Lect (3)　　　بسم الله الرحمن الرحيم　　　Drive 2022

* Assembelj language for ARM Processor

→ Syntax : Assembly language instructions have four fields separated by spaces or tabs

1) label Field : is optional and starts in the first column and is used to identify the position in memory of the current instruction. You must choose a unique name for each label

2) OP-Code Field : specifies the processor command to execute

3) operand Field : specifies where to find the data to execute the instruction

4) Comment field :

is also optional and is ignored by the assembler, but it allows you to describe the software making it easier to understand. You can add optional spaces between operands in the operand field. However, a semicolon must separate the operand and comment fields. Good programmers add comments to explain the software

Label	Opcode	Operands	Comment
Func	**MOV**	**R0, #100**	**; this sets R0 to 100**
BX LR	**; this is a function return**		

→ bad Comments

* Good Comment : explains why an operation is being performed, how it is used, how it can be changed, or how it was debugged.

* bad Comment : explains what the operation does

When describing assembly instructions we will use the following list of symbols

Ra Rd Rm Rn Rt and **Rt2** represent registers
{Rd,} represents an optional destination register
#imm12 represents a 12-bit constant, 0 to 4095
#imm16 represents a 16-bit constant, 0 to 65535
operand2 represents the flexible second operand as described
{cond} represents an optional logical condition
{type} encloses an optional data type
{S} is an optional specification that this instruction sets the condition code bits
Rm {, shift} specifies an optional shift on **Rm**

Rn {, #offset} specifies an optional offset to **Rn**

1st example

$\boxed{\text{ADD R1,\#10}}$; R1= R1 +10

So, constant part after "#"

$\boxed{\text{ADD R1,R0,\#10}}$; R1 =R0+10

general form of ADD istruction is :

$\boxed{\text{ADD \{Cond\} \{Rd,\} Rn, \#imm12}}$

optional optional

MOV → instr. move data within process without accessing memory.

MOV R0 #10 ; R0 =10

* LSL R2,#2 ; R2 x4
logical shift left

ARM data instructions

- Basic format:

① ADD R0,R1,R2
  - Computes R1+R2, stores in R0.

② ADD R0,R1 ; R0 = R0 + R1

So, if Rd [destination register] is missed Rn is the destination

example

- Immediate operand:

  ADD R0,R1,#2
  - Computes R1+R2, stores in R0.

ADD Rd, Rn, Rm {,shift} ;Rd = Rn+Rm
ADD Rn, Rm {,shift} ;Rn = Rn+Rm

for more inf. about Rm{, shift} open text book @ 124

ARM data instructions

- ADD, ADC : add (w. carry)
- SUB, SBC : subtract (w. carry)
- MUL: multiply
- AND, ORR, EOR
- BIC : bit clear
- LSL, LSR : logical shift left/right
- ROR : rotate right

imp.

A **Rn**	B **Operand2**	A&B **AND**	A|B **ORR**	A^B **EOR**	A&(~B) **BIC**	A|(~B) **ORN**
0	0	0	0	0	0	1
0	1	0	1	1	0	0
1	0	0	1	1	1	1
1	1	1	1	0	0	1

ARM load/store instructions

load word   load Half-Word   load byts

- LDR, LDRH, LDRB : load (half-word, byte)
- STR, STRH, STRB : store (half-word, byte)
- Addressing modes:
  - register indirect : LDR R0,[R1] ; R0 = Value Pointed to by R1
  - with constant : LDR R0,[R1,#4]

imp. Notes

```
LDR   R0,[R1]      ; R0= value pointed to by R1
LDR   R0,[R1,#4]   ; R0= word pointed to by R1+4
LDR   R0,[R1,#4]!  ; first R1=R1+4, then R0= word pointed to by R1
LDR   R0,[R1],#4   ; R0= word pointed to by R1, then R1=R1+4
LDR   R0,[R1,R2]   ; R0= word pointed to by R1+R2
LDR   R0,[R1,R2, LSL #2] ; R0= word pointed to by R1+4*R2
```

