Chapter 6: Modular Programming

Mohamed M. Sabry Aly
N4-02c-92

1

Learning Objectives

- Describe ARM instructions in implement subroutines
- Describe passing parameters to subroutines using:
  - Registers
  - Memory

2

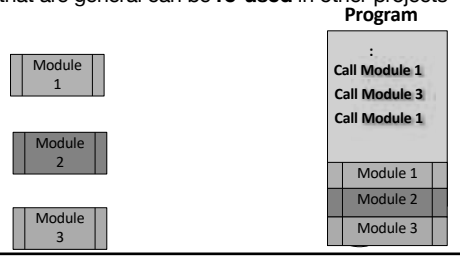
Modular Program Design

- Real-world applications are very large and complex
  - Google Chrome: ~6.7 Millions line of code
  - Android OS: 12-15 Millions line of code
  - Boeing 787: ~6.5 Millions line of code
- Large software cannot be a single function
- It is decomposed into several less complex **modules**

3

Modular Program Design

- Software decomposed to several less complex **modules**
  - Modules can be designed and tested **independently**
  - Modules can reduce overall **program size**
    - Same module may be required in several places
  - Modules that are general can be **re-used** in other projects



4

Characteristics of a Good SW Module

- Loose coupling—**data** within module is entirely **independent** of other modules (local variables)
- Strong modularity—should perform a **single** logically coherent **task**

5

Example: Standard Deviation

Case 1: all in one c main()

```
int main() {
    .
    float avg=0.0; //initialization
    for(int i=0;i<N;i++){
       avg +=x[i];
    }
    avg/=N;
    float sigma = 0.0;
    for(int i=0;i<N;i++){
       sigma +=pow(x[i]-avg),2);
    }
    sigma/=N;
    sigma=sqrt(sigma);
    return 0;
}
```

6

6

Example: Standard Deviation

Case 2: Modular

```
int main() {
    …
    float avg = sum(x,N)/N;
    float sigma = Sigma_f(x,avg,N);
    return 0;
}
```

```
float sum( int* x, int N){
    float tot=0.0; //initialization
    for(int i=0;i<N;i++){
       tot +=x[i];
    }
    return tot;
}
```

```
float Sigma_f( int* x, float avg, int N){
    float sigma = 0.0;
    for(int i=0;i<N;i++){
       sigma +=pow(x[i]-avg),2);
    }
    sigma/=N;
    sigma=sqrt(sigma);
    return sigma;
}
```

7

7

Example: Standard Deviation

Case 2: Modular

```
int main() {
    …
    float avg = sum(x,N)/N;
    float sigma = Sigma_f(x,avg,N);
    return 0;
}
```

How will this be translated to assembly instructions?

How will ARM go to these two functions and return?

```
float sum( int* x, int N){
    float tot=0.0; //initialization
    for(int i=0;i<N;i++){
       tot +=x[i];
    }
    return tot;
}
```

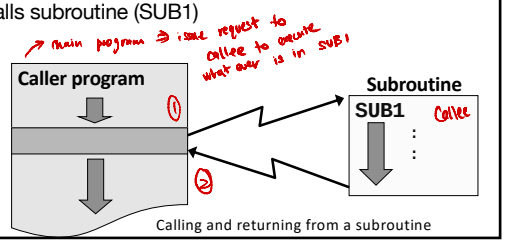
```
float Sigma_f( int* x, float avg, int N){
    float sigma = 0.0;
    for(int i=0;i<N;i++){
       sigma +=pow(x[i]-avg),2);
    }
    sigma/=N;
    sigma=sqrt(sigma);
    return sigma;
}
```

8

8

Subroutines

- Modules (e.g., functions in C) are implemented as **subroutines**
- Subroutine can be called from various parts of the program
- Caller and callee
  - Caller: the program that calls subroutine (SUB1)
  - Callee: subroutine (SUB1)



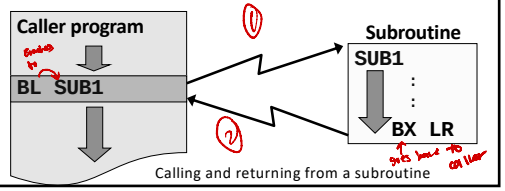
Calling and returning from a subroutine

9

Subroutines

- On completion, subroutine returns control to the caller
  - Exactly after the subroutine was called
- Calling and returning from a subroutine
  - To go to subroutine (SUB1): **BL SUB1**
  - To return to caller program: **BX LR**

`BL: branch with link`
`BX: branch and exchange`



Calling and returning from a subroutine

10

need to store where you should be getting back after (1)

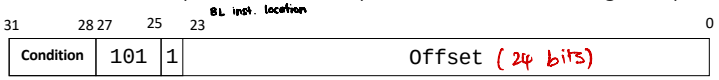
Why BL and BX and not B?

- Main program branches to subroutine:
  - Can be done with B ➔ what is the draw back?
  - B overwrites value in PC ➔ oldPC value in main program is LOST
  - Maybe add another branch at the end of subroutine ➔ need to know the exact mem location during compilation, not an effective approach

11

Branch with Link (BL)

- **BL** used to make subroutine call
- Return address (~~PC contents~~ + 4) is stored in the link register (R14)



- We can also conditionally make a functional call (more of this later)

12

Branch with Link (BL)

- **BL** used to make subroutine call

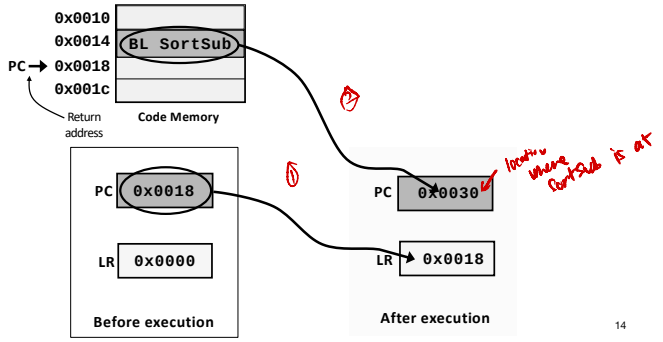
```
Subroutine Call
BL SortSub
```

- Execution sequence:
- Return address (~~PC contents~~ + 4) is stored in the link register (R14)
- The subroutine address is stored to the PC

13

BL (Execution example)

- BL SortSub (assume SortSub is in 0x0030)

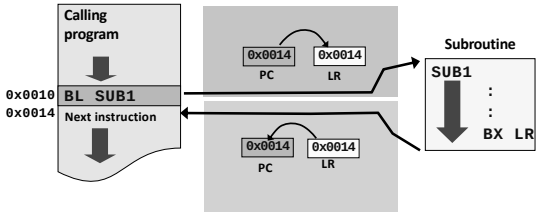


14

14

BX instruction

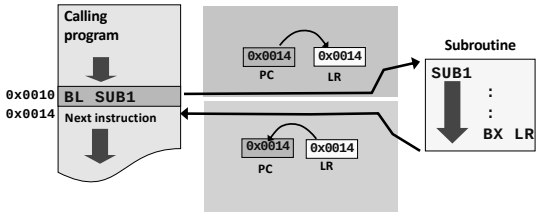
- `BX lr` returns from subroutine
- `lr` contains the return address, the instruction copies the value over to PC



15

BX instruction

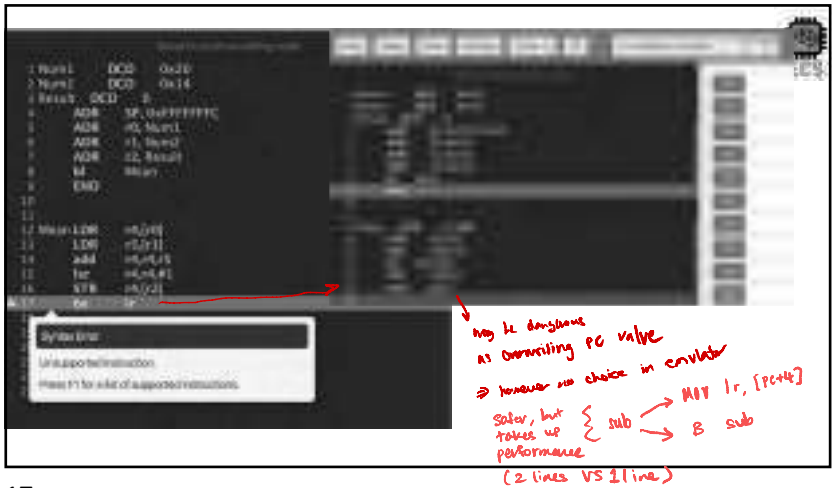
- `BX lr` returns from subroutine
- `lr` contains the return address, the instruction copies the value over to PC



- Note: VisUAL does not support `bx lr`
  - Use `MOV PC, LR`

16

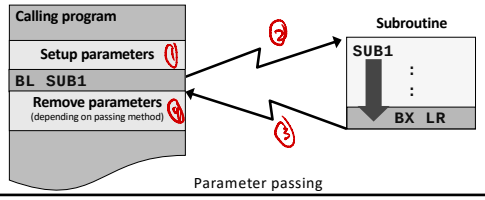
* allocate some place in the memory where it has its variable; allocation known to caller then send address to callee to perform operations



17

Parameter Passing

- Calling programs need to pass parameters to influence a subroutine's execution.
- Parameters must be setup properly before the subroutine is called and appropriately removed after returning.
- There are three basic methods to pass parameters, via **registers**, **memory block** or the **system stack**.



