

CG2028
Computer Organization

Lecture 2: ARM Instruction Set & Assembly Language

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



```
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]
      :      :
```

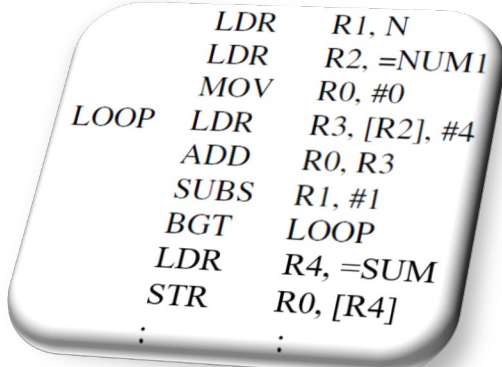
ARM Instruction Set & Asm

➤ 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]
      :
```

1. Introduction to ARMv7-M

1.1 Why learn Assembly Language?

- Assembly programs are **quicker, smaller & have larger capabilities** than those created with high-level languages.
- A **direct representation** of the actual machine language; through assembly you can have **total control** of the CPU.
- Assembler allows an **ideal optimization** in programs, be it their **size** or their **execution speed**.
- However, developing applications with assembly is **tedious & error-prone**.
- **Combination of C and asm** is a **powerful** method
→ our approach in this module!

1.2 Calling an Assembler Function from a C Program

- Call assembler function from C program (.c), e.g.
 - **extern int** *my_asm_func*(int x, int y);
 - It will be treated as just another subroutine by C program
 - Input parameters: R0, R1, R2, R3 (maximum of 4)
 - Output (return) parameter: R0
- Define assembler function in asm program (.s)
 - *my_asm_func*: ...
 - *my_asm_func* may use **BX LR** to return to the calling C program: Branch Indirect (via register): $PC \leftarrow LR$ (*will be covered later*)
- This method is used in Assignment
 - Refer to the assignment skeleton code

1.3 ARMv7-M Glossary: i. label

➤ Common instruction format

label: opcode operand1, operand2, .. @ Comments

Comments/Remarks are inserted after “@”

The **first operand** is usually the **destination** of an Arithmetic/Logic/Move operation's result

➤ label is optional

- But a convenient way to refer to a **memory address** holding:
- 1. an **instruction**
(as shown in the e.g. above) especially when doing:
 - branching & looping
 - subroutine calls
- 2. a **user variable** (declared using **.word** assembler directive)
 - e.g. A, B, C, SUM, NUM1, POINTER, etc. Recall we can “Load R2, A”!
 - e.g. RADIUS: **.word** 100, etc. @ variable RADIUS initialized to 100
- 3. a **user-defined constant** (declared using **.equ** or **.word**)
 - e.g. **.equ** PI, 314 @ constant PI whose value is 314

1.3 ARMv7-M Glossary: ii. option(al)s

Note

In [Table 612](#):

- angle brackets, $\langle \rangle$, enclose alternative forms of the operand
- braces, $\{ \}$, enclose optional operands
- the Operands column is not exhaustive
- $Op2$ is a flexible second operand that can be either a register or a constant
- most instructions can use an optional condition code suffix.

For more information on the instructions and operands, see the instruction descriptions.

Table 612. Cortex-M3 instructions

Mnemonic	Operands	Brief description	Flags	Page
ADC, ADCS	$\{Rd, \} Rn, Op2$	Add with Carry	N,Z,C,V	Section 34.2.5.1
ADD, ADDS	$\{Rd, \} Rn, Op2$	Add	N,Z,C,V	Section 34.2.5.1
ADD, ADDW	$\{Rd, \} Rn, \#imm12$	Add	N,Z,C,V	Section 34.2.5.1
ADR	$Rd, label$	Load PC-relative address	-	Section 34.2.4.1
AND, ANDS	$\{Rd, \} Rn, Op2$	Logical AND	N,Z,C	Section 34.2.5.2
ASR, ASRS	$Rd, Rm, \langle Rs \mid \#n \rangle$	Arithmetic Shift Right	N,Z,C	Section 34.2.5.3
B	$label$	Branch	-	Section 34.2.9.1

1.3 ARMv7-M Glossary: iii. Op2

Note

In [Table 612](#):

- angle brackets, <>, enclose alternative forms of the operand
- braces, {}, enclose optional operands
- the Operands column is not exhaustive
- **Op2** is a flexible second operand that can be either a register or a constant
- most instructions can use an optional condition code suffix.

