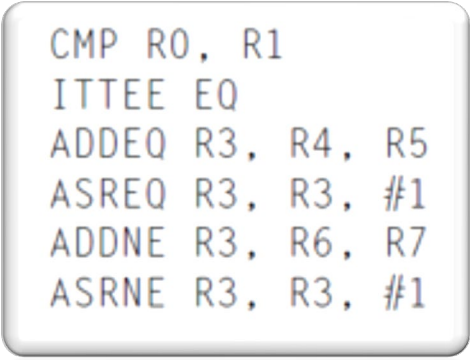
- Understand the characteristics of the ARMv7-M ISA, memory addressing & asm instructions

➤ Outline:

- 1. Introduction to ARMv7-M Assembly Language
 - 1.1 Why asm?
 - 1.2 Calling asm from C Program
 - 1.3 ARMv7-M Glossary: label, option(al)s, Op2, #Imm, Pre- & Suffix
- 2. Memory Addressing
 - 2.1 Memory Allocation for Data
 - 2.2 Offset Addressing
 - 2.3 Offset Addressing – with Pre/Post Index
 - 2.4 PC-Relative Addressing
 - 2.5 Pseudo-Instruction Addressing
- 3. ARMv7-M Ctrl & Arithmetic Instructions
 - 3.1 Move
 - 3.2 Add & Subtract
 - 3.3 Multiply & Multiply with Accumulate
 - 3.4 Compare
 - 3.5 Branch
- 4. Conditional Execution & Condition Code Suffixes
 - 4.1 Conditional Branch
 - 4.2 IT Block
- 5. ARMv7-M Logic Instructions
 - 5.1 And, Or, Xor
 - 5.2 Not
 - 5.3 Shift & Rotate
 - 5.4 Test
- 6. Stack & Subroutines/Functions



```
LDR R1, N
LDR R2, =NUM1
MOV R0, #0
LOOP LDR R3, [R2], #4
      ADD R0, R3
      SUBS R1, #1
      BGT LOOP
      LDR R4, =SUM
      STR R0, [R4]
      ;
      ;
```



```
CMP R0, R1
ITTEE EQ
ADDEQ R3, R4, R5
ASREQ R3, R3, #1
ADDNE R3, R6, R7
ASRNE R3, R3, #1
```

Eg 7: Memory Addressing Comparison

➤ 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	0x100
	0x0000010C	LDR R4, [R3], #4		R2	LDR R1, NUM1
	0x00000110	LDR R5, [R1, #4]		R3	0x00000804
	0x00000114	LDR R6, [R4, #4]!		R4	0x104
	0x00000118	STR R7, [R3]		R5	LDR R2, [R1]
	:	:		R6	LDR R2, [R1]
NUM1:	0x00000800	0x100		R7	0x7
	0x00000804	0x7		:	

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

Eg 8a

.EQU C, xxx	@	.equ is always located near the top of the code, so we can conveniently key in the last-minute entry
LDR R1, A	@	load R1 with 'value' A, but we know it's actually A's location resolved with the "imaginary <i>invisible square brackets</i> " – remember, once you know assembly, you'll associate a label with its address more than its value.
LDR R2, B	@	another PC-Relative load as above
LDR R3, =C	@	we want the assembler to substitute C's value & we want it literally 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, ARMv7-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 literally 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	
LDR R1, =A	@	Pseudo-Instruction load R1 with 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 literally 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 literally want the address, use Pseudo-Instruction, with the equal sign)	
STR R0, [R5]	@	then load the final result in R0 to the memory location pointed by R5, which holds the address of ANS	
.LCOMM ANS 4			

3. ARMv7-M Ctrl & Arithmetic Instructions

3.1 Move Instructions

➤ MOV & MOVW

➤ Assembly language format:

MOV *Rd, Op2* →
MOVW, MOV *Rd, #imm16*

e.g.

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

Operand2

➤ Examples:

MOV Rd, Rm
performs
 $Rd \leftarrow Rm$

MOV Rd, #value
MOVW Rd, #value
performs
 $Rd \leftarrow \text{value}$

or
(restricted to using 16-bit constants)

LDR or STR vs MOV:

LDR or STR is for transfers to/from a register from/to memory (hence the []), 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 constant in a register (e.g. R1)	MOV R1, #7 or LDR R1, =7	LDR R1, CONST CONST: .word 7	LDR R1, #7 
2. There is a constant in the memory (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 result in another register (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 result in the memory (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 or STR ANS, R2 
5. I need to set up a pointer (e.g. with R3)	LDR R3, =NUM1	LDR R0, =NUM1 MOV R3, R0	MOV R3, =NUM1 or STR =NUM1, R3 

3.2 Arithmetic Instructions: ADD, SUB

➤ **ADD & SUB** (Add & Subtract)

➤ Assembly language format:

ADD/SUB {*Rd*,} *Rn*, *Op2*

or

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

e.g.

