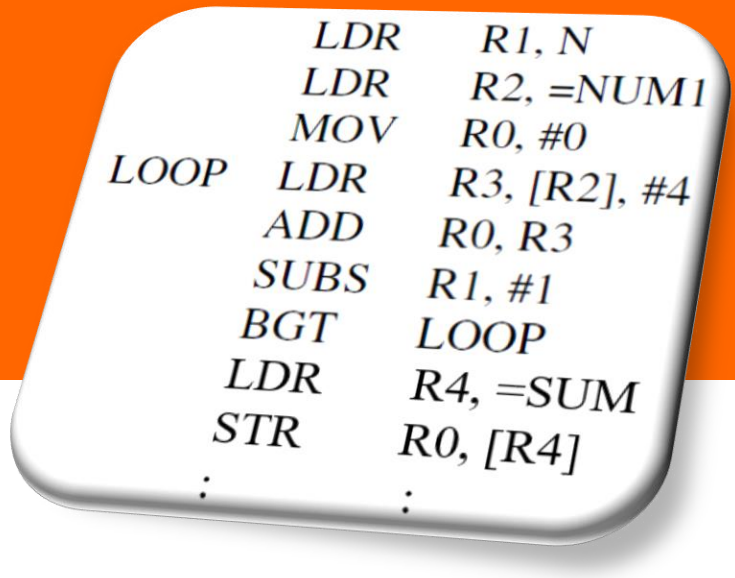
CG2028

Computer Organization

Lecture 2 & 3

ARM Assembly Language

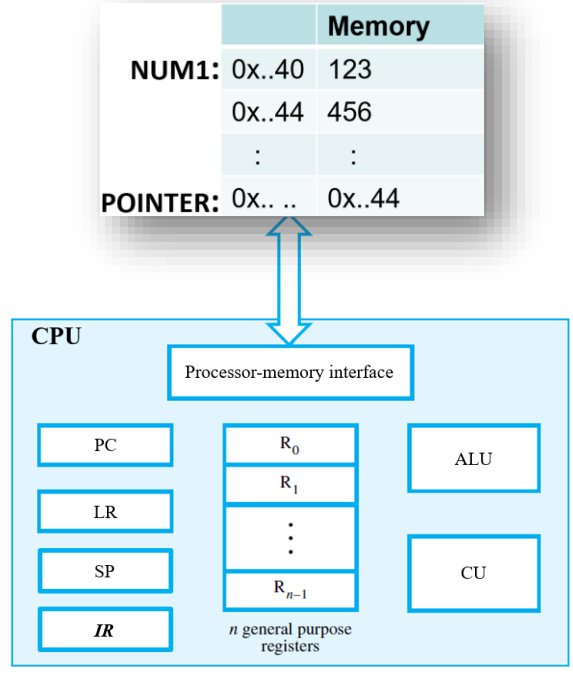
Dr Henry Tan, ECE, NUS
E-mail: eletanh@nus.edu.sg



ARM Instruction Set & Asm

- Objectives:
  - Understand the characteristics of the ARMv7E-M instruction set architecture (ISA), memory addressing & asm instructions

- Outline:
  - 1. Introduction to ARMv7E-M Assembly Language
    - 1.1 Why asm?
    - 1.2 Calling asm from C Program
    - 1.3 ARMv7E-M Glossary: label, optional, Op2, #Imm, Pre- & Suffix
  - 2. Memory Addressing
    - 2.1 Memory Allocation for User Data (using Assembler Directives)
    - 2.2 Offset Addressing
    - 2.3 Offset Addressing – with Pre/Post Index
    - 2.4 PC-Relative Addressing
    - 2.5 Pseudo-Instruction Addressing
  - 3. ARMv7E-M Ctrl & Arithmetic Instructions
    - 3.1 Move
    - 3.2 Add & Subtract
    - 3.3 Multiply & Multiply with Accumulate/Subtract, Divide
    - 3.4 Compare
    - 3.5 Branch
  - 4. Conditional Execution & Condition Code Suffixes
    - 4.1 Conditional Branch
    - 4.2 IT Block
  - 5. ARMv7E-M Logic Instructions
    - 5.1 And, Or, Xor
    - 5.2 Not
    - 5.3 Shift & Rotate
    - 5.4 Test
  - 6. Stack & Subroutines/Functions