For more information on the instructions and operands, see the instruction descriptions.

Table 612. Cortex-M3 instructions

Mnemonic	Operands	Brief description	Flags	Page
ADC, ADCS	{Rd,} Rn, Op2	Add with Carry	N,Z,C,V	Section 34.2.5.1
ADD, ADDS	{Rd,} Rn, Op2	Add	N,Z,C,V	Section 34.2.5.1
ADD, ADDW	{Rd,} Rn, #imm12	Add	N,Z,C,V	Section 34.2.5.1
ADR	Rd, label	Load PC-relative address	-	Section 34.2.4.1
AND, ANDS	{Rd,} Rn, Op2	Logical AND	N,Z,C	Section 34.2.5.2
ASR, ASRS	Rd, Rm, <Rs #n>	Arithmetic Shift Right	N,Z,C	Section 34.2.5.3
B	label	Branch	-	Section 34.2.9.1

1.3 ARMv7-M Glossary: iii. Op2

- *Operand2* – a flexible 2nd **source** operand
 - Available to some Arithmetic/Logic & Move instructions
 - 1. Constant or 2. Register (with or without shift)

1. Constant (aka Immediate)

- Specify a **constant** in the form: **#constant**
 where constant must be **<imm8m>**

Operand2

e.g.

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

2. Register, with optional shift

- Specify a **register** in the form: *Rm* {, <shift>}
 where:
 - *Rm* is the register holding the data for the second operand
 - <shift> is an **optional shift** applied to *Rm*, e.g. Logical Shift Left
 (*Shift will be discussed later in the Logic Instructions section*)

1.3 ARMv7-M Glossary: iv. #Immediate

➤ Immediate operands – explicitly hardcoded (#constants)

32-bit instruction word in IR :

opcode ...	registers ...	value of #immX (X-bit)
------------	---------------	-------------------------

➤ #imm8

- 2^8 Range: 0 to 255 or -128 to 127

➤ #imm8m (*more flexible than #imm8*)

- Ignore the details - treat this like #imm8 !

➤ #imm12

- 2^{12} Range: 0 to 4095 or -2048 to 2047

ADD, ADDW	{Rd,} Rn, #imm12	Add
SUB, SUBW	{Rd,} Rn, #imm12	Subtract

➤ #imm16

- 2^{16} Range: 0 to 65535 or -32768 to 32767

MOVT	Rd, #imm16	Move Top
MOVW, MOV	Rd, #imm16	Move 16-bit constant

1.3 ARMv7-M Glossary: v. Pre- & Suffix

An Arithmetic/Logic/Move instruction may have:

1. Prefix S- or U-

- **S**: perform **signed** operation of the instruction
- **U**: perform **unsigned** operation of the instruction
- e.g. for division: `SDIV {Rd,} Rn, Rm`
vs `UDIV {Rd,} Rn, Rm`

2. Suffix -S (*optional*)

- `op{S}`: updates condition code flags according to the result of Arithmetic/Logic/Move operation `op`
- e.g. `ADDS {Rd,} Rn, Op2`
`MOVS Rd, Op2`

1.3 ARMv7-M Glossary: v. Suffix

(Condition Code Flags)

- The Application Program Status Register (APSR) contains the following condition flags:
 - **N** : Set to 1 if the result of the operation was **negative**, else cleared to 0
 - **Z** : Set to 1 if the result of the operation was **zero**, else cleared to 0
 - **C** : Set to 1 if the operation resulted in a **carry**, else cleared to 0
 - **V** : Set to 1 if the operation caused **overflow**, else cleared to 0
- A **Carry** (for unsigned operations) occurs:
 - if the result of an **addition** is greater than or equal to 2^{32}
 - if the result of a **subtraction** is positive or zero
 - as the result of an **inline barrel shifter operation** in an **arithmetic/logic/move** instruction (*to be discussed later in the Logic Instructions section*)