18

18

Parameter Passing using Registers

- Parameters are placed into the register before calling the subroutine
- **Number** of parameters passed are **limited** to the **available registers**
  - Useful when number of parameters are **small**
  - Not all **R0-R12** are preferred to pass parameters

Total registers : 16
⇒ However, Program Counter, Link Register & Stack Pointer cannot be used
Left : 13 (0 - 12)

19

Parameter Passing using Registers

- Parameters are placed into the register before calling the subroutine
- **Number** of parameters passed are **limited** to the **available registers**
  - Useful when number of parameters are **small**
  - Not all **R0-R12** are preferred to pass parameters

Calling convention

"free to use"

Later to store back up → in the stack

R0-R3	R4-R11
Can be used to pass argument values Also used to return values from subroutine Subroutine can modify values	Used to hold local variables Not for passing arguments Must be preserved in the subroutine

⇒ caller can pass parameters
⇒ callee can modify values

⇒ callee needs to store & copy that in a stack before it can work on its parameters

20

PC always pointing 8 bytes ahead of current instruction
⇒ point program counter to base address of subroutine first
⇒ then it will go 8 bytes ahead after pointing at base address

Parameter Passing using Registers

- Parameters are placed into the register before calling the subroutine
- **Number** of parameters passed are **limited** to the **available registers**
  - Useful when number of parameters are **small**
  - Not all **R0-R12** are preferred to pass parameters

Calling convention

pass by value or reference etc

R0-R3	**R4-R11**
Can be used to pass argument values to return values from subroutine Subroutine can modify values	Used to hold local variables Not for passing arguments Must be preserved in the subroutine

sometimes here

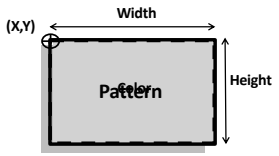
R12: Scratchpad register, does not need to be preserved
Can be used sometimes as return register

21

Parameter Passing using Registers

- Parameters are placed into the register before calling the subroutine
- **Number** of parameters passed are **limited** to the **available registers**
  - Useful when number of parameters are **small**
  - Not all **R0-R12** are preferred to pass parameters

```
Draw_rectangle(
X, Y,
Height, Width,
Border, Line_style
Fill, Color, Pattern,
Shadow);
```



22

22

Parameter Passing using Registers

- Parameters are placed into the register before calling the subroutine
- **Number** of parameters passed are **limited** to the **available registers**
  - Useful when number of parameters are **small**
  - Not all **R0-R12** are preferred to pass parameters

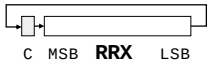
- **Pro** – **efficient** as parameters are already in register within the subroutine and can be used immediately.
- **Con** – **lacks generality** due to the limited number of registers.

23

23

Example: Bit Counting Subroutine

- Write a subroutine to:
  - Count the number of "1" bits in a word.
  - Return result in register **R0**.

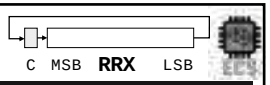


- **Design considerations:**
- How do we transfer the word into the subroutine?
  - Put the word into a **register**, which can then be accessed within the subroutine (e.g. register **R1**).
- How do we check if each individual bit is a "1" or a "0"?
  - Rotate **R1** right 32 times with the carry bit. After each rotate, test carry bit to check if (**C**=1). If yes, increment bit counter register **R0**.

24

24

Solution #1

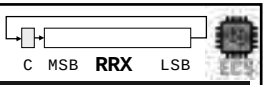



```
Count1s

        Start by labeling the subroutine
        and placing the return instruction

        MOV     PC, LR      ; same as bx lr
```

Value passed in R1, return value in R0 VisUAL does not support `bx lr`

25

Solution #1

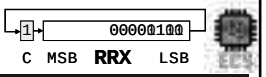



```
Count1s  MOV     R0, #0   ;Clear R0
         MOV     R2, #32  ; Set counter R2 with 32

        Initialize R0 with 0 and
        the counter register R2 with 32

         MOV     PC, LR      ; same as bx lr
```

Value passed in R1, return value in R0 VisUAL does not support `bx lr`

26

Solution #1

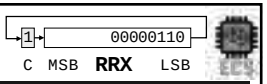



```
Count1s  MOV     R0, #0   ;Clear R0
         MOV     R2, #32  ; Set counter R2 with 32
Loop     RRXS    R1,R1    ; Rotate right and extend R1
        Rotate to the right (arithmetic), and use the carry
        S: set the status registers after rotation
         MOV     PC, LR      ; same as bx lr
```

Value passed in R1, return value in R0 VisUAL does not support `bx lr`

27

Solution #1

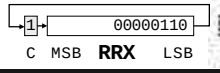



```
Count1s  MOV     R0, #0   ;Clear R0
         MOV     R2, #32  ; Set counter R2 with 32
Loop     RRXS    R1,R1    ; Rotate right and extend R1
         ADC     R0,R0,#0 ; Add the carry to R0
        Add the carry value to R0
         MOV     PC, LR      ; same as bx lr
```

Value passed in R1, return value in R0 VisUAL does not support `bx lr`

28

Solution #1




```
Count1s  MOV     R0, #0        ; Clear R0
         MOV     R2, #32       ; Set counter R2 with 32
Loop     RRXS    R1,R1         ; Rotate right and extend R1
         ADC     R0,R0,#0      ; Add the carry to R0
         SUBS    R2,R2,#1      ; decrement counter by 1
         BNE     Loop          ; loop if not zero
         MOV     PC, LR        ; same as bx lr
```

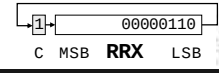
Decrement and set Status registers

Value passed in R1, return value in R0

VisUAL does not support `bx lr`

29

Solution #1




```
Count1s  MOV     R0, #0        ;Clear R0
         MOV     R2, #32  ; Set counterR2 with 32
Loop     RRXS    R1,R1         ; Rotate right and extend R1
         ADC     R0,R0,#0      ; Add the carry to R0
         SUBS    R2,R2,#1      ; decrement counter by 1
         BNE     Loop          ; loop if not zero
         MOV     PC, LR        ; same as bx lr
```

Issue, the carry of SUBS
Will be used in RRXS → R1 value is destroyed

Value passed in R1, return value in R0

VisUAL does not support `bx lr`

30

Solution #2

→ instead of MOV

```
Count1s  EOR     R0,R0,R0      ;Clear R0
         ADD     R2, R0, #32   ; Set counterR2 with 32
         ADD     R3, R0, #1    ; Set R3 with 1
Loop     AND     R4, R3, R1, ROR R2        ; ?
         ADD     R0,R0,R4      ; Add the lsb of R0
         SUBS    R2,R2,#1      ; decrement counter by 1
         BNE     Loop          ; loop if not zero
         MOV     PC, LR        ; same as bx lr
```

Value passed in R1, return value in R0

VisUAL does not support `bx lr`

31

Solution #2

```
Count1s  EOR     R0,R0,R0      ;Clear R0
         ADD     R2, R0, #32   ; Set counterR2 with 32
         ADD     R3, R0, #1    ; Set R3 with 1
Loop     AND     R4, R3, R1, ROR R2        ; ?
         ADD     R0,R0,R4      ; Add the lsb of R0
         SUBS    R2,R2,#1      ; decrement counter by 1
         BNE     Loop          ; loop if not zero
         MOV     PC, LR        ; same as bx lr
```

Value passed in R1, return value in R0

VisUAL does not support `bx lr`

32

Do an AND operator using R3 (mask) and rotated value of R1

R1, ROR R2 : take value of R1 and rotates it by R2 values

stores into RP

Solution #2

```
Count1s  EOR     R0,R0,R0     ;Clear R0
         ADD     R2, R0, #32  ; Set counterR2 with 32
         ADD     R3, R0, #1   ; Set R3 with 1
Loop     AND     R4, R3, R1, ROR R2        ; ?
```

Set R3 as a mask with a value of 1
Then apply this mask with a rotated value of R1

```
         MOV     PC, LR       ; same as bx lr
```

Value passed in R1, return value in R0 VisUAL does not support `bx lr`

33

Solution #2

```
Count1s  EOR     R0,R0,R0     ;Clear R0
         ADD     R2, R0, #32  ; Set counterR2 with 32
         ADD     R3, R0, #1   ; Set R3 with 1
Loop     AND     R4, R3, R1, ROR R2        ; ?
```

Set R3 as a mask with a value of 1
Then apply this mask with a rotated value of R1
The **rotated R1 is not stored anywhere**

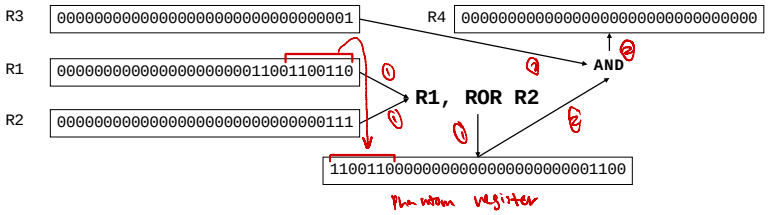
```
         MOV     PC, LR       ; same as bx lr
```

Value passed in R1, return value in R0 VisUAL does not support `bx lr`

34

A Deep Dive

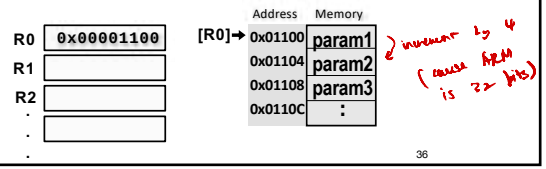
```
AND     R4, R3, R1, ROR R2
```



35

Parameter Passing using Memory

- A region in memory is treated like a mailbox and is used by both the calling program and subroutine.
  - Parameters to be passed are gathered into a **block** at a predefined memory location
  - The start address of the memory block is passed to the subroutine via an **address register**.
  - Useful for passing **large number of parameters**.



36

Example: Lower to Upper Case Subroutine

- Write a subroutine to:
  - To convert an ASCII string from lower to upper case.
  - The string is terminated by a NULL character (**0x00000000**).
  - The start address of the string is passed via **R0**.
  - A segment of the calling program shows how the parameter is setup and the subroutine called:

```
;Calling program
   :
MOV   R0,#0x100   ;move start addr. of string to R0
BL    Lo2Up       ;branch to Lo2Up subroutine
   :
```

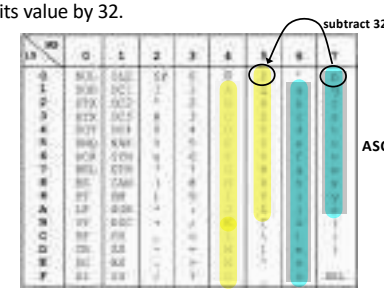
Address	Memory
0x100	"a"
0x104	"p"
0x108	"p"
0x10C	"l"
0x110	"e"
0x114	0x000
0x118	:

37

37

Algorithm Design

- How to convert an ASCII character from lower to upper case?
  - Check that the character's value is between 'a' and 'z'.
  - If so, subtract its value by 32.