ARM LDR instruction

- Load from memory into a register

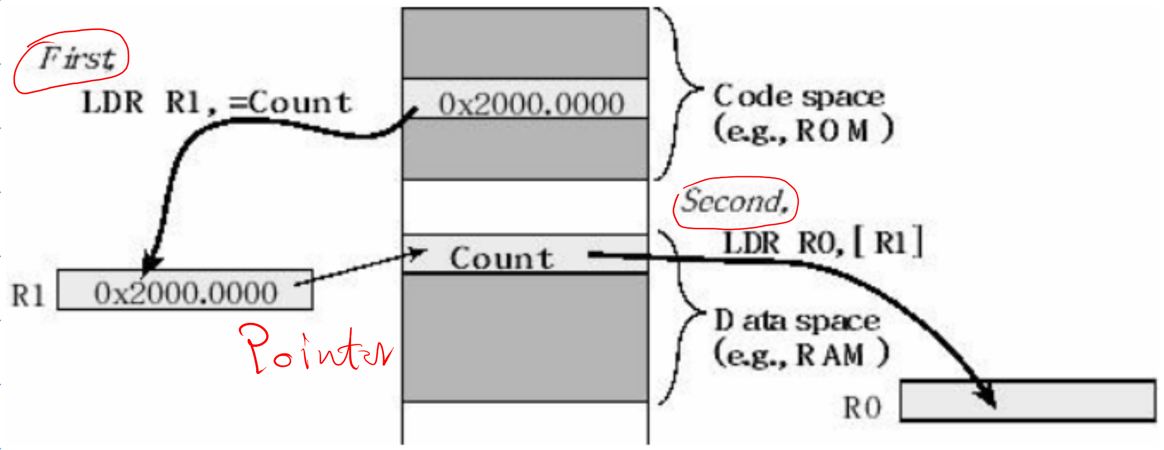
```
LDR R8,[R10]
```

for the following examle

it takes two instructions to access data in RAM or I/O.
The first instruction uses PCrelative addressing to create a pointer to the object, and the second instruction accesses the memory using the pointer.We can use the =Something operand for any symbol defined by our program.
In this case Count is the label defining a 32-bit variable in RAM

```
LDR   R1,=Count    ; R1 points to variable Count, using PC-relative
LDR   R0,[R1]      ; R0= value pointed to by R1
LDR   R1,=Count    ; R1 points to variable Count, using PC-relative
LDR   R0,[R1]      ; R0= value pointed to by R1
```

The operation caused by the abovetwo **LDR** instructions is illustrated in Figure 3.13. Assume a 32-bit variable **Count** is located in the data space at RAM address 0x2000.0000. First, **LDR R1,=Count** makes R1 equal to 0x2000.0000. I.e., R1 points to **Count** . The assembler places a constant 0x2000.0000 in code spaceand translates the **=Count** into the correct PC-relative access to the constant (e.g., **LDR R1,[PC,#28]**). In this case, the constant 0x2000.0000, the address of **Count** will be located at PC+28. Second, the **LDR R0,[R1]** instruction will dereference this pointer, bringing the 32-bit contents at location 0x2000.0000 into R0. Since **Count** is located at 0x2000.0000, these two instructionswill read the value of **Count** into R0.



imp. for the last slide

PC-relative addressing. PC-relative addressing is indexed addressing mode using the PC as the pointer. The PC always points to the instruction that will be fetched next, so changing the PC will cause the program to branch. A simple example of PC-relative addressing is the unconditional branch. In assembly language, we simply specify the label to which we wish to jump, and the assembler encodes the instruction with the appropriate PC-relative offset.

```
B    Location    ; jump to Location, using PC-relative addressing
```

The same addressing mode is used for a function call. Upon executing the **BL** instruction, the return address is saved in the link register (LR). In assembly language, we simply specify the label defining the start of the function, and the assembler creates the appropriate PC-relative offset.

```
BL   Subroutine   ; call Subroutine, using PC-relative addressing
```

B{cond}	**label**	**;branch to label**
BX{cond}	**Rm**	**;branch indirect to location specified by Rm**
BL{cond}	**label**	**;branch to subroutine at label**
BLX{cond}	**Rm**	**;branch to subroutine indirect specified by Rm**

Assembly code	C code
`LDR R2, =G      ; R2 = &G`	
`LDR R0, [R2]    ; R0 = G`	`uint32_t G;`
`CMP R0, #7      ; is G == 7 ?`	`if(G == 7){`
`BNE next1       ; if not, skip`	`  GEqual7();`
`BL  GEqual7     ; G == 7`	`}`
`next1`	
`LDR R2, =G      ; R2 = &G`	
`LDR R0, [R2]    ; R0 = G`	`if(G != 7){`
`CMP R0, #7      ; is G != 7 ?`	`  GNotEqual7();`
`BEQ next2       ; if not, skip`	`}`
`BL  GNotEqual7 ; G != 7`	
`next2`	