- **Overflow** (for signed operations) occurs if the result of an add, subtract or compare is **greater than or equal** to 2^{31} , **or less than** -2^{31}
- **Homework**: read up on two's-complement representation of numbers

2. Memory Addressing

2.1 Memory Allocation for Data_1

➤ Data declarations, by using **assembler directive(s)** either at the *beginning* (**.equ**) or at the *end* (**.word**) of an asm program/function:

➤ 1. **Constants** (using **.equ** or **.word**)

e.g. `.equ STACK_TOP, 0x20008000`

`.equ PI, 314` @ .equ sets value of PI to 314

`NUM1: .word 123, 456` @ single/multiple values, e.g. array

`POINTER: .word NUM1+4` @ useful for accessing an array

- **.equ** symbol, expression – sets the value of symbol to expression; not unlike C preprocessor *#define* statement - it gets substituted in any subsequent code; e.g. `LDR R1, =PI` @ load R1 with 314

- **.word** allocates a word-sized amount of storage space in that memory location. It can also initialize that location with a given value (& its consecutive locations for multiple values); not unlike *unsigned int*; e.g. `LDR R1, NUM1` @ load R1 with 123 (via .word)

	Memory	
NUM1:	0x40	123
	0x44	456
	:	:
POINTER:	0x..	0x....44

2. Memory Addressing

2.1 Memory Allocation for Data_2

	Memory
ANSWER: 0x4C	RESERVED
0x50

➤ 2. Static Variables (using **.lcomm**)

e.g. `.lcomm ANSWER 4 @` reserves 4 bytes (1 word)

- A static variable **retains its value** even when the subroutine/function exits
- Its lifetime (or "extent") is the entire run of the program
- `.lcomm` reserves the specified **number of bytes** of memory location for **global variable** whose value is **not yet available** at the time of coding
- `.equ`, `.word` & `.lcomm` are all assembler directives, not ARM instructions – no need `#` for the constants specified

*Note: `PI`, `NUM1`, `POINTER` and `ANSWER` in the above examples are all **labels (hardcoded)***

2.2 Offset Addressing

- **LDR or STR, Offset addressing :**

LDR/STR $Rt, [Rn \{, \#offset\}]$

Square brackets are compulsory!
They indicate **memory access**.
 $Rn \{, \#offset\}$ is a pointer.

- LDR : loads register Rt with value from stated memory
- STR : stores value in register Rt to stated memory
- Rn (the **base register**) contains the address of the memory location containing data, but

i.e. the **value** in Rn is the **memory address** we're interested!.
- If an **offset** is present, its value is added to (or subtracted from) the address obtained from the base register Rn
- The **result** (aka the **effective address, EA**) is used as the address for the memory access, & its content transferred
- The value in base register Rn is **unaltered**

Eg 1: Offset Addressing – No Offset

LDR R3, [R2]

	Memory
0x.....40	0x....0123
0x.....44	0x....0456
0x.....48	0x.... ..40
0x.....4C
0X.....50

	Registers
R2	0x.....40
R3	0x....0123
R4	
...	
...	

(Base Register)

Note: In all these examples, data & addresses are all 32-bit values.

Whenever fewer than 32 bits are shown, please assume that the given value is left extended to 32 bits.