ADD R0, R1,

Operand2

#0xFF

ADD R0, R1,

R2

ADD R0, R1,

R2, LSL #0x4

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

➤ Examples:

ADD R0, R2, R4

SUB R0, R3, #6

performs

performs

$R0 \leftarrow R2 + R4$

$R0 \leftarrow R3 - 6$

3.3 Arithmetic Instructions: MUL, MLA NUS

National University
of Singapore

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

➤ Assembly language format:

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

MLA Rd, Rn, Rm, Ra

Example:

MUL R0, R1, R2	<i>while</i>	MLA R0, R4, R5, R6
performs		performs
$R0 \leftarrow R1 \times R2$		$R0 \leftarrow (R4 \times R5) + R6$

- **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

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



- To get **64-bit products**, use the **long** versions that come with suffix **L**, in either unsigned (**U**) or signed (**S**) variants, e.g. **UMULL**, **UMLAL**, **SMULL**, & **SMLAL**
- Fyi, there is a counterpart to MLA: Multiply with Subtract (**MLS**) instruction
- **Division** can either be SDIV or UDIV (Signed/Unsigned Divide)
- Assembly language format:
SDIV {Rd,} Rn, Rm or **UDIV {Rd,} Rn, Rm** **@ Rd ← 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.

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

Program Example 1 - Sum of Products_1

```
/*
 * LPC1769_asm_basic : asm.s
 *
 * Note: this is a stand-alone assembly language program written for LPC1769
 *
 * Simple assembly language program to compute
 * ANSWER = A*B + C*D
 */
```

@ Directives

```
.thumb @ (same as saying '.code 16')
.cpu cortex-m3
.syntax unified
.align 2
```

← Directives to the GNU Assembler

@ Equates

```
.equ STACKINIT, 0x10008000
```

← Set the address of the top of stack

@ Vectors

vectors:

```
.word STACKINIT @ stack pointer value when stack is empty
.word _start + 1 @ reset vector (manually adjust to odd for thumb)
.word _nmi_handler + 1 @
.word _hard_fault + 1 @
.word _memory_fault + 1 @
.word _bus_fault + 1 @
.word _usage_fault + 1 @
.word 0 @ checksum
```

← Define table for exception handlers

```
.global _start
```

Program Example 1 - Sum of Products_2

- Main code executes once, but upon completion, it loops endlessly for inspection of registers & memory

```
@ Start of executable code
.section .text

_start:

@ code starts
@ Calculate ANSWER = A*B + C*D
    LDR R0, A @ PC-relative load
    LDR R1, B
    MUL R0, R1, R0
    LDR R1, C
    LDR R2, D
    MLA R0, R1, R2, R0
    LDR R3, =ANSWER
    STR R0, [R3]

@ Loop at the end to allow inspection of registers and memory
loop:
    b loop
```

← Indicates program section

← Main assembler program

← Loads R3 with the address of ANSWER

← More on Branch instructions later

Program Example 1 - Sum of Products_3

➤ Constants definition and memory reservation are done towards the end of the assembly program

```
@ Loop if any exception gets triggered
```

```
_exception:
_nmi_handler:
_hard_fault:
_memory_fault:
_bus_fault:
_usage_fault:
    b _exception
```

← Exception handling

```
@ Define constant values
```

```
A:
    .word 100
B:
    .word 50
C:
    .word 20
D:
    .word 400
```

← Declare variables/constants used

```
@ Store result in SRAM (4 bytes)
.lcomm    ANSWER    4
.end
```

← Reserve 1 word of memory to store the sum of products

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:

$Rn - Op2$

$Rn + Op2$

& **updates condition flags** N, Z, C & V 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: *for/while/do-while* loop; *if-else/switch-case*
BL or **BLX**: To jump to a Function from Main
BX: To go back to Main, from a Function

where:

- **L**: branch with **link**, i.e. writes the address of the next instruction to **LR**
- **X**: branch **indirect**, via register **Rm** that indicates an **address** to branch to
- *label*: a PC-relative expression indicating an **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 (and 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 unequal, 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 references

- **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:

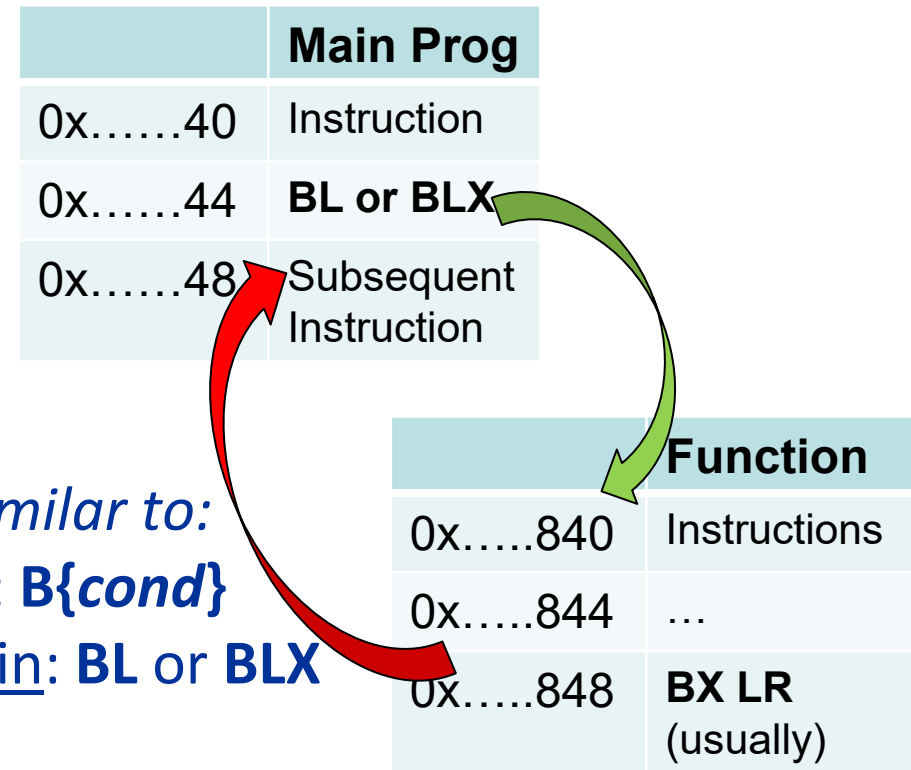
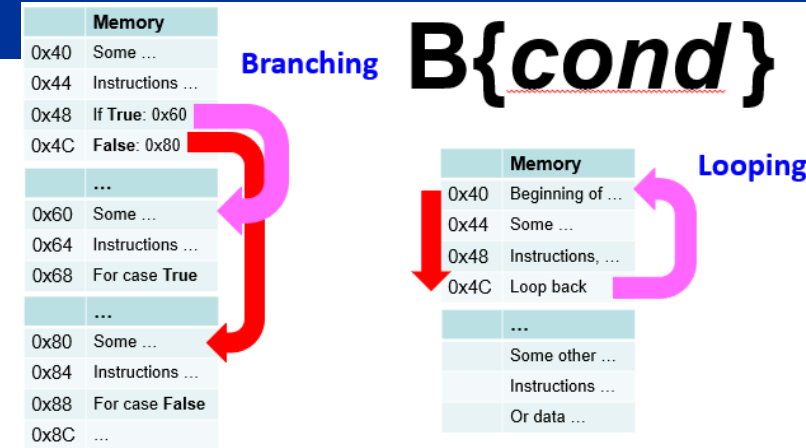
BX{cond} Rm

performs: branch to location indicated by Rm

$PC \leftarrow Rm$

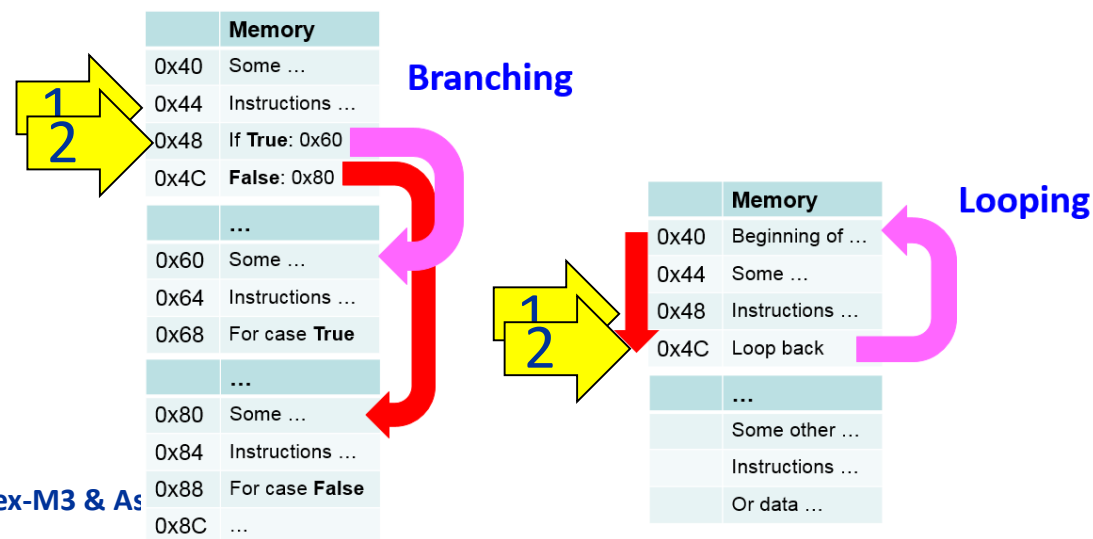
- Example: BX LR

- **When to use what?** Usually, in situations similar to:
 - ✓ For/While loop; If-Else/Switch-Case branch: **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