1. Introduction to ARMv7E-M

1.1 Why learn Assembly Language?

- Assembly programs are **quicker**, **smaller** & have **more 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.
- Assembly allows an **ideal optimization** in programs, be it their size or their execution speed.
- However, developing applications in assembly is **tedious** & **error-prone**. .: *Reserved for startup/boot code, low-level OS features, etc*
- **Combining C and asm** is more **feasible** & **powerful**
  → our approach in this course! Both co-exist in a project/file

1.2 Calling an Assembler Function from a C Program

- Call assembler function from the C program (.c)
  - `extern int my_asm_func(int x, int y);`
  - It will be treated as just another C function by the C program
  - Input parameters: `R0`, `R1`, `R2`, `R3` (maximum of 4)
  - Output (return) parameter: `R0`
- Define assembler function in the 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 ← LR *(will be covered later)*
- This method is used in Assignment
  - Refer to the assignment skeleton code

1.3 ARMv7E-M Glossary: i. label

NUS National University of Singapore

Common instruction format

```
label: opcode operand1, operand2, .. @ Comments
```

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

Comments/Remarks are inserted after @ or ;

`label` is optional

i.e. **labels** are symbolic representations of **addresses!**

- 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/function calls
- 2. a **user-defined data** (e.g. using **.word** assembler directive)
  - e.g. PI: **.word** 314 @ constant PI whose value is 314, recall we can "*Load R2, A*"!

Another way of defining a constant without using **.word** & label, but **.equ** & symbol:
e.g. **.equ** PI, 314 @ constant PI whose value is 314 too *(more on .word & .equ later …)*

1.3 ARMv7E-M Glossary: ii. optional

NUS National University of Singapore

➢ Optional:

{ } Any item bracketed by { and } is optional. A description of the item and of how its presence or absence is encoded in the instruction is normally supplied by subsequent text.

Many instructions have an optional destination register. Unless otherwise stated, if such a destination register is omitted, it is the same as the immediately following source register in the instruction syntax.

Mnemonic	Operands	Brief description	Flags
ADC, ADCS	{Rd,} Rn, Op2	Add with Carry	N,Z,C,V
ADD, ADDS	{Rd,} Rn, Op2	Add	N,Z,C,V
ADD, ADDW	{Rd,} Rn, #imm12	Add	N,Z,C,V
ADR	Rd, label	Load PC-relative address	-
AND, ANDS	{Rd,} Rn, Op2	Logical AND	N,Z,C

e.g. ADD R0, R1, R2 performs R0 ← R1+R2

ADD R1, R2 performs R1 ← R1+R2

1.3 ARMv7E-M Glossary: iii. Op2

➢ *Operand2* – a flexible 2nd ***source*** operand

- Available to some Arithmetic/Logic & Move instructions
- can be 1. a Constant or 2. a Register (with or without shift*)

1. Constant (aka Immediate)

- Specify a constant in the form: ***#constant***
  where ***constant*** must be **<imm8>**

Operand2

```
e.g.
ADD R0, R1, #0xFF
ADD R0, R1, R2
ADD R0, R1, R2, LSL #0x4
```

2. Register, with an optional shift

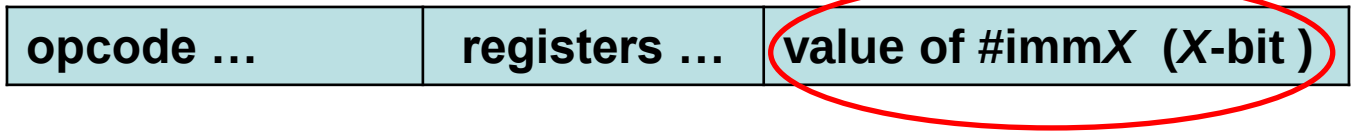
- Specify a register in the form: *Rm {, <shift>}*
  where:
- *Rm* is the register holding the data for the 2nd source operand
- *<shift>* is an optional shift applied to *Rm*, e.g. Logical Shift Left