- No *Immediate* Offset is specified
- *Immediate* offset is **0x0** in this example
- $EA = R2 + 0x0 = 0x.....40$
- $R3 \leftarrow 0x....0123$
- $R2 = 0x.....40$ (unchanged)

Eg 2: Offset Addressing – With #offset

LDR R3, [R2, #0x4]

What other way can we write this instruction?

	Memory
0x.....40	0x....0123
0x.....44	0x....0456
0x.....48	0x.... ..40
0x.....4C
0X.....50

	Registers
R2	0x.....40
R3	0x....0456
R4	
...	
...	

(Base Register)

- **Immediate offset is 0x4** in this example
- $EA = R2 + 0x4 = 0x.....44$
- $R3 \leftarrow 0x....0456$
- $R2 = 0x.....40$ (unchanged)

2.3 Offset Addressing – with Pre/Post Index

- Assembly language format:

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

@ pre-indexed addressing

LDR/STR Rt , [Rn], #offset

@ post-indexed addressing

- Data is loaded into register Rt

- **Pre-indexed addressing**

- The offset value is added to (or subtracted from, if the offset is negative) the address obtained from the base register Rn
- The **result is used as the Effective Address (EA)** for the memory access
- Content of the base register Rn is **updated with this EA**

- **Post-indexed addressing**

- The address obtained from the base register Rn **is used as the EA**
- The offset value is added to (or subtracted from) the address in Rn
- Content of the base register Rn is **updated with this result (*not used as EA!*)**

They are extensions of Offset Addressing:
LDR/STR Rt , [Rn {, #offset}]

2.3 Offset Addressing: Pre-Indexed vs Post-Indexed

➤ That is:

Pre-Indexed Addressing *while*

LDR Rd, [Rn, #offset]!

performs

$Rd \leftarrow [Rn + \text{offset}]$

followed by

$Rn \leftarrow Rn + \text{offset}$

Post-Indexed Addressing

LDR Rd, [Rn], #offset

performs

$Rd \leftarrow [Rn]$

followed by

$Rn \leftarrow Rn + \text{offset}$

Recall: In Offset Addressing **LDR/STR Rt, [Rn {, #offset}]**,
the value in the base register **Rn** is **unaltered !**

Eg 3: Offset Addressing – Pre-Index_1

LDR R3, [R2, #4]!

	Memory
0x.....40	0x....0123
0x.....44	0x....0456
0x.....48	0x....0789
0x.....4C
0X.....50

	Registers
R2	0x.....40
R3	
R4	
...	
...	

(Base Register)

- **Immediate** offset is 4 in this example
- $EA = R2 + 4 = 0x.....44$

Eg 3: Offset Addressing – Pre-Index_2

LDR R3, [R2, #4]!

	Memory
0x.....40	0x....0123
0x.....44	0x....0456
0x.....48	0x....0789
0x.....4C
0X.....50

	Registers
R2	0x.....44
R3	0x....0456
R4	
...	
...	

(Base Register)

- *Immediate* offset is 4 in this example
- $EA = R2 + 4 = 0x.....44$
- $R3 \leftarrow 0x....0456$
- $R2 \leftarrow 0x.....44$

Content of the Base Register is offset
& the **result** is used as the EA!

Eg 4: Offset Addressing – Post-Index_1

LDR R3, [R2], #4

	Memory
0x.....40	0x....0123
0x.....44	0x....0456
0x.....48	0x....0789
0x.....4C
0X.....50

	Registers
R2	0x.....40
R3	
R4	
...	
...	

(Base Register)

- ***Immediate* offset is 4** in this example
- **EA = R2 = 0x.....40**

Eg 4: Offset Addressing – Post-Index_2

LDR R3, [R2], #4

	Memory
0x.....40	0x....0123
0x.....44	0x....0456
0x.....48	0x....0789
0x.....4C
0X.....50

	Registers
R2	0x.....44
R3	0x....0123
R4	
...	
...	