لازم عشان أستخدم أي Branch أستعمل CMP الأول

وعلى أساس ال C-Code أحدد نوع ال Branch

طب ليه ما أستعملش Branch بس على طول ليه لازم CMP

عشان ال Compiler يترجم إنه في مقارنة بين الاتنين وبعد كده

السطر اللي بعده يحدد نوع المقارنة سواء >= ولا <=

ولا == ولا !=

وإنه في ال ARM assembl

ال Branches لما بتكتبها في الكود بيجي بعدها label أو

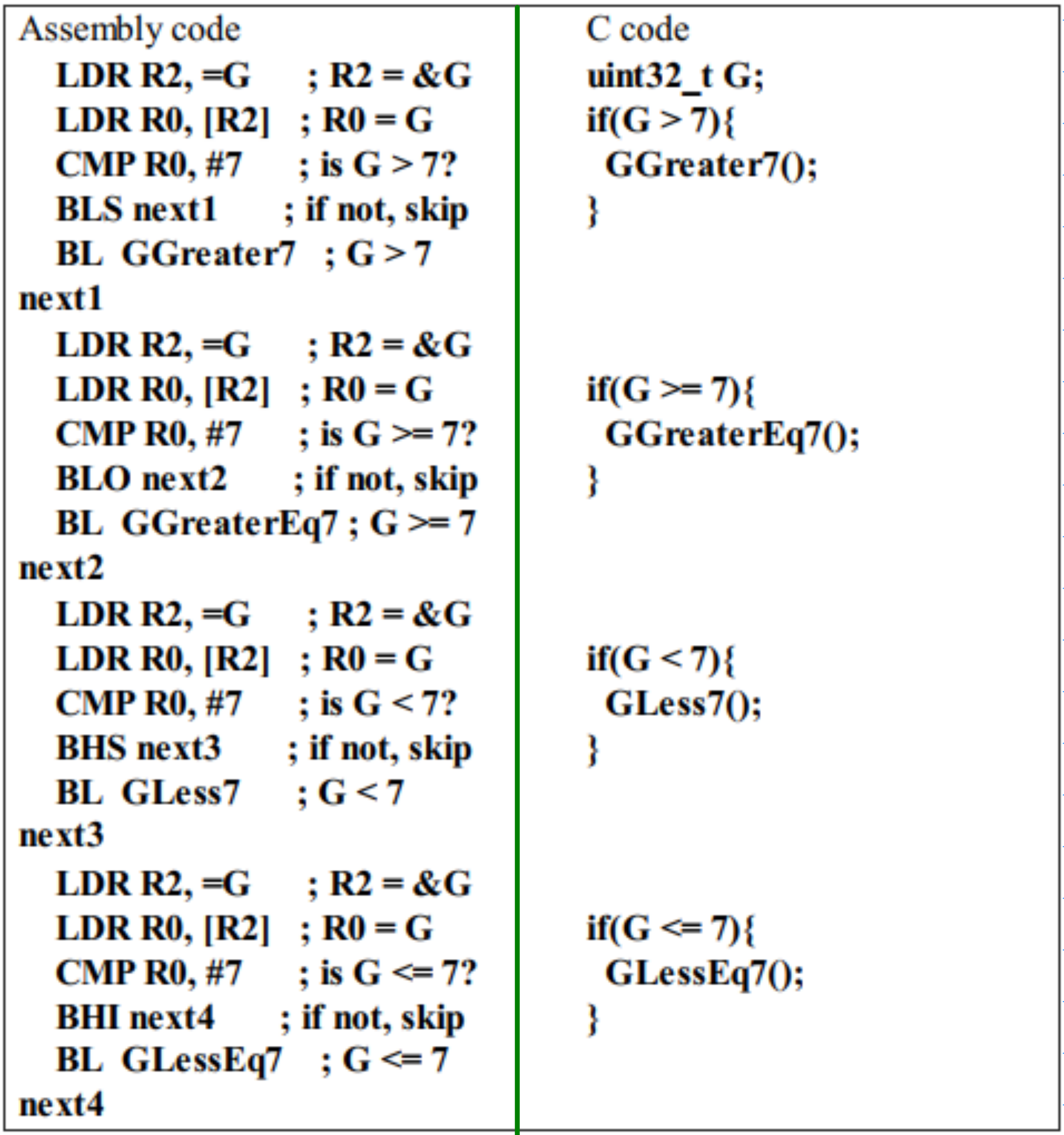
ممكن register واحد بس ساعتها هتكون jump

ف بستعمل CMP عشان أكتب الحاجتين اللي عاوز أقارنه مابينهم

على عكس ال MIPS كان بيسمحلي أكتب بعد beq

bne

It takes three steps to perform a comparison. You begin by
reading the first value into a register. If the second value is not a constant,
it must be read into a register, too. The second step is to compare the first value with
the second value. You can use either a subtract instruction with the S (SUBS) or
a compare instruction (CMPCMN). The CMP CMN
SUBS instructions set the condition code bits. The last step is a conditional branch