38

38

Possible Solution

```
Lo2Up   STMFD   SP!,{r0,r1}   ;save registers used within subroutine
Loop    LDR     R1,[R0],#4    ;get current char from string in memory
        CMP     R1,#0         ;Compare with NULL
        BEQ     Done          ;if NULL char, branch to Done
        CMP     R1,#0x061     ;compare with lower limit "a"
        BLT     Loop          ;if smaller than "a", do not convert
        CMP     R1,#0x07A     ;compare with upper limit "z"
        BGT     Loop          ;if greater than "z" do not convert
        SUB     R1,R1,#32     ;convert to upper case by subtracting 32
        STR     R1,[R0,#-4]   ;write modified char back to string in memory
        B       Loop          ;branch back to Loop
Done    LDMFD   SP!,{r0,r1}   ;restored saved registers before returning
        MOV     PC,LR         ;return from subroutine
```

Always

Z=1

Note: Subroutine modifies **R0** & **R1** but restores their original contents before returning. This produces a **transparent subroutine** that does not effect the proper operation of calling program

39

39

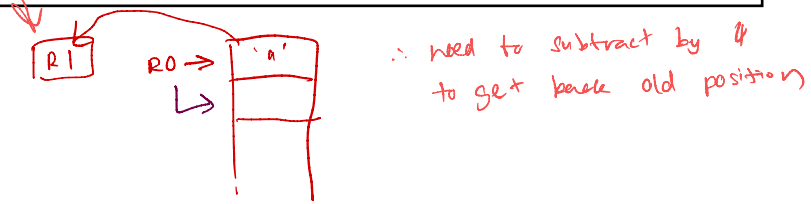
Possible Solution

```
Lo2Up   STMFD   SP!,{r0,r1}   ;save registers used within subroutine
Loop    LDR     R1,[R0],#4    ;get current char from string in memory
        Increment R0 by 4 and update R0
        (to point to next memory location)
        Refer to the location to R0-4 (the original R0)
        STR     R1,[R0,#-4]   ;write modified char back to string in memory
        B       Loop          ;branch back to Loop
Done    LDMFD   SP!,{r0,r1}   ;restored saved registers before returning
        MOV     PC,LR         ;return from subroutine
```

Note: Subroutine modifies **R0** & **R1** but restores their original contents before returning. This produces a **transparent subroutine** that does not effect the proper operation of calling program

40

40



Possible Solution

```
Lo2Up   STMFD   SP!,{r0,r1}   ;save registers used within subroutine
                                  → R0=0x100
Loop    LDR     R1,[R0],#4    ;get current char from string in memory
        Increment R0 by 4 and update R0
        (to point to next memory location)
                                  → R0=0x104
                                  → R0-4=0x100
        Refer to the location to R0-4 (the original R0)
        STR     R1,[R0,#-4]   ;write modified char back to string in memory
        B       Loop          ;branch back to Loop
Done    LDMFD   SP!,{r0,r1}   ;restored saved registers before returning
        MOV     PC,LR         ;return from subroutine
```

Note: Subroutine modifies **R0** & **R1** but restores their original contents before returning. This produces a **transparent subroutine** that does not effect the proper operation of calling program

41

41

Optimizing Memory Usage

- Example assumes each character in 32 bits
  - But each character requires 8 bits only
- Can we access each Byte separately?

Address	Memory
0x100	"a"
0x101	"p"
0x102	"p"
0x103	"l"
0x104	"e"
0x105	0x000
0x106	:

42

42

Optimizing Memory Usage

- Example assumes each character in 32 bits
  - But each character requires 8 bits only
- Can we access each Byte separately?

Use the {B} option in Access

LDRB
STRB

Address	Memory
0x100	"a"
0x101	"p"
0x102	"p"
0x103	"l"
0x104	"e"
0x105	0x000
0x106	:

43

43

Efficient Memory Solution

```
Lo2Up   STMFD   SP!,{r0,r1}   ;save registers used within subroutine
Loop    LDRB    R1,[R0],#1    ;get current char from string in memory
        CMP     R1,#0         ;Compare with NULL
        BEQ     Done          ;if NULL char, branch to Done
        CMP     R1,#0x061     ;compare with lower limit "a"
        BLT     Loop          ;if smaller than "a", do not convert
        CMP     R1,#0x07A     ;compare with upper limit "z"
        BGT     Loop          ;if greater than "z" do not convert
        SUB     R1,R1,#32     ;convert to upper case by subtracting 32
        STRB    R1,[R0,#-1]   ;write modified char back to string in memory
        B       Loop          ;branch back to Loop
Done    LDMFD   SP!,{r0,r1}   ;restored saved registers before returning
        MOV     PC,LR         ;return from subroutine
```

gets byte from mem puts in least 8 bits of register then extend by 0s

just puts in the right location w/o altering anything

Note: Subroutine modifies **R0** & **R1** but restores their original contents before returning. This produces a **transparent subroutine** that does not effect the proper operation of calling program

44

44

Summary

- `BL` is required to call a subroutine, return address is in `LR` register
- A return from subroutine is done with `BX LR` or `MOV PC,LR`
- Passing parameters **using registers** is the simplest and fastest.
  - Number of parameters that can be passed is **limited** by the **available registers**.
- Passing parameters using a **memory block** can support a **large number** of parameters or data types like arrays.

45

Review Question

Q1 – CALL

Qns: What registers may be affected by executing:

BL RegR1

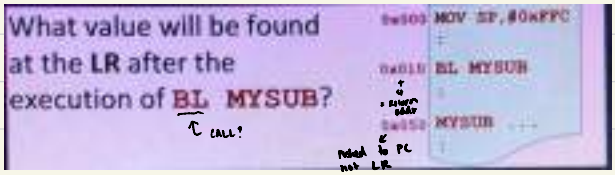
Ans: PC & LR

Take location of instruction "BL RegR1" + 4 store into Link Register (LR)

Then take RegR1 make a copy of the above memory location and put into PC

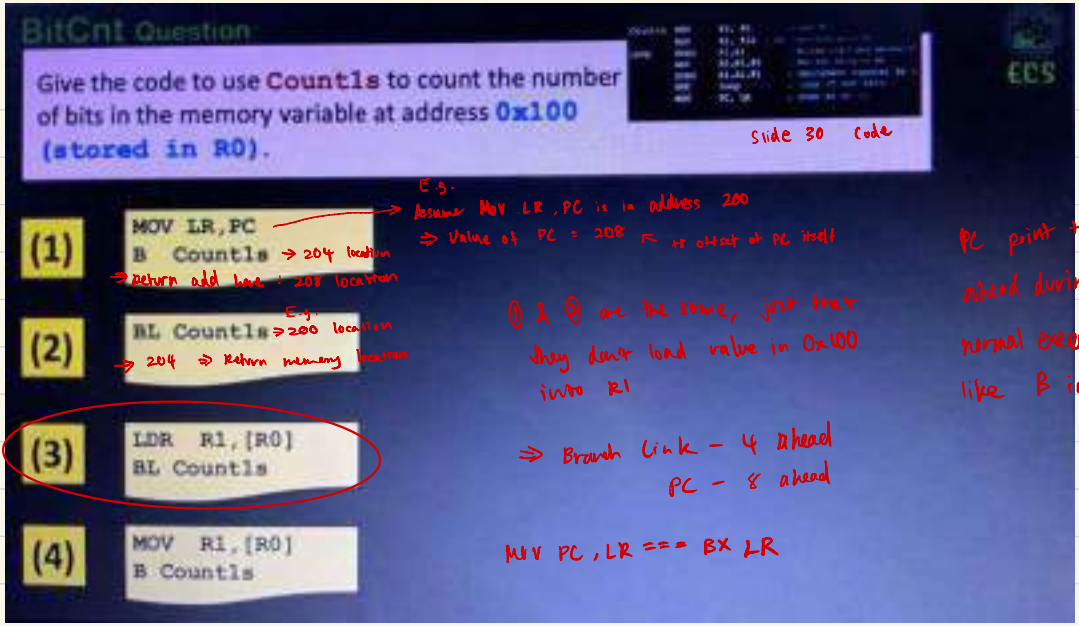
Stack Pointer not affected by BL instruction in this case

Q2 - CALL



Ans : 0x014

* Every instruction a fixed 4 bytes.



(1) "MOV LR, PC" places PC (address of MOV+8) to LR, followed by a branch

⇒ Actualy the correct return address however, value not loaded to R1 prior to subroutine

⇒ MOV LR,PC & Countls = BL Countls

(2) Have to prepare the parameter first before calling the countls subroutine

Countls requires the parameter to be passed via the R1 register

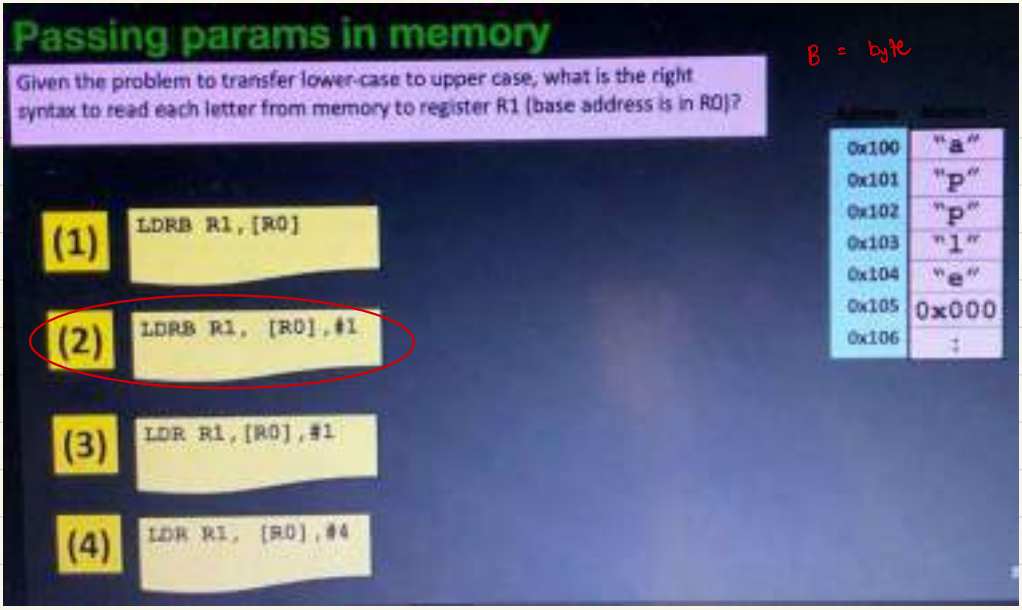
(3)

1. Set up parameter into R1 register
2. Call Countls subroutine with BL instruction using the appropriate subroutine label

(3) MOV an invalid instruction; cannot be used to read from memory

Operands can be only

* (1) if another instruction was placed between MOV & branch, it would loop infinitely



(1) Byte is read but does not update R0 for the next instruction

⇒ increment R0 in another instruction

E.g. ADD R0, R0, #1

(2) instruction can read a byte, and increments R0 with 1 after fetching

⇒ by auto indexing

(3) instruction can read a 32-bit word, and increments R0 with 1 after fetching

"#1" will cause an error ⇒ runtime cause trying to get unaligned word

"0x100" → "0x101" ✗ "0x104" allowed

(4) instruction can read a 32-bit word, and increments R0 with 4 after fetching

"#4" will not cause an error

Chapter 6: Modular Programming-Cont

Mohamed M. Sabry Aly
N4-02c-92

1

Learning Objectives

- List the stack manipulation operations & its implementation
- Describe passing parameters to subroutines using the stack
- Identify the difference between passing by value and by reference.
- Describe how a transparent subroutine can be implemented.