(Base Register)

- *Immediate* offset is 4 in this example
- EA = R2 = 0x.....40
- R3 \leftarrow 0x....0123
- R2 \leftarrow R2+4 = 0x.....44

Content of the **Base Register** is used as the EA!

2.4 PC-Relative Addressing (*Load only*)

- Most convenient way of memory addressing, \approx *Load* $R2, A$
- But **only for LDR**, **not available for STR** in ARMv7-M!
- The **base register** is *always* the PC
- LDR loads a register (e.g. Rd) with the value from a PC-relative memory address that is specified by a **label**
- Location of data is *always relative* to that of the instruction
- Assembly language format:

LDR Rd , ITEM

performs

$Rd \leftarrow [PC + \text{offset}]$

where $EA = PC + \text{offset}$ &

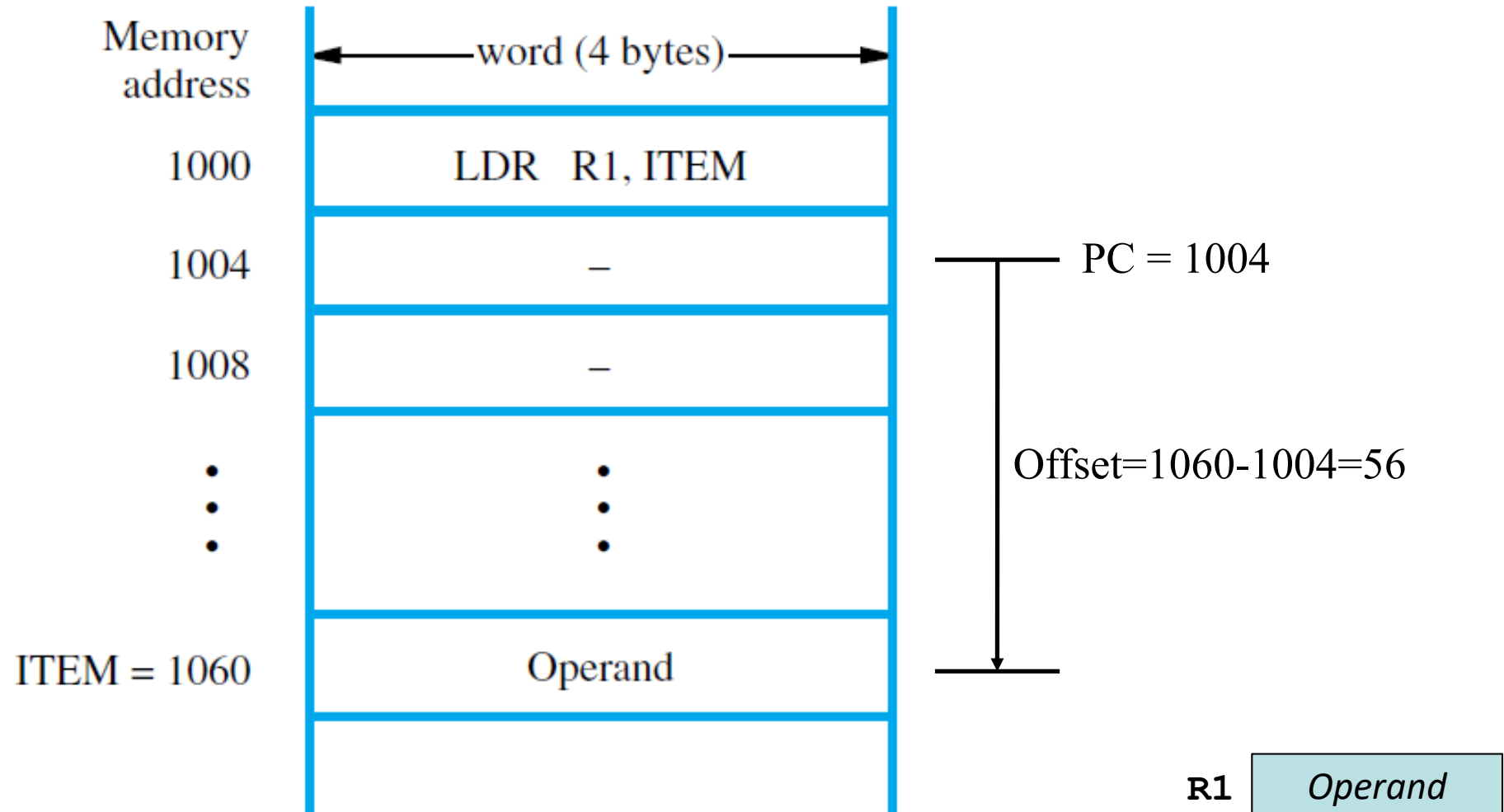
this **offset** is calculated by the assembler

ITEM here is a hardcoded **label** (*declared via .word*) which must be within a limited offset range of **-4095 to 4095** for successful PC-Relative Addressing! But this is the assembler's responsibility.