**Inline Barrel Shifter will be discussed later in the Logic Instructions section*

1.3 ARMv7E-M Glossary: iv. #Immediate

- **Immediate operands** – explicitly hardcoded (**#**constants)

*32-bit instruction word in **IR** :*



- #imm8
  - 2^8 Range: 0 to 255 or -128 to 127
- #imm12
  - 2^{12} Range: 0 to 4095 or -2048 to 2047
- #imm16
  - 2^{16} Range: 0 to 65535 or -32768 to 32767

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

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

1.3 ARMv7E-M Glossary: v. Pre- & Suffix

Some Arithmetic/Logic instructions may have:

- **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`

Most Arithmetic/Logic/Move instruction may ***optionally*** have:

- **Suffix -S**
  - *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 ARMv7E-M Glossary: v. Suffix (Condition *Code* Flags)

NUS National University of Singapore

- The Application Program Status Register (APSR, a special register) 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** (meaningful 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** (meaningful for signed operations) occurs:
  - if the result of an **addition/subtraction/compare** is **greater than or equal to 2^{31}, or less than -2^{31}**

Note: With a few exceptions, most operations use the same computation hardware for both signed and unsigned operands. Flags are then set accordingly. The programmers must know which flag to check

2. Memory Addressing

2.1 User Data Declaration_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 the 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; similar to C pre-processor directive ***#define*** – every occurrence of the symbol is substituted by the expression in the program; e.g. LDR R1, **=**PI @ loads R1 with 314, no memory is allocated

- **.word** allocates a word-sized amount of storage space in that memory location. It also initializes that location with the given value (& its consecutive locations if multiple values are declared); roughly, similar to ***const int***, ***int****, & their ***unsigned*** equivalents in C
  e.g. LDR R2, NUM1 @ loads R2 with 123

		Memory
NUM1:	0x..40	123
	0x..44	456
	:	:
POINTER:	0x.. ..	0x..44

2. Memory Addressing
2.1 User Data Declaration_2

	Memory
ANSWER: 0x..4C	*RESERVED*
0x..50	….

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

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

- A static variable **retains its value** even when the function exits
- Its lifetime (or "extent") is the entire run of the program (i.e. global)
- .lcomm reserves the specified **number of bytes** of memory location for **global variable** whose value is **not yet available** at the time of coding

Note:

- .equ, .word & .lcomm are all assembler directives, ***not*** ARM instructions – **no need to include #** for the constants specified (unlike #immediate)
- STACK_TOP, PI, NUM1, POINTER & ANSWER in the above examples are all **hardcoded**