2

R0 - R3 : Manipulitive registers

R4 - R11 : local variables

↳ if callee wants to modify, need to create a

R12 : scratch pad

System Stack

- A stack is a first-in, last-out linear data structure that is maintained in the memory's data area.

Still can read all data

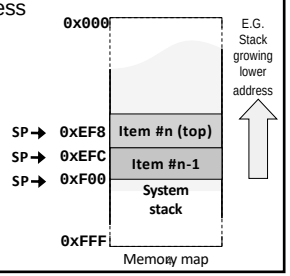
first do → put inside, last thing to take out

Stack of warm plates on dispenser

3

System Stack

- A stack is a first-in, last-out linear data structure that is maintained in the memory's data area.
  - The system stack in the ARM processor is maintained by a dedicated stack pointer (**SP, R13**).
  - Stack can grow towards lower or higher memory address
    ! Preferred direction is lower, start from max address ! Why?
  - The **SP** points to the top item on the system stack.
    why not zeroed.
    ⇒ points to valid data, NOT free space
  - The 3 basic stack operations are **push**, **pop** and **access** items on the stack.



Ref: Null & Lobur (3rd Ed) section A.2.3 – Stacks

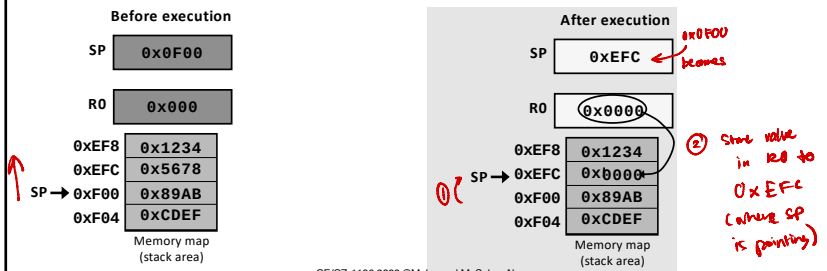
4

① Update value at location one step above (lower address) that the SP is pointing at

② Make SP point to the newly added location

Push Data to the Stack

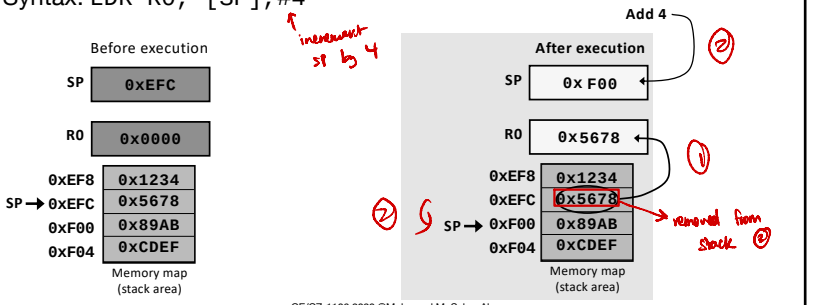
- Writing to stack can be done using **STR** instruction
  - Need to also increase the stack pointer before storing
- Syntax: `STR R0, [SP,#-4]!` ← pre-indexing



5

Pop Data off the Stack

- Popping from the stack can be done with **LDR** instruction
  - Need to update pointer AFTER read
- Syntax: `LDR R0, [SP],#4`

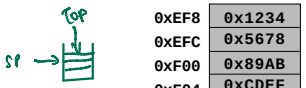


6

Removing from stack

- After popping, is the data **erased** from the stack?

NO



items considered "in the stack" as long as it's underneath the stack pointer

only does not exist in data structure of stack

how to retrieve back

- Data still resides in memory and can be read ⇒ as long as not overwritten by another value
  `LDR R0, [SP, #-4]` SP is not changed

⇒ to erase value memory, can just update with 0

7

Pushing Registers to Stack

- E.g., want to push R0 and R1

Method 1:

```
STR R0, [SP, #-4]!
STR R1, [SP, #-4]!
```

Any value to be stored into the stack should be placed into a register first

⇒ cannot push constant / number directly into stack

⇒ get the constant, push into register, push register into stack

8

E.g. STMFD R3, R1
STMFD R7, R5

Result:

R5
R7
R1
R3

2 separate instructions
What happens to LDMFD then?

Pushing Registers to Stack

- E.g., want to push R0 and R1

Method 1:

```
STR  R0, [SP, #-4]!
STR  R1, [SP, #-4]!
```

Before Execution

SP	0xF00

Address	Value
0xEF8	0x1234
0xEFC	0x5678
SP → 0xF00	0x89AB
0xF04	0xCDEF

Memory map (stack area)

Method 2:

```
STMFD  SP!,{R1,R0}
```

Use Stack pointer and write back updated value after operation

Write registers in descending order

Store multiple registers fully descending

sequence doesn't matter here, will still store in descending order

list registers to store

Descending related to how to update SP

(STMFA : Ascending)

After Execution

SP	0xEF8

Address	Value
SP → 0xEF8	R0
0xEFC	R1
0xF00	0x89AB
0xF04	0xCDEF

Memory map (stack area)

read this direction ↓ R1, R5, R4

CE/CZ-1106 2020 ©Mohamed M. Sabry Aly 9

9

Pushing and popping to stack

e.g. R1, R5, R4

```
STMFD SP!, {list of registers}
LDMFD SP!, {list of registers}
```

↳ Load multiple

```
STMFD SP!,{R1,R0}
```
→ decremented by 8

```
LDMFD SP!,{R1,R0}
```
→ increase by 8 steps

→ assumes that you spread in a decending way, hence will read from acending way

read downwards

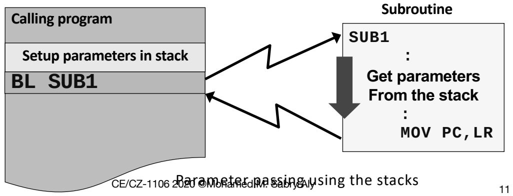
CE/CZ-1106 2020 ©Mohamed M. Sabry Aly 10

10

between STMFD & LDMFD, ensure code returns SP back to where STMFD was
⇒ same set of registers to both instruction

Parameter Passing using Stack

- Parameters are pushed onto the stack before calling the subroutine and retrieved from the stack within the subroutine.
- Most **general** method of parameter passing – no registers needed, supports recursive programming.
- **Large** numbers of parameters can be passed as long as stack does not overflow.

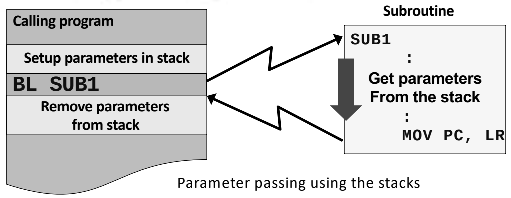


Parameter passing using the stacks

CE/CZ-1106 2020 ©Mohamed M. Sabry Aly 11

11

Parameter Passing using Stack



Parameter passing using the stacks

- Parameters pushed to the stack must be **removed** by the calling program immediately after returning from subroutine. → cause infinite loop
- If not, repeated pushing of parameters to the stack will lead to a stack overflow.

removal of parameters DOES NOT happen in subroutine
⇒ it has to happen in caller program

CE/CZ-1106 2020 ©Mohamed M. Sabry Aly 12

12

Sum from 1 to N

• Write a subroutine to:

- Sum the positive numbers from 1 to **N**, where **N** is a value passed to the subroutine.
- The computed sum should be directly updated to a memory variable **Answer**, whose address is **0x100**.
- All parameters are to be passed via the **stack**.

Solution:

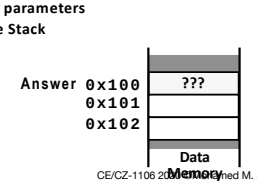
- Push two parameters on stack, the value of **N** and **address** of memory variable **Answer**.

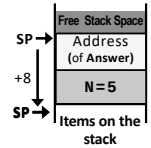
13

Calling the Subroutine

```
        :                          ;find the sum of (1+2+3+4+5), where N=5
Parameters setup
        MOV   R1, #5               ;Set R0 with #5
        MOV   R0, #0x100           ;Set R1 with the address
        STMFD SP!, {r1,r0}         ;push R0&R1 to stack
        BL    Sum1N                ;call subroutine Sum1N
        ADD   SP,SP,#8             ;add 8 to pop the two parameters from stack
        :
```

Remove parameters from the Stack





14

Calling the Subroutine

```
        :                          ;find the sum of (1+2+3+4+5), where N=5
Parameters setup
        MOV   R1, #5               ;Set R0 with #5
        MOV   R0, #0x100           ;Set R1 with the address
        STMFD SP!, {R1,R0}         ;push R0&R1 to stack
        BL    Sum1N                ;call subroutine Sum1N
        ADD   SP,SP,#8             ;add 8 to pop the two parameters from stack
        :
```

Note:
1. Parameters set up before calling the subroutine are **removed** immediately after returning from subroutine.
2. Removal is done by returning the contents of the stack pointer to its **original value** before the parameters were pushed to the stack.