Assembly code	C code
LDR R2, =G ; R2 = &G	uint32_t G;
LDR R0, [R2] ; R0 = G	if(G > 7){
CMP R0, #7 ; is G > 7?	GGreater7();
BLS next1 ; if not, skip	}
BL GGreater7 ; G > 7	
next1	
LDR R2, =G ; R2 = &G	
LDR R0, [R2] ; R0 = G	if(G >= 7){
CMP R0, #7 ; is G >= 7?	GGreaterEq7();
BLO next2 ; if not, skip	}
BL GGreaterEq7 ; G >= 7	
next2	
LDR R2, =G ; R2 = &G	
LDR R0, [R2] ; R0 = G	if(G < 7){
CMP R0, #7 ; is G < 7?	GLess7();
BHS next3 ; if not, skip	}
BL GLess7 ; G < 7	
next3	
LDR R2, =G ; R2 = &G	
LDR R0, [R2] ; R0 = G	if(G <= 7){
CMP R0, #7 ; is G <= 7?	GLessEq7();
BHI next4 ; if not, skip	}
BL GLessEq7 ; G <= 7	
next4	

BL $\rightarrow$ likes jal in MIPS

B $\rightarrow$ " j " "

imp. Note

→ CMP → do Subtraction between it's operands

if I said CMP R0, R1 ; means $\boxed{R0 - R1}$

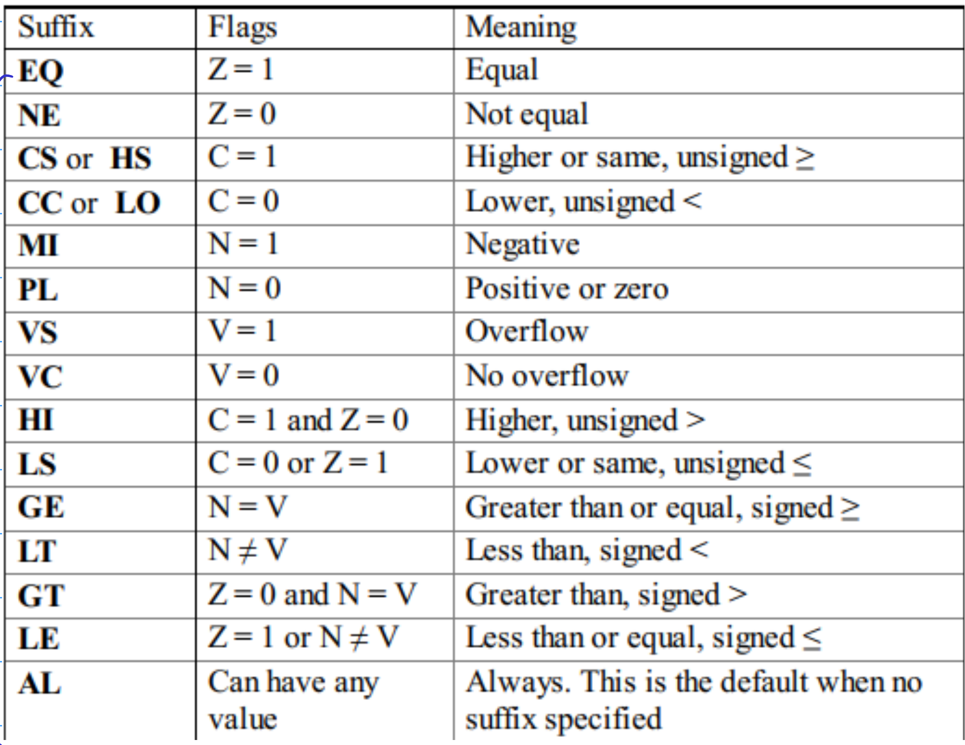
→ branches [BEQ , BNE, ...] consist from 2-parts

$\underset{\text{jump to certain label}}{B}$ + Suffix.

after doing sub we compare between it's result and flags [BEQ, BNE → check Z flag , ...]

2 instead of using Branches I can use Suffix →

let's explain with example.