4. 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

4.1 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

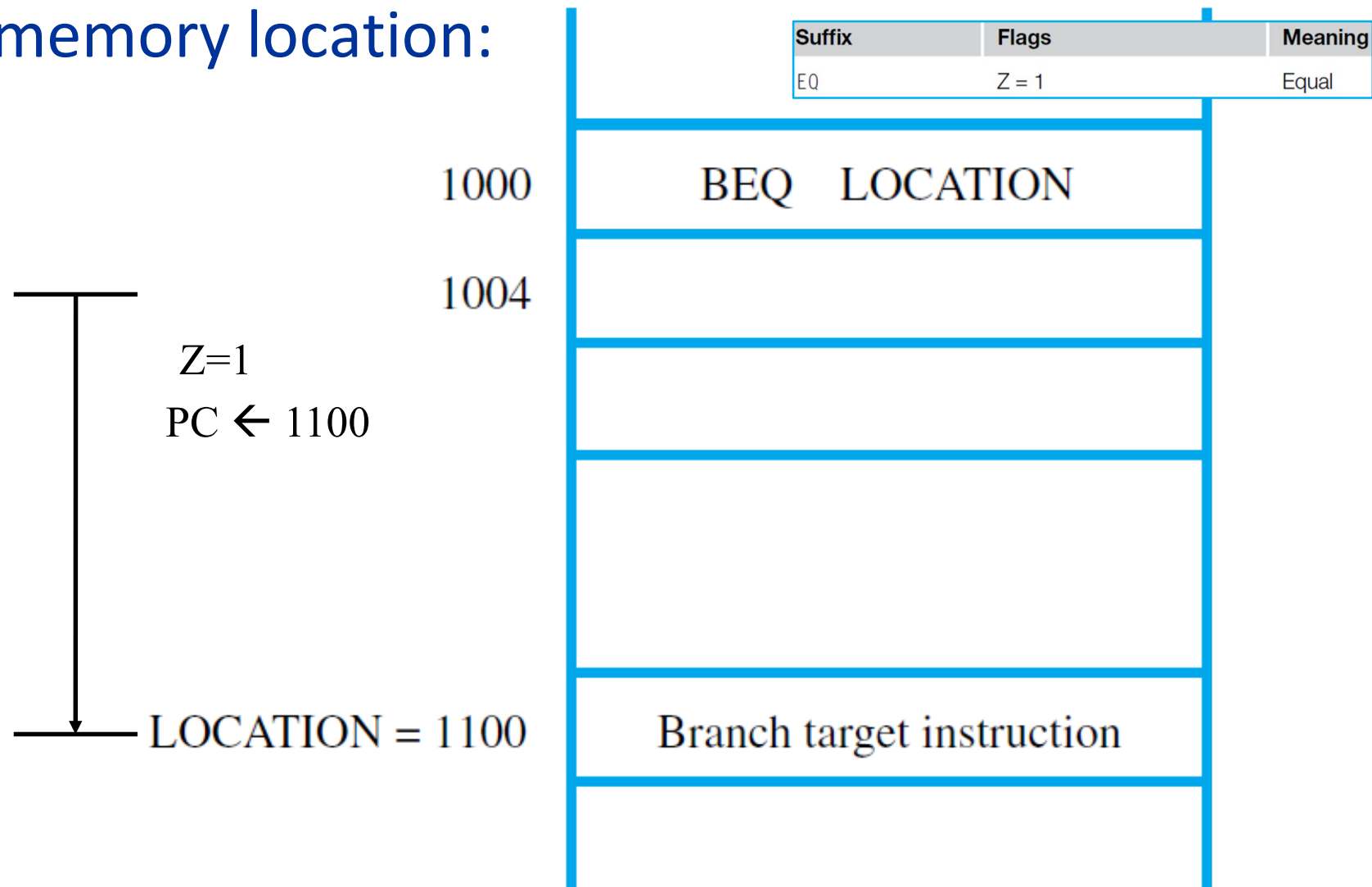
branches to label LOCATION if **Z=1**

Suffix	Flags	Meaning
EQ	Z = 1	Equal

➤ 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

4.1 Conditional Execution: Branch_2

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



4.2 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!

4.2 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
```


4.2 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
```

Program Example 2 - A Sum Loop

- An asm program for adding a list of N numbers (at location labelled N) stored in consecutive memory locations, starting from 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 ARMv7-M;
it takes 2 instructions to complete the task.

5. ARMv7-M Logic Instructions

5.1 Logic Instructions: AND, ORR, EOR

➤ **AND, ORR & EOR** (*bit-wise* logical 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* logical AND of the operands in registers Rn & Rm, writes the result into register Rd & **updates condition flags** N & Z; C may be updated due to the calculation of Op2 (e.g. LSL #0x4, see Shift)

5.2 Logic Instructions: MVN (NOT)

- **MVN** (Move NOT: *bit-wise* logical NOT)
- Assembly language format:

MVN{S} Rd, Op2

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

$Rd \leftarrow (\sim Op2)$

- Example:

MVNS Rd, Rn

performs the *bit-wise* logical NOT on *Rn*, writes the result into register *Rd* & **updates condition flags** N & Z; C may be updated due to the calculation of *Op2* (e.g. LSL #0x4)

5.3 Shift & Rotate Instructions_1

➤	LSL	Logical Shift Left
	LSR	Logical Shift Right
	ASR	Arithmetic Shift Right
	ROR	Rotate Right
	RRX	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:

$\left. \begin{array}{l} \text{op}\{S\} \quad Rd, Rm, Rs \\ \text{op}\{S\} \quad Rd, Rm, \#n \\ \text{RRX}\{S\} \quad Rd, Rm \end{array} \right\}$ where op is one of the top 4, Rs & n (shift length) limited to $\leq 31/32$, &

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

5.3 Shift & Rotate Instructions_2

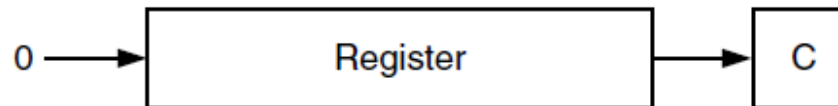
Instructions:

Logical Shift Left (LSL)

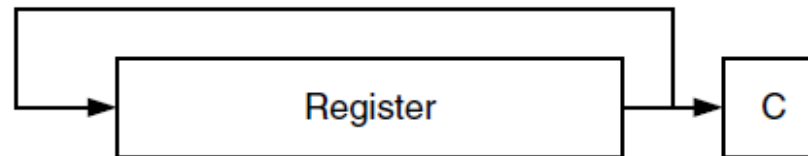
Carry flag



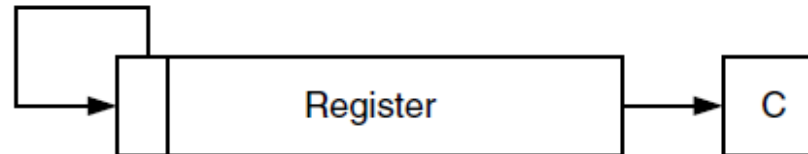
Logical Shift Right (LSR)