15

Sum from 1 to N Subroutine
Possible Solution

R6 Answer Addr
R5 N

```
Sum1N   STMFD  SP!,{R4,R5,R6}    ;save registers to stack
Retrieve stack parameters
        LDR    R5,[SP,#16]       ;Load N from stack to R5
        LDR    R6,[SP,#12]       ;Load Answer's From stack
        MOV    R4,#0             ;clear summation register R0 to 0
Loop    ADD    R4,R4,R5          ;add current value in R5 to R4
        SUBS   R5,R5,#1          ;decrement current value in R4 by 1
Z=0     BNE    Loop              ;jump back to Loop if R4 not yet zero
Z=1     STR    R4,[R6]           ;write sum to Answer
        LDMFD  SP!,{R4,R5,R6}    ;restored saved registers
        MOV    PC,LR             ;return from subroutine
```

Note: The subroutine needs three registers. **R4** to compute the sum from 1 to N. **R6** to be an address pointer to the memory variable **Answer** where the results will written to. **R5** holds the value of N, which is decremented by 1 after each loop till it reaches 0.

16

Explanation:

Why 16? Why 12? ↓

Sum from 1 to N Subroutine
Accessing Stack Parameters

Calling program

```
:
STMFD   SP!,{r1,r0}
BL      Sum1N
```

Subroutine Sum1N

```
STMFD  SP!,{R4-R6}
LDR    R5,[SP,#16]
LDR    R6,[SP,#12]
```

Offset from SP

After 3 SP →	SP+0	Saved R4
	SP+4	Saved R5
	SP+8	Saved R6
After 1 SP →	SP+12	Address (of Answer)
	SP+16	N=5
Before 1 SP →	SP+20	

Displacements for stack parameters

Memory map (stack area)

Note: SP indirect with offset can be used to access parameters on the stack but knowledge of all items on the stack is needed to compute the correct offset from the current **SP** position.

17

17

Transparent Subroutines

- A transparent subroutine will **not affect any CPU resources** used by the program calling it.
- To achieve this, all local registers used by the subroutine (R4-R11) must be **saved on the stack** on entry and **restored from stack** before returning.

```
SUB1 STMFD SP!,{R4-R7}     ; save R4 to R7 to stack
     :
     :                     ; registers R4 to R7 are
     :                     ; used in subroutine
     LDMFD SP!,{R4-R7}     ; restore R4 to R7 from stack
     MOV PC,LR             ; return to calling program
```

18

18

Passing by value and by reference

- Parameters are passed to subroutines in 2 ways:
  - **Pass by value** – the value of the data (or variable) is passed to the subroutine.
    - e.g. use of integer
  - **Pass by reference** – the address of the variable is passed to the subroutine .
    - e.g. use of pointer
- When is passing by reference used?
  - When the **parameter** passed is to be **modified** by the subroutine.
  - **Large quantity** of data (e.g. array) have to be passed between subroutine and calling program.

Calling program

```
MOV    R1,#5
MOV    R0,#0x100
STMFD  SP!,{R1,R0}
BL     Sum1N
```


19

19

C Function Example

- **C** function to compute the mean value of **N** elements in an integer **Array**.

Passing by reference

Passing by value

```
int Mean (int *Array, int N)
{
        int avg, i;
        int sum = 0;
        for (i=0; i<N; i++)
          sum = sum + Array[i];
        avg = sum / N;
        return avg;
}
```

Local variables used only by the function

Function's single output is normally returned via the R0 (or R12) register

Note: Observe that **local variables** are required by the function to compute the result.

20

20

C Function Example

- **C** function to compute the mean value of **N** elements in an integer **Array**.
- Pass both parameters **by reference**.

```
int Mean (int *Array, int *N)
{
        int avg, i;
        int sum = 0;
        for (i=0; i<*N; i++)
          sum = sum + Array[i];
        avg = sum / *N;
        return avg;
}
```

whatever pointing is referring

Note: Observe that **local variables** are required by the function to compute the result.

21

Local Variables

- Subroutines often use local variables whose scope and **life span** exist only during the execution of the subroutine.

Review

- **Characteristics of a good software module.**
- Loose coupling – **data** within module is entirely **independent** of other modules (local variables).

22

Local Variables

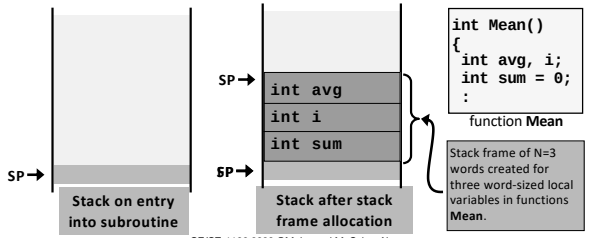
- Subroutines often use local variables whose scope and **life span** exist only during the execution of the subroutine.
  - Memory storage for these variables is **created on entry** into the subroutine and **released on exit** from subroutine.
  - The **system stack** is the ideal place to create memory space for temporary variables.
  - This temporary memory space is called the **stack frame**.

is subset of Stack
⇒ has variables to be used in subroutine

23

Stack Frame

- Stack frame of **N** words is created on the system stack immediately on entry into a subroutine.
- Memory space within the stack frame can be accessed using with a **frame pointer** (**FP**) or the stack pointer (**SP**).
- Frame pointer in ARM can be any register
  - General convention **FP** is **R11**



i, avg, sum are the local variables

24

each functional call — frame pointer is a register

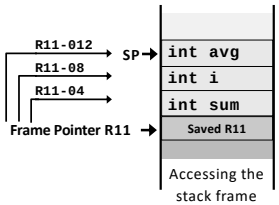
only 1 frame pointer

⇒ 1 function → 1 frame pointer — each word → 4 bytes

Accessing Stack Frame Variables

Using the Frame Pointer

- Consider the use of a frame pointer register **R11**.
  - Original contents of **R11** is saved on the stack before it is used as the frame pointer.
  - Frame pointer (**R11**) now points to the saved **R11** and a stack frame is created by adding frame size **4N** to **SP**.
  - **R11** is the frame reference and an appropriate negative displacement from **R11** can be used to access any stack frame variable.
  - When exiting the subroutine, the stack frame can be destroyed by adding **4N+4** to **SP** and copying the saved **R11** value on the stack back into **R11**.



25

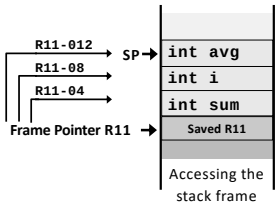
Accessing Stack Frame Variables

Using the Frame Pointer

- When exiting the subroutine, the stack frame can be destroyed by adding **4N+4** to **SP** and copying the saved **R11** value on the stack back into **R11**.

For N=3

```
ADD SP, SP, #16
LDR R11,[R11]
```

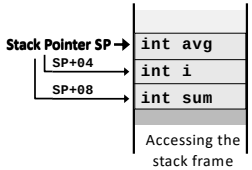


26

Accessing Stack Frame Variables

Using the Stack Pointer

- A more efficient approach is to use the stack pointer (**SP**) itself.
  - A stack frame is created by adding frame size **4N** to **SP**.
  - **SP** is used as a reference to access all local variables.
  - Appropriate **positive** displacements from **SP** is used to access any of the stack frame variables.
  - **Pro** - This method is more efficient because there is no need to setup a frame pointer.
  - **Con** – More restrictive as system stack cannot be used within subroutine without changing the reference **SP**.



28

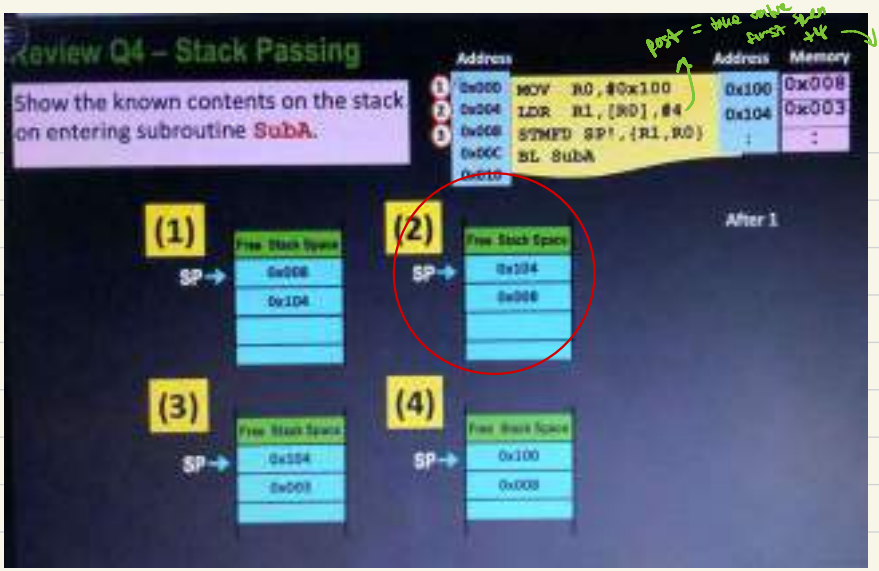
More efficient cause using less resources

However, if new variable created, might have issue

Summary

- Using the **stack** is the most favored means of passing parameters.
  - A combination of methods can be used to implement **Functions**, e.g. parameters passed in via stack and a single result value passed out via a register (e.g. **R0**).
  - Stack-based parameter passing supports **recursion**.
- Parameters is passed by **value** or **reference**.
  - Passing by reference allows the subroutine to directly access memory variables within the calling program .
- A **transparent** subroutine requires registers **used within** the routine to be saved on the **stack** on entry and restore before returning.
- Local variables within a subroutine are usually maintained on the system stack using a **stack frame**.