2. Memory Addressing

2.1 User Data Declaration_Program E.g.

```
@ Simple ARM assembly file to demonstrate basic asm instructions
        .syntax unified
        .global main

@ Equates, equivalent to #define in C program
        .equ C, 20
        .equ D, 400

main:
@ Code starts here
@ Calculate ANSWER = A*B + C*D

        LDR R0, A
        LDR R1, B
        MUL R0, R0, R1
        LDR R1, =C
        LDR R2, =D
        MLA R0, R1, R2, R0
        MOV R4, R0
        LDR R3, =ANSWER
        STR R4, [R3]

HALT:
        B HALT

@ Define constant values
A:      .word   100
B:      .word   50

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

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	.equ symbol, expression
Constant, by memory allocation	LABEL: .word constant-value(s) or pointer
Static Variable	.lcomm LABEL no.-of-bytes-reserved

Load/Store:		
	Basic Offset	LDR/STR Rt, [Rn]
		LDR/STR Rt, [Rn, #offset]
	Pre-index Offset	LDR/STR Rt, [Rn, #offset]!
	Post-index Offset	LDR/STR Rt, [Rn], #offset
	PC-relative	LDR Rd, LABEL
	Pseudo-instruction	LDR Rd,=value or =LABEL

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 the value **from** the stated memory
- STR : stores the value in register *Rt* **to** the stated memory

- *Rn* (the **base register**) contains the address of the memory location containing data,

i.e. the **value** of *Rn* is the **memory address** we're interested in!

- **but** 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 contents transferred ***(for LDR)***

- The value in register *Rn* is unchanged

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	(Base Register)
R3	0x….0123	
R4		
…		
…		

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

Note: In all these examples, data & addresses are all 32-bit values.
When fewer than 32 bits is shown, please assume that the given value is left extended to 32 bits.

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	(Base Register)
R3	0x….0456	
R4		
…		
…		

- ***Immediate* offset is 0x4** in this example
- EA = R2 + 0x4 = 0x……44
- R3 ← 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*

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

- **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
  - Contents 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***
  - Contents of the base register ***Rn*** is **updated with this result** ***(but not used as EA!)***

2.3 Offset Addressing: Pre-Indexed vs Post-Indexed

NUS National University of Singapore

➢ That is:

Difference: **offset** is used to determine **EA** before memory access in Pre-indexed, while the **offset** is only used to update ***Rn*** (and not **EA**) after the memory access is completed in Post-indexed.