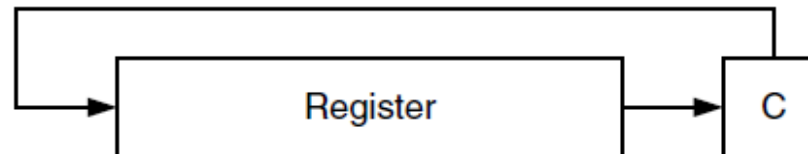
Rotate Right (ROR)



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
(33rd bit is carry flag)

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

5.3 Shift & Rotate Instructions: Example

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 $R1 = 0xA0000014$ ($1010\dots\dots01\ 0100$)₂

$R1 \gg 2 = 0xE8000005$ ($1110\ 10\dots\dots00\ 0101$)₂

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***th 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* logical AND of the two operands
- TEQ performs: *bit-wise* logical Exclusive OR of the two operands
- Both **update condition flags** N & Z; C may be updated due to the calculation of Op2 (e.g. LSL #0x4); V not affected
- TST & TEQ are similar to ANDS & EORS respectively, but they **discard** results

5.4 Test Instructions: Examples

➤ Examples:

TST R3, #1 @ *bit-wise* logical AND

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

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

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

TEQ R2, #5 @ *bit-wise* logical Exclusive OR

sets Z = 1 if R2 contains 5

sets Z = 0 otherwise

(also useful for testing the sign of a value; check N flag)

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**

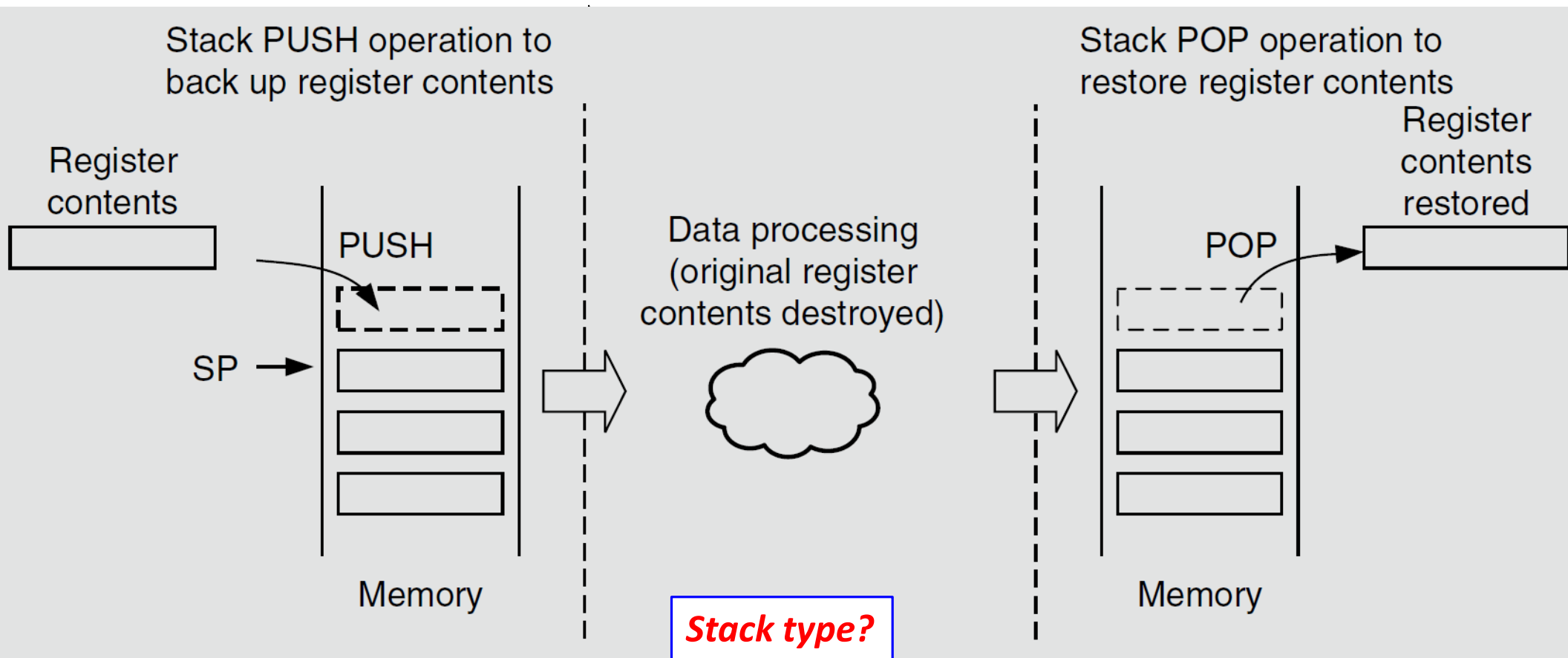
➤ **Stack Types:**

➤ 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-M3

- **Cortex-M3 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. 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 register ranges, e.g. {R0-R7}