29



After 1

R0	0x100
R1	

After 2

R0	0x104
R1	0x008

LDR R1,[R0],#4

1) go to address in R0 (0x100), take the value (0x008) and put into R1
2) increment R0 by 4 (0x100 → 0x104)

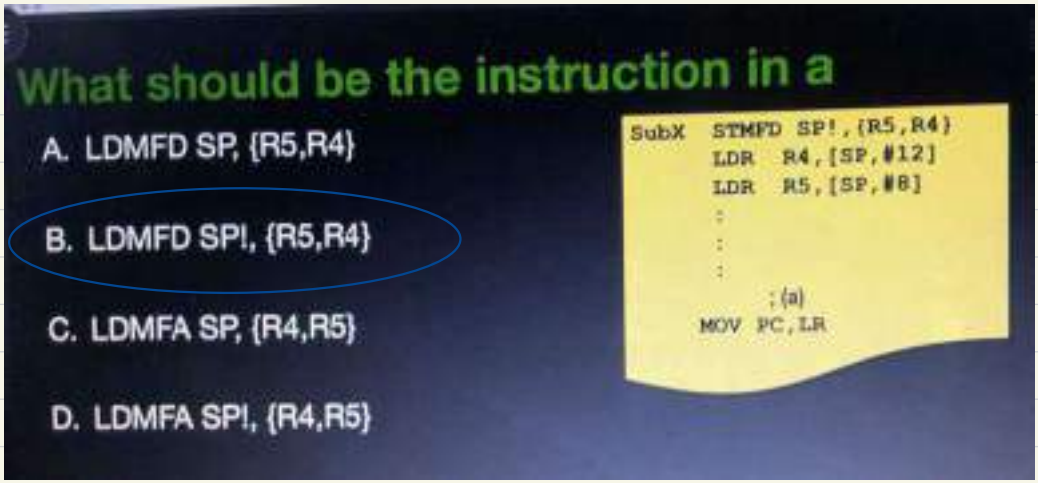
After 3

R0	0x104
R1	0x008

↓ ↓ put into stack

R0	0x104
R1	0x008

Clock cycles of STMFD
⇒ For 1 register, 3 cycles.
⇒ +1 cycle for each additional register



C & D : Since STMFD was used. C & D not valid here

A : does not update stack pointer

Chapter 6: Modular Programming-Cont

Mohamed M. Sabry Aly
N4-02c-92

1

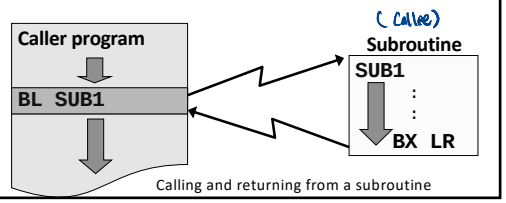
Learning Objectives

- Understand the concept of nested subroutine
- Describe how nested subroutines are implemented in ARM
- Describe recursive functions and their implementation in ARM

2

Subroutines (Recap)

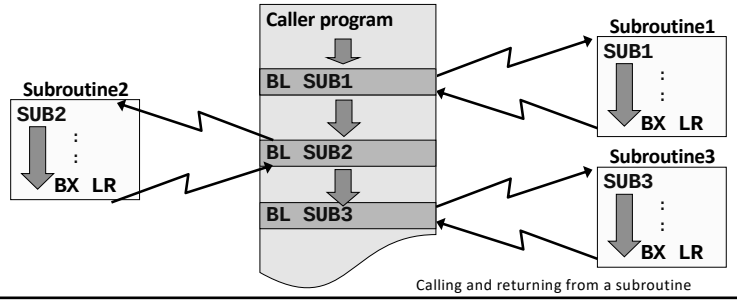
- Modules (e.g., functions in C) are implemented as **subroutines**
- Subroutine can be called from various parts of the program
- Caller and callee
  - Caller: the program that calls subroutine (SUB1)
  - Callee: subroutine (SUB1)



3

Multiple Subroutines

- Caller program can call multiple subroutines
- Support of sequential subroutine execution



4

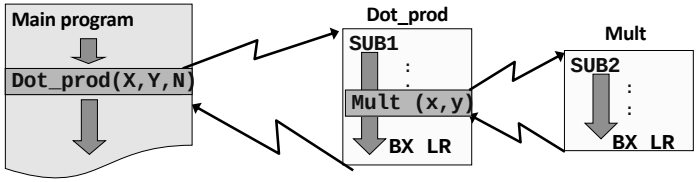
Do all applications have just 1-level functional call?

No

5

Example: Dot product of two arrays

- A subroutine will call another subroutine

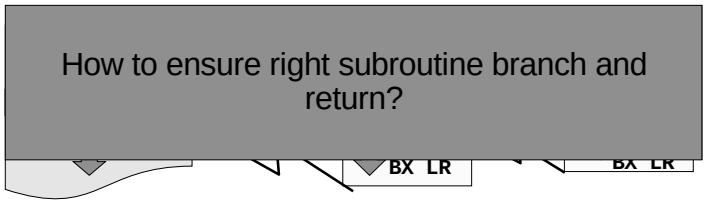


6

Example: Dot product of two arrays

- A subroutine will call another subroutine

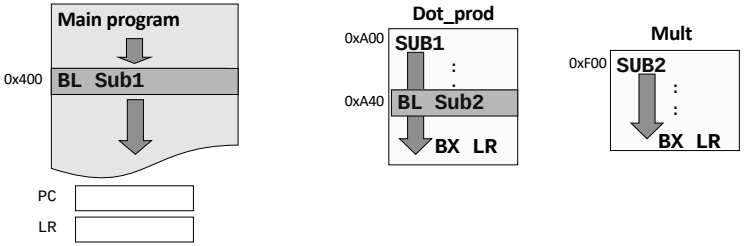
How to ensure right subroutine branch and return?



7

Example: Dot product of two arrays

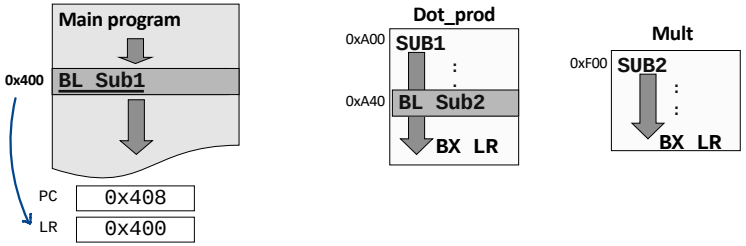
- A subroutine will call another subroutine



8

Example: Dot product of two arrays

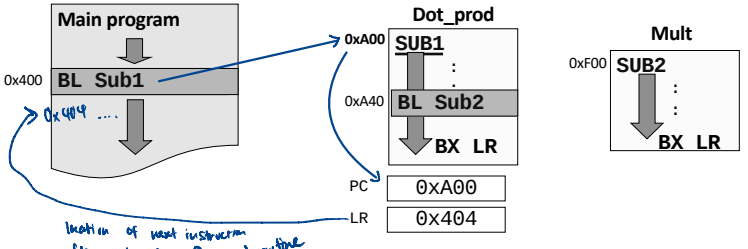
- A subroutine will call another subroutine



9

Example: Dot product of two arrays

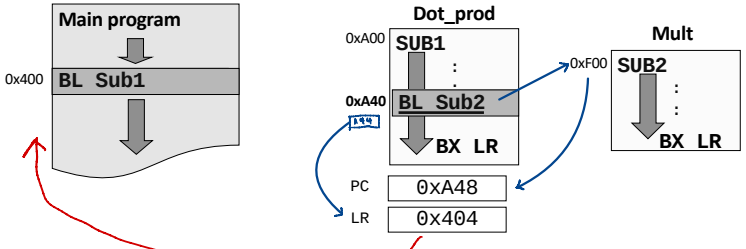
- A subroutine will call another subroutine



10

Example: Dot product of two arrays

- A subroutine will call another subroutine

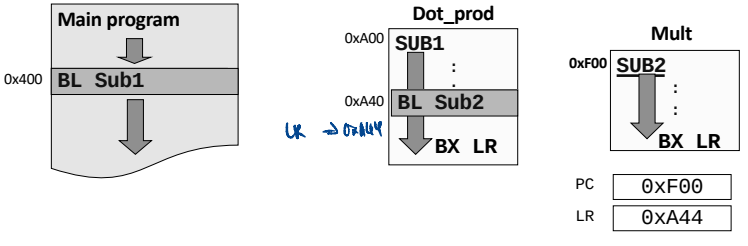


11

Was the link to Main program

Example: Dot product of two arrays

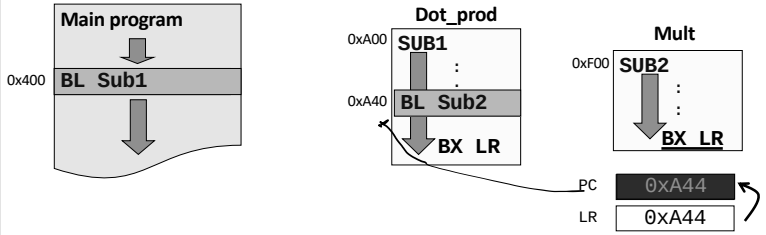
- A subroutine will call another subroutine



12

Example: Dot product of two arrays

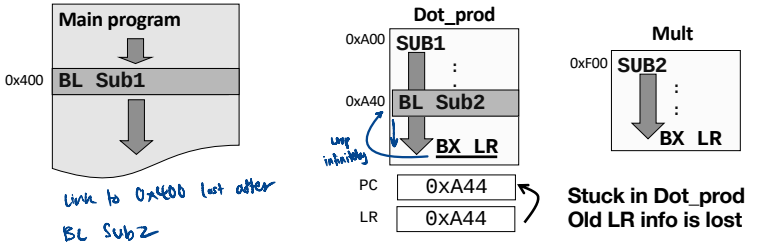
- A subroutine will call another subroutine



13

Example: Dot product of two arrays

- A subroutine will call another subroutine



14

Support for nested Subroutine

- LR needs to be saved somewhere safe → **System stack**
- When so save it? Two options:
  - Just before any BL instruction →
  - At the beginning of each subroutine →

15

Example: Dot-product

- Write a subroutine that calculates the dot product of two vectors of **positive integers** using this equation:

$$Z = \sum_{i=0}^{N-1} X_i \times Y_i$$

- Bass addresses of X (0x100), Y(0x200), the array size N(20) and Z(0x300) are passed through the stack
- The subroutine will call another subroutine to calculate the product of two numbers:
  - Numbers are passed in R0,R1, return value in R12