Suffix	Flags	Meaning
EQ	Z = 1	Equal
NE	Z = 0	Not equal
CS or **HS**	C = 1	Higher or same, unsigned ≥
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 ≤
GE	N = V	Greater than or equal, signed ≥
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 ≤
AL	Can have any value	Always. This is the default when no suffix specified

ex:

```
r2 = 0;
while (r1 != 0) {
    if ((r1 & 1) != 0) {
        r2 += r0;
    }
    r0 <<= 1;
    r1 >>= 1;
}
while (1); // halting loop
```

Soln

يعني هنفذ ADD في حالة عدم التساوي
إنما في حالة التساوي هسكيب ADD

```
LOOP_2      MOV   R2, #0 ;   r2 = 0
            CMP   R1, #0 ;   is (r1 == 0)
            BEQ   Halt_LOOP;           نفذ ده
            TST   R1, #1
            ADDNE R2, R2, R0
            LSL   R0, R0, #1
            ASR   R1, R1, #1
            B     LOOP_2

Halt_LOOP   B     Halt_LOOP
```

TST هي Sub ، AND بس مش بتخزن الresult في مكان معين حاجة في الهوا كده يعني

$\underbrace{\text{ADD}}_{\text{inst.}}\ \underbrace{\text{NE}}_{\hookrightarrow \text{Suffix } \{\text{Cond.}\}}$ (not equal)

(1) without using branches we used ADDNE --> where NE is our detector to jump to label or continue to the next instruction

(2) simply, i can say that i will execute ADDNE in case of not equal otherwise skip ADDNE and execute the next instruction which is "LSL" in this example

ANDS → $\boxed{\text{AND} + \text{Sub}}$

These instructions test the value in a register against `Operand2`. They update the condition flags on the result, but do not place the result in any register.

The `TST` instruction performs a bitwise AND operation on the value in `Rn` and the value of `Operand2`. This is the same as a `ANDS` instruction, except that the result is discarded.

The `TEQ` instruction performs a bitwise Exclusive OR operation on the value in `Rn` and the value of `Operand2`. This is the same as a `EORS` instruction, except that the result is

Example: C assignments

- C:

```
x = (a + b) - c;
```

- Assembler:

```
LDR R4,=A               ; get address for a
LDR R0,[R4]                  ; get value of a
LDR R4,=B               ; get address for b, reusing r4
LDR R1,[R4]                  ; get value of b
ADD R3,R0,R1                 ; compute a+b
LDR R4,=C               ; get address for c
LDR R2,[R4]                  ; get value of c
```

C assignment, cont’d.

```
SUB R3,R3,R2        ; complete computation of x
LDR R4,=X           ; get address for x
STR R3,[R4]         ; store value of x
```

Example: C assignment

- C:

```
y = a*(b+c);
```

- Assembler:

```
LDR R4,=B ; get address for b
LDR R0,[R4] ; get value of b
LDR R4,=C ; get address for c
LDR R1,[R4] ; get value of c
ADD R2,R0,R1 ; compute partial result
LDR R4,=A ; get address for a
LDR R0,[R4] ; get value of a
```

C assignment, cont’d.

```
MUL R2,R2,R0 ; compute final value for y
LDR R4,=Y ; get address for y
STR R2,[R4] ; store y
```

recall

<< LSL

>> LSR

| OR

& AND

Example: C assignment

- C:

```
Z = (A << 2) |  (B & 15);
```

- Assembler:

```
LDR R4,=A  ; get address for a
LDR R0,[R4]  ; get value of a
LSL R5,R0,#2  ; perform shift
LDR R4,=B  ; get address for b
LDR R1,[R4]  ; get value of b
AND R1,R1,#15  ; perform AND
ORR R1,R5,R1  ; perform OR
```

```
LDR R4, =A
LDR R0,[R4]
LSL R5,R0,#2
LDR R4, =B
LDR R1, [R4]
AND R1, R1, #15
ORR R1, R1, R5
LDR R4, =Z
STR R1,[R4]
```

R4 Pointer to R0

C assignment, cont’d.

```
LDR R4,=Z ; get address for z
STR R1,[R4] ; store value for z
```

If statement, cont'd.

Comment

```
    ; false block
label fblock LDR R4,=C ; get address for c
       LDR R0,[R4] ; get value of c   R0 = Value of C
       LDR R4,=D ; get address for d
       LDR R1,[R4] ; get value for d  R1 = Value of D
       SUB R0,R0,R1 ; compute a-b     R0 = R0 - R1
       LDR R4,=X ; get address for x
       STR R0,[R4] ; store value of x
    after ...
```

Example: if statement

- C:

```
if (a > b) { x = 5; y = c + d; } else x = c - d;
```

- Assembler:

```
; compute and test condition
   LDR R4,=A ; get address for a
   LDR R0,[R4] ; get value of a
   LDR R4,=B ; get address for b
   LDR R1,[R4] ; get value for b
   CMP R0,R1 ;
   BLE fblock ;
```

Branch less than or equal