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

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

- Eg: **PUSH {R0}**

performs

$R13 \leftarrow R13 - 4$

followed by

$\text{Memory}[R13] \leftarrow R0$

POP {R0}

performs

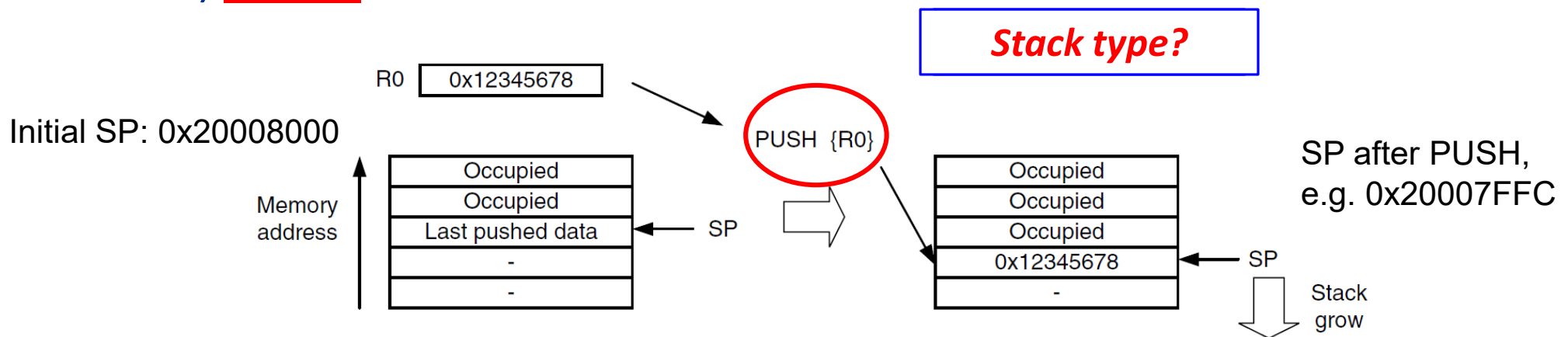
$R0 \leftarrow \text{Memory}[R13]$

followed by

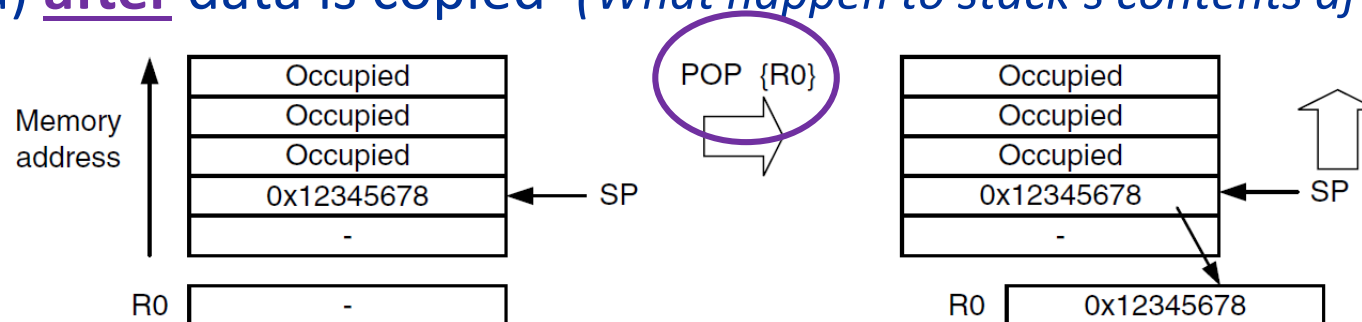
$R13 \leftarrow R13 + 4$

6. Stack: Operation Details

- LPC initial SP set to point at: 0x20008000 (always empty)
 - SP (R13) 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. 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 to restore their original contents
- Multiple-registers **PUSH** & **POP** are similar to multiple-word **STR** & **LDR** respectively (*see STM & LDM for details, not in syllabus*)

Main program

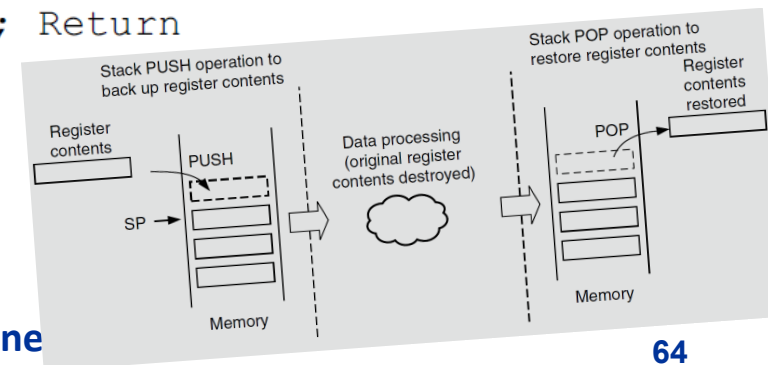
```
...  
; R0 = X, R1 = Y, R2 = Z  
BL    function 1
```