Pre-Indexed Addressing *while*

LDR Rd, [Rn, #offset]!

performs

Rd ← [Rn + offset]

followed by

Rn ← Rn + offset

Post-Indexed Addressing

LDR Rd, [Rn], #offset

performs

Rd ← [Rn]

followed by

Rn ← Rn + offset

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

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	(Base Register)
R3		
R4		
…		
…		

- ***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**	(Base Register)
R3	**0x….0456**	
R4		
…		
…		

- ***Immediate* offset is 4** in this example
- EA = R2 + 4 = 0x……44
- R3 ← 0x….0456
- R2 ← 0x……44

Contents 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	(Base Register)
R3		
R4		
…		
…		

- ***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**	(Base Register)
R3	**0x….0123**	
R4		
…		
…		

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

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

2.4 PC-Relative Addressing (*LDR only*)

- **Most convenient way** of memory addressing, ≈ *Load R2, A*
- But **only for LDR**, **not available for STR** in Cortex-M4!
- Assembly language format:

LDR ***Rd*, ITEM**

performs

Rd ← [PC + offset]

where EA = PC + offset &
this **offset is calculated by the assembler**

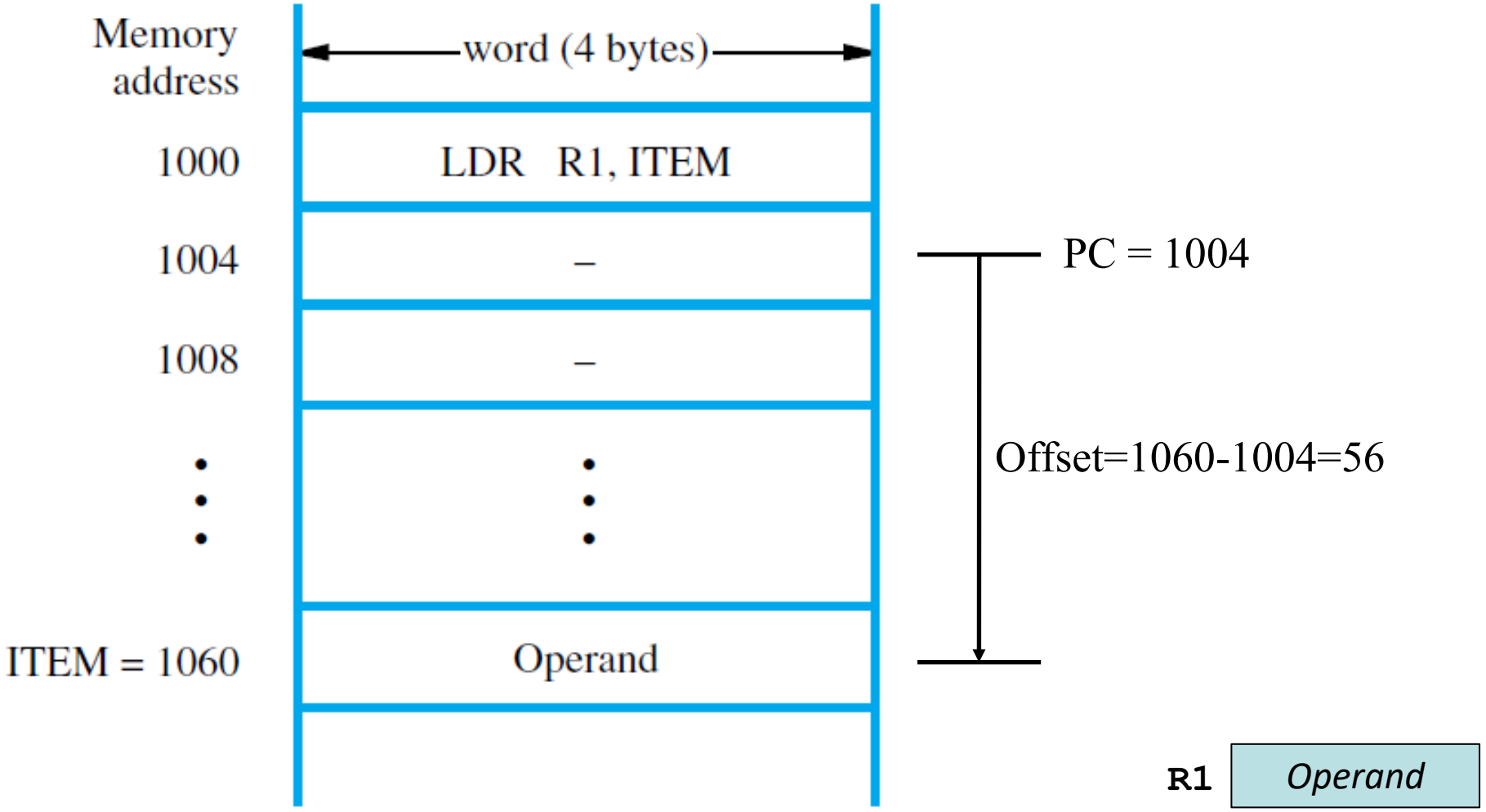
ITEM here is a hardcoded **label** which must be within a limited offset range of **-4095 to 4095** for successful PC-Relative Addressing! But this is the assembler's job.

- The **base register is *always* the PC**
- LDR loads the destination register (Rd) with the value from a PC-relative memory address specified by the **label** (ITEM)
- Location of data is ***always relative*** to that of the instruction

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 (*LDR only*)

- Pseudo-instruction – **NOT a real** assembly instruction
- It does not have a direct machine language equivalent *(& no STR!)*
- But the assembler will convert it into **PC-relative LDR**, i.e.
  Rd ← [PC + offset] during assembly to produce the required data
- Useful for loading a 17 to 32-bit **value**/a memory **address** ***(often)***
- E.g. 1, loading a **32-bit** value:

```
LDR     R1, =0xA123B456     (programmer: instruction)
```

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

is implemented with:

```
        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. 2, 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":

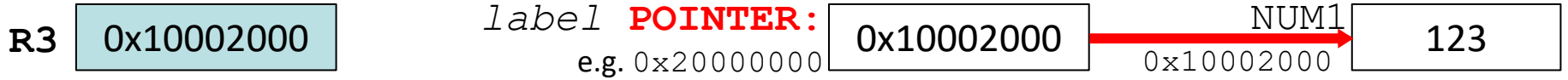
  LDR R3, POINTER (assembler: instruction)

- & 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:	0x00008000	0x100		R7	0x7
	0x00008004	0x104		:	

Eg 8: Memory Addressing Application

NUS National University of Singapore

- 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