Eg 5: PC-Relative Addressing

LDR R1, ITEM

How is this instruction processed by the assembler?



Note: decimal memory addresses shown here for easier offset computation.

2.5 Pseudo-Instruction (*Load only*)

- Pseudo-instruction – **NOT** a real assembler instruction
- It does not have a direct machine language equivalent
- But the assembler will convert it into **PC-relative LDR** during assembly to produce the required data
- Useful for loading of ≤ 32 -bit **values/memory addresses** (*often*)
- E.g., loading a **32-bit** value:

LDR R1, =0xA123B456 (programmer: instruction)

is implemented with:

Remember the **equal sign** for pseudo-instruction!

LDR R1, MEMLOC (assembler: instruction)

MEMLOC: .word 0xA123B456 (assembler: data)

- i.e. the assembler allocates 1 word of memory, fills it with the stated value & lets the label (MEMLOC) refer to its address; &
- fulfills with an instruction similar to PC-relative addressing

Eg 6: Pseudo-Instruction Addressing

- E.g., loading an **address** (represented by a label):

LDR R3, =NUM1 (programmer: instruction)

is implemented by the combination of:

- An instruction similar to PC-relative addressing:

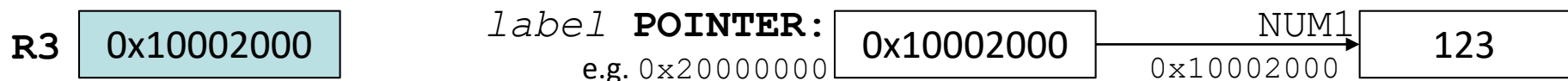
LDR R3, POINTER (assembler: instruction)

- & a data declaration:

POINTER: .word NUM1 (assembler: data)

Note: **NUM1: .word 123** (programmer: data)

- Effectively, *R3 contains the **address** of NUM1; a **pointer** to 123*



Eg 7: Memory Addressing Comparison

- Identify the addressing instructions & explain their differences. Update the register & memory contents after all the instructions have been executed.

	Main Memory			Registers	
	0x00000100	LDR R1, NUM1		PC	
	0x00000104	LDR R2, [R1]		:	
	0x00000108	LDR R3, =NUM1		R1	0x1
	0x0000010C	LDR R4, [R3], #4		R2	0x2
	0x00000110	LDR R5, [R1, #4]		R3	0x3
	0x00000114	LDR R6, [R4, #4]!		R4	0x4
	0x00000118	STR R7, [R3]		R5	0x5
	:	:		R6	0x6
NUM1:	0x00000800	0x100		R7	0x7
	0x00000804	0x104		:	

Eg 8: Memory Addressing Application

➤ a. Using the assembly instructions that you've just learnt, find the sum of $(A+5) + (B+10) + C$ & store the result to 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, all variables can be assumed to be .word-declared by default.]*

➤ b. What if the numbers 5 & 10 above are stored in 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:	0x40	A's value
	0x44	5
	0x48	10