16

Multiplication subroutine

```
Mult    MOV     R12,#0      ;Clear R12
```

Start by labeling the subroutine
and placing the return instruction
Clearing (R2)

```
        MOV     PC, LR      ; same as bx lr
```


17

Multiplication subroutine

Iteratively add R0 to R12 for R1 times

```
Mult    MOV     R12,#0      ;Clear R12
Loop    ADD     R12,R12,R0  ;Add R0 to R12
        SUBS    R1,R1,#1    ;Decrement R1 with 1
        BNE     Loop
        MOV     PC, LR      ; same as bx lr
```

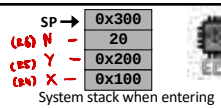
if R1 ≠ 0

R1 = 0

R12

18

The Dot product Subroutine



System stack when entering DotProd

4 regs

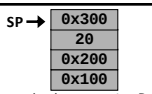
```
DotProd STMFD   SP!,{R4-R7} ;Store regs to stack
        LDR     R4 ,[SP,#28] ;Read location of X
        LDR     R5 ,[SP,#24] ;Read location of Y
        LDR     R6 ,[SP,#20] ;Read arrays length


        LDMFD   SP!,{R4-R7} ; Restore registers
        MOV     PC, LR      ; same as bx lr
```


19

register 1 – store addresses of x
2 – store addresses of y
3 – counter
4 – address of destination location

R7 will contain partial sum

The Dot product Subroutine



System stack when entering DotProd

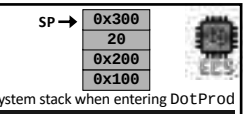
```
DotProd STMFD   SP!,{R4-R7} ;Store regs to stack
        LDR     R4 ,[SP,#28] ;Read location of X
        LDR     R5 ,[SP,#24] ;Read location of Y
        LDR     R6 ,[SP,#20] ;Read arrays length
        MOV     R7, #0      ; Clear R7
Loop1   LDR     R0 ,[R4],#4 ; Get X[i]
        LDR     R1 ,[R5],#4 ; Get Y[i]

        BL      Mult        ; Call Mult Subroutine

        LDMFD   SP!,{R4-R7} ; Restore registers
        MOV     PC, LR      ; same as bx lr
```

update R4 with next pointer

e.g. 104

20

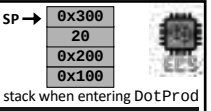
The Dot product Subroutine




```
DotProd     STMFD       SP!,{R4-R7}     ;Store regs to stack
            LDR         R4 ,[SP,#28]    ;Read location of X
            LDR         R5 ,[SP,#24]    ;Read location of Y
            LDR         R6 ,[SP,#20]    ;Read arrays length
            MOV         R7, #0          ; Clear R7 (Sum)
Loop1       LDR         R0 ,[R4],#4     ; Get X[i]
            LDR         R1 ,[R5],#4     ; Get Y[i]

            BL          Mult

            ADD         R7,R7,R12

            LDMFD       SP!,{R4-R7}     ; Restore registers
            MOV         PC, LR          ; same as bx lr
```

21

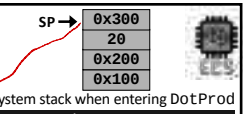
The Dot product Subroutine




```
DotProd     STMFD       SP!,{R4-R7}     ;Store regs to stack
            LDR         R4 ,[SP,#28]    ;Read location of X
            LDR         R5 ,[SP,#24]    ;Read location of Y
            LDR         R6 ,[SP,#20]    ;Read arrays length
            MOV         R7, #0          ; Clear R7 (Sum)
Loop1       LDR         R0 ,[R4],#4     ; Get X[i]
            LDR         R1 ,[R5],#4     ; Get Y[i]

            BL          Mult            ; Call Mult Subroutine

            ADD         R7,R7,R12       ; Add the product to R7
            SUBS        R6,R6,#1        ; Reduce the counter by 1
            BNE         Loop1           ; not 0 then repeat

            LDMFD       SP!,{R4-R7}     ; Restore registers
            MOV         PC, LR          ; same as bx lr
```

22

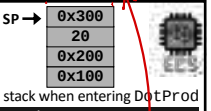
The Dot product Subroutine




```
DotProd     STMFD       SP!,{R4-R7}     ;Store regs to stack
            LDR         R4 ,[SP,#28]    ;Read location of X
            LDR         R5 ,[SP,#24]    ;Read location of Y
            LDR         R6 ,[SP,#20]    ;Read arrays length
            MOV         R7, #0          ; Clear R7 (Sum)
Loop1       LDR         R0 ,[R4],#4     ; Get X[i]
            LDR         R1 ,[R5],#4     ; Get Y[i]

            BL          Mult            ; Call Mult Subroutine

            ADD         R7,R7,R12       ; Add the product to R7
            SUBS        R6,R6,#1        ; Reduce the counter by 1
            BNE         Loop1           ; not 0 then repeat
            LDR         R4, [SP,#16]    ;     read destination address
            STR         R7, [R4]        ; Store in destination address
            LDMFD       SP!,{R4-R7}     ; Restore registers
            MOV         PC, LR          ;     same as bx lr
```

23

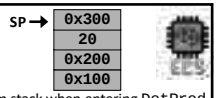
The Dot product Subroutine




```
DotProd     STMFD       SP!,{R4-R7}     ;Store regs to stack
            LDR         R4 ,[SP,#28]    ;Read location of X
            LDR         R5 ,[SP,#24]    ;Read location of Y
            LDR         R6 ,[SP,#20]    ;Read arrays length
            MOV         R7, #0          ; Clear R7 (Sum)
Loop1       LDR         R0 ,[R4],#4     ; Get X[i]
            LDR         R1 ,[R5],#4     ; Get Y[i]
            STR         LR ,[SP,#-4]!   ; Push Link register to SP
            BL          Mult            ; Call Mult Subroutine
            LDR         LR ,[SP],#4     ; Pop Link Register from SP
            ADD         R7,R7,R12       ; Add the product to R7
            SUBS        R6,R6,#1        ; Reduce the counter by 1
            BNE         Loop1           ; not 0 then repeat
            LDR         R4, [SP,#16]    ; read destination address
            STR         R7, [R4]        ; Store in destination address
            LDMFD       SP!,{R4-R7}     ; Restore registers
            MOV         PC, LR          ; same as bx lr
```

24

The Dot product Subroutine



System stack when entering DotProd

✓ Straight forward, no other impact on the code

Alternative: STMFD , SP! , {LR}

```
STR     LR ,[SP,#-4]!     ; Push Link register to SP
BL      Mult              ; Call Mult Subroutine
LDR     LR ,[SP],#4       ; Pop Link Register from SP
```

Alternative: LDMFD

✗ Different instruction structure

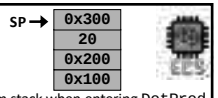
25

Support for nested Subroutine

- LR needs to be saved somewhere safe → **System stack**
- When so save it? Two options:
  - Just before any BL instruction
  - At the beginning of each subroutine

26

The Dot product Subroutine



System stack when entering DotProd

```
DotProd   STMFD   SP!,{R4-R7,LR}   ;Store regs to stack
          LDR     R4 ,[SP,#28]     ;Read location of X
          LDR     R5 ,[SP,#24]     ;Read location of Y
          LDR     R6 ,[SP,#20]     ;Read arrays length
          MOV     R7, #0           ; Clear R7 (Sum)
Loop1     LDR     R0 ,[R4],#4      ; Get X[i]
          LDR     R1 ,[R5],#4      ; Get Y[i]
          STMFD   SP! ,{LR}
          BL      Mult             ; Call Mult Subroutine
          LDMFD   SP!, {LR}
          ADD     R7,R7,R12        ; Add the product to R7
          SUBS    R6,R6,#1         ; Reduce the counter by 1
          BNE     Loop1            ; not 0 then repeat
          LDR     R4, [SP,#16]     ; read destination address
          STR     R7, [R4]         ; Store in destination address
          LDMFD   SP!,{R4-R7,LR}   ; Restore registers
          MOV     PC, LR           ; same as bx lr
```

if LR not at the start code

27

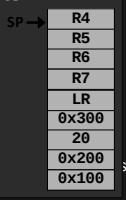
Simpler & faster code

The Dot product Subroutine

System stack when entering DotProd

```
DotProd   STMFD   SP!,{R4-R7,LR}   ;Store regs to stack
          LDR     R4 ,[SP,#28]     ;R
          LDR     R5 ,[SP,#24]     ;R
          LDR     R6 ,[SP,#20]     ;R
          MOV     R7, #0           ;
Loop1     LDR     R0 ,[R4],#4      ;
          LDR     R1 ,[R5],#4      ;
          BL      Mult             ;
          ADD     R7,R7,R12        ;
          SUBS    R6,R6,#1         ;
          BNE     Loop1            ;
          LDR     R4, [SP,#16]     ;
          STR     R7, [R4]         ;
          LDMFD   SP!,{R4-R7,LR}   ;
          MOV     PC, LR           ; same as bx lr
```

We now push 5 registers to stack
Need to update the offsets



28

Purpose of restoring register at the top: to not lose the initial values it might have after using the registers

The Dot product Subroutine

System stack when entering DotProd

```
DotProd   STMFD   SP!,{R4-R7,LR}   ;Store regs to stack
          LDR     R4 ,[SP,#32]     ;R
          LDR     R5 ,[SP,#28]     ;R
          LDR     R6 ,[SP,#24]     ;R
          MOV     R7, #0           ;
Loop1     LDR     R0 ,[R4],#4      ;
          LDR     R1 ,[R5],#4      ;

          BL      Mult             ;

          ADD     R7,R7,R12        ;
          SUBS    R6,R6,#1         ;
          BNE     Loop1            ;
          LDR     R4, [SP,#20]     ;
          STR     R7, [R4]         ;
          LDMFD   SP!,{R4-R7,LR}   ;
          MOV     PC, LR           ; same as bx lr
```

We now push 5 registers to stack
Need to update the offsets

29

The Dot product Subroutine

System stack when entering DotProd

```
DotProd   STMFD   SP!,{R4-R7,LR}   ;Store regs to stack
          LDR     R4 ,[SP,#32]     ;R
          LDR     R5 ,[SP,#28]     ;R
          LDR     R6 ,[SP,#24]     ;R
          MOV     R7, #0           ;
Loop1     LDR     R0 ,[R4],#4      ;
          LDR     R1 ,[R5],#4      ;

          BL      Mult             ;

          ADD     R7,R7,R12        ;
          SUBS    R6,R6,#1         ;
          BNE     Loop1            ;
          LDR     R4, [SP,#20]     ;
          STR     R7, [R4]         ;
          LDMFD   SP!,{R4-R7,PC}   ;
```

Why is this correct?