Subroutine

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. 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 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
```

6. Subroutines/Functions_3

➤ Exceptions and Interrupts (*automatic stack operation*):

Certain registers are automatically PUSHed before exception/interrupt handler runs & automatically POPped back at the end of its execution

Table 3.4 Exception Types in Cortex-M3

Exception Number	Exception Type	Priority	Function
1	Reset	−3 (Highest)	Reset
2	NMI	−2	Nonmaskable interrupt
3	Hard fault	−1	All classes of fault, when the corresponding fault handler cannot be activated because it is currently disabled or masked by exception masking
4	MemManage	Settable	Memory management fault; caused by MPU violation or invalid accesses (such as an instruction fetch from a nonexecutable region)
5	Bus fault	Settable	Error response received from the bus system; caused by an instruction prefetch abort or data access error
6	Usage fault	Settable	Usage fault; typical causes are invalid instructions or invalid state transition attempts (such as trying to switch to ARM state in the Cortex-M3)
7–10	—	—	Reserved
11	SVC	Settable	Supervisor call via SVC instruction
12	Debug monitor	Settable	Debug monitor
13	—	—	Reserved
14	PendSV	Settable	Pendable request for system service
15	SYSTICK	Settable	System tick timer
16–255	IRQ	Settable	IRQ input #0–239

Program Example 3 - Lab 1 'Blinky'

➤ Excerpt from Lab Manual:

- The programs turn on and off a digital output line in the Fast General Purpose IO (GPIO) **port 0 pin 22** that is connected to the red LED. However, the physical pin of the LPC1769 has several functions and the **GPIO functionality** has to be selected by writing a **'00' pattern in bits 12 and 13** of the **PINSEL1** register (at address 0x4002C004).
- After the GPIO functionality has been selected, we need to set the **direction of port 0 pin 22 to be output** by **writing** a bit pattern into the **FIOODIR** register (at address 0x2009C000).
- **Fyi**, the **state of GPIO port 0 pin 22**, i.e. whether it is on/high or off/low, can be determined by **reading** **bit 22 of the FIOOPIN** register (at address 0x2009C014). (**Note: this is not used in asm_blinky**)
- To **turn on** the red LED, make the state of GPIO port 0 pin 22 on/high by **writing '1' to bit 22 of the FIOOSET** register (at address 0x2009C018).
- To **turn off** the red LED, make the state of GPIO port 0 pin 22 off/low by **writing '1' to bit 22 of the FIOOCLR** register (at address 0x2009C01C). (**Note: you do not turn off the red LED by writing '0' to a bit in a register!**)
- Finally, there is a **delay loop** to make the red LED stay on or off for a short duration before switching to the other state. The whole cycle runs repeatedly.

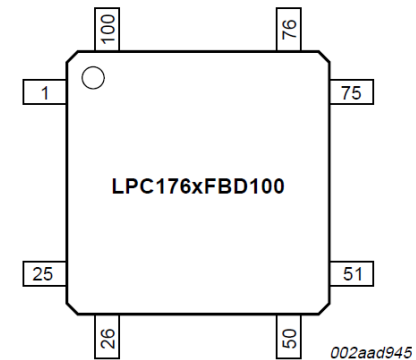
Program Example 3 - Lab 1 'Blinky': PINSEL, GPIO

Pin Function Select Register 1 (PINSEL1 - 0x4002 C004)

The PINSEL1 register controls the functions of the upper half of Port 0. The direction control bit in the FIO0DIR register is effective only when the GPIO function is selected for a pin. For other functions the direction is controlled automatically.

Table 80. Pin function select register 1 (PINSEL1 - address 0x4002 C004) bit description

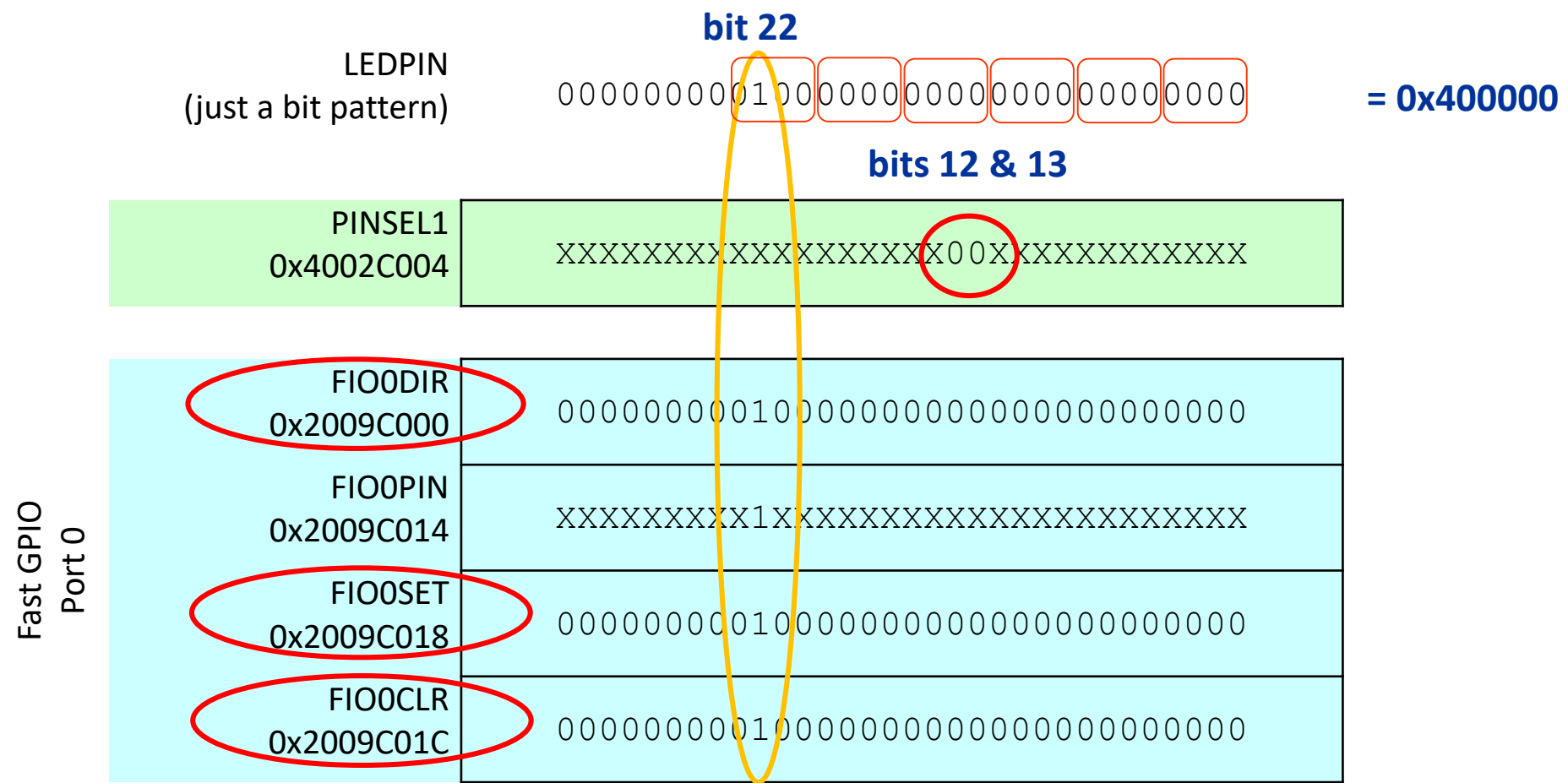
PINSEL1	Pin name	Function when 00	Function when 01	Function when 10	Function when 11	Reset value
1:0	P0.16	GPIO Port 0.16	RXD1	SSEL0	SSEL	00
3:2	P0.17	GPIO Port 0.17	CTS1	MISO0	MISO	00
5:4	P0.18	GPIO Port 0.18	DCD1	MOSI0	MOSI	00
7:6	P0.19 ^[1]	GPIO Port 0.19	DSR1	Reserved	SDA1	00
9:8	P0.20 ^[1]	GPIO Port 0.20	DTR1	Reserved	SCL1	00
11:10	P0.21 ^[1]	GPIO Port 0.21	RI1	Reserved	RD1	00
13:12	P0.22	GPIO Port 0.22	RTS1	Reserved	TD1	00
15:14	P0.23 ^[1]	GPIO Port 0.23	AD0.0	I2SRX_CLK	CAP3.0	00
17:16	P0.24 ^[1]	GPIO Port 0.24	AD0.1	I2SRX_WS	CAP3.1	00
19:18	P0.25	GPIO Port 0.25	AD0.2	I2SRX_SDA	TXD3	00
21:20	P0.26	GPIO Port 0.26	AD0.3	AOUT	RXD3	00
23:22	P0.27 ^{[1][2]}	GPIO Port 0.27	SDA0	USB_SDA	Reserved	00
25:24	P0.28 ^{[1][2]}	GPIO Port 0.28	SCL0	USB_SCL	Reserved	00



LQFP100 package

Program Example 3 - Lab 1 'Blinky': PINSEL, GPIO, LED

Registers related to 'Blinky' functionality:



Program Example 3 - Lab 1 'Blinky'_1

➤ LPC1769_asm_blinky - 1 (only main program is shown)

```

:
@ Equates
.equ STACKINIT,      0x10080000
.equ PINSEL1,        0x4002C004
.equ FIO0DIR,        0x2009C000
.equ FIO0SET,        0x2009C018
.equ FIO0CLR,        0x2009C01C
.equ LEDPIN,         0x400000
.equ LEDDELAY,       2000000

@ Start of executable code
.section .text

_start:
    @ set the pinselect
    ldr r6, =PINSEL1
    ldr r0, [r6]
    and r0, 0xffffcfff
    str r0, [r6]

    @ and port direction
    ldr r6, =FIO0DIR
    mov r0, LEDPIN
    str r0, [r6]

    @ Load R2 with the led pin number
    mov r2, LEDPIN
    @ value to turn on/off LED

```

Addresses of particular registers

Specifying particular bit patterns

Main program starts here

Writing '00' to bits 12 & 13 of PINSEL1 to select GPIO

Writing '1' to bit 22 of FIO0DIR to set it as output

Program Example 3 - Lab 1 'Blinky'_2

➤ LPC1769_asm_blinky - 2 (only main program is shown)

```
:
loop:    ldr r6, =FIO0SET
        str r2, [r6]           @ set Port 0, pin 22, turning on LED
        ldr r1, =LEDDELAY

delay1:  subs r1, 1
        bne delay1

        ldr r6, =FIO0CLR
        str r2, [r6]           @ clear Port 0, pin 22, turning off LED
        ldr r1, =LEDDELAY

delay2:  subs r1, 1
        bne delay2

        b loop                 @ continue forever
```

Note the use of suffix S in subs
& branching is immediately after

Main program ends

THE END
Questions?