We now push 5 registers to stack
Need to update the offsets

30 Not recommended though ; for optimization purposes

Recursive Subroutine

- A recursive routine calls itself within its own body.
- Recursion is an elegant way to solve algorithms or mathematical expressions that have **systematic repetitions**. (e.g. factorial **n! = n×(n-1)×(n-2)….2×1**)

```
int factorial(int n){
    if (n<0)
     return -1; //sanity check
    if (n==0)
     return 1;
    else
     return n * factorial(n-1);
}
```

stop condition

31

31

Recursive Subroutine

- A recursive routine calls itself within its own body.
- Recursion is an elegant way to solve algorithms or mathematical expressions that have **systematic repetitions**. (e.g. factorial **n! = n×(n-1)×(n-2)….2×1**)

Main problem: Without saving LR, execution will be stuck

Recursive Code Example

```
Recur :
      BL  Recur    ; call Recur with Recur routine
      :
      BX  LR       ; return to calling program
```

; Subroutine Recur

Another problem: Some condition must be reached that allows **BL Recur** to be skipped in order to avoid infinite recursion.

need a stopping condition

32

32

not go appear in exam

Recursive Subroutine

- A recursive routine calls itself within its own body.
- Recursion is an elegant way to solve algorithms or mathematical expressions that have **systematic repetitions**. (e.g. factorial **n! = n×(n-1)×(n-2)….2×1**)

Recursive Code Example

```
Recur   STMFD SP!,{…,LR}        ; Subroutine Recur
        Condition to terminate the subroutine (branch to Done)
        :
        BL Recur                ; call Recur with Recur routine
        :
Done    LDMFD SP!,{…,LR}
        BX LR                   ; return to calling program
```

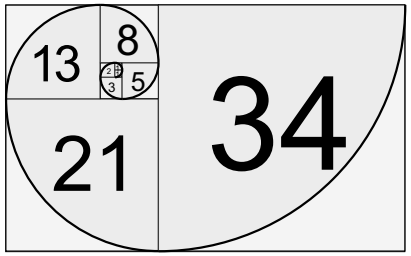
33

33

Example: Fibonacci Sequence

- Number sequence
  0,1,1,2,3,5,8,13,21,34,55
- Starting from index 1
  $F(n) = F(n-1) + F(n-2)$
  $F(2) = 1$
  $F(1) = 0$

  Pass parameter in stack
  Return result in R12



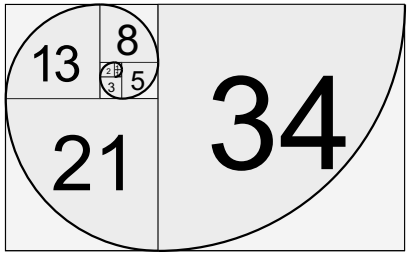
CE/CZ-1106 2020 ©Mohamed M. Sabry Aly 34

34

Example: Fibonacci Sequence

- Number sequence
  0,1,1,2,3,5,8,13,21,34,55

```
int Fibonacci(int n){
     int result;
    if (n==1)
     result=0;
    elseif (n==2)
     result=1;
    else
     result = Fibonacci(n-1) + Fibonacci(n-2);
    return result;
}
```



CE/CZ-1106 2020 ©Mohamed M. Sabry Aly 35

35

Example Solution

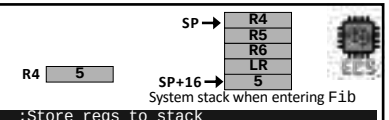
SP → 5

System stack when entering Fib

```
Fib     STMFD   SP!,{R4-R6,LR}   ;Store regs to stack

Done    LDMFD   SP!,{R4-R6,LR}   ; Restore registers
        MOV     PC, LR           ; same as bx lr
```

36

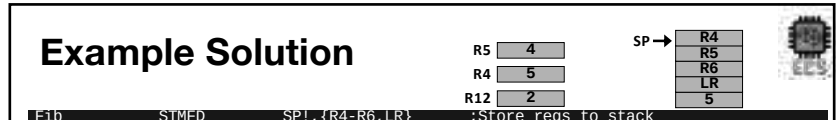
Example Solution




```
Fib     STMFD   SP!,{R4-R6,LR}  ;Store regs to stack
        LDR     R4 ,[SP,#16]    ;Read Value of n
        CMP     R4 ,#1          ;Compare with 1 (if n==1)
        MOVEQ   R12,#0          ;Assign result with 0
        CMP     R4 ,#2          ;Compare with 2 (if n==2)
        MOVEQ   R12,#1          ; Assign result with 1
        BLE     Done            ; if n less than equal 2 finish

Done    LDMFD   SP!,{R4-R6,LR}  ; Restore registers
        MOV     PC, LR          ; same as bx lr
```

37

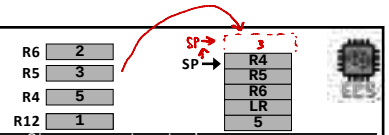
Example Solution




```
Fib     STMFD   SP!,{R4-R6,LR}  ;Store regs to stack
        LDR     R4 ,[SP,#16]    ;Read Value of n
        CMP     R4 ,#1          ;Compare with 1 (if n==1)
        MOVEQ   R12,#0          ;Assign result with 0
        CMP     R4 ,#2          ;Compare with 2 (if n==2)
        MOVEQ   R12,#1          ; Assign result with 1
        BLE     Done            ; if n less than equal 2 finish
        SUB     R5,R4,#1        ; Get n-1
        STMFD   SP!,{R5}        ;Push n-1 to stack
        BL      Fib             ; calculate Fib(n-1)
        LDMFD   SP!,{R5}        ; Pop from stack

Done    LDMFD   SP!,{R4-R6,LR}  ; Restore registers
        MOV     PC, LR          ; same as bx lr
```

38

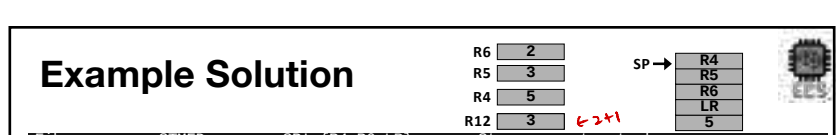
Example Solution




```
Fib     STMFD   SP!,{R4-R6,LR}  ;Store regs to stack
        LDR     R4 ,[SP,#16]    ;Read Value of n
        CMP     R4 ,#1          ;Compare with 1 (if n==1)
        MOVEQ   R12,#0          ;Assign result with 0
        CMP     R4 ,#2          ;Compare with 2 (if n==2)
        MOVEQ   R12,#1          ; Assign result with 1
        BLE     Done            ; if n less than equal 2 finish
        SUB     R5,R4,#1        ; Get n-1
        STMFD   SP!,{R5}        ;Push n-1 to stack
        BL      Fib             ; calculate Fib(n-1)
        LDMFD   SP!,{R5}        ; Pop from stack
        MOV     R6,R12          ;Save result in temp register
        SUB     R5,R4,#2        ; Get n-2
        STMFD   SP!,{R5}        ;Push n-2 to stack
        BL      Fib             ; calculate Fib (n-2)
        LDMFD   SP!,{R5}        ; Pop from stack

Done    LDMFD   SP!,{R4-R6,LR}  ; Restore registers
        MOV     PC, LR          ; same as bx lr
```

39

Example Solution




```
Fib     STMFD   SP!,{R4-R6,LR}  ;Store regs to stack
        LDR     R4 ,[SP,#16]    ;Read Value of n
        CMP     R4 ,#1          ;Compare with 1 (if n==1)
        MOVEQ   R12,#0          ;Assign result with 0
        CMP     R4 ,#2          ;Compare with 2 (if n==2)
        MOVEQ   R12,#1          ; Assign result with 1
        BLE     Done            ; if n less than equal 2 finish
        SUB     R5,R4,#1        ; Get n-1
        STMFD   SP!,{R5}        ;Push n-1 to stack
        BL      Fib             ; calculate Fib(n-1)
        LDMFD   SP!,{R5}        ; Pop from stack
        MOV     R6,R12          ;Save result in temp register
        SUB     R5,R4,#2        ; Get n-2
        STMFD   SP!,{R5}        ;Push n-2 to stack
        BL      Fib             ; calculate Fib (n-2)
        LDMFD   SP!,{R5}        ; Pop from stack
        ADD     R12,R12,R6      ; Calculate Fib(n-1) + Fib(n-2)
Done    LDMFD   SP!,{R4-R6,LR}  ; Restore registers
        MOV     PC, LR          ; same as bx lr
```

40

Example Solution

```
Fib         STMFD       SP!,{R4-R6,LR}      ;Store regs to stack
            LDR         R4 ,[SP,#16]        ;Read Value of n
            CMP         R4 ,#1              ;Compare with 1 (if n==1)
            MOVEQ       R12,#0              ;Assign result with 0
            CMP         R4 ,#2              ;Compare with 2 (if n==2)
            MOVEQ       R12,#1              ; Assign result with 1
            BLE         Done                ; if n less than equal 2 finish
            SUB         R5,R4,#1            ; Get n-1
            STMFD       SP!,{R5}            ;Push n-1 to stack
            BL          Fib                 ; calculate Fib(n-1)
            LDMFD       SP!,{R5}            ; Pop from stack
            MOV         R6,R12              ;Save result in temp register
            SUB         R5,R4,#2            ; Get n-2
            STMFD       SP!,{R5}            ;Push n-2 to stack
            BL          Fib                 ; calculate Fib (n-2)
            LDMFD       SP!,{R5}            ; Pop from stack
            ADD         R12,R12,R6          ; Calculate Fib(n-1) + Fib(n-2)
Done        LDMFD       SP!,{R4-R6,LR}      ; Restore registers
            MOV         PC, LR              ; same as bx lr
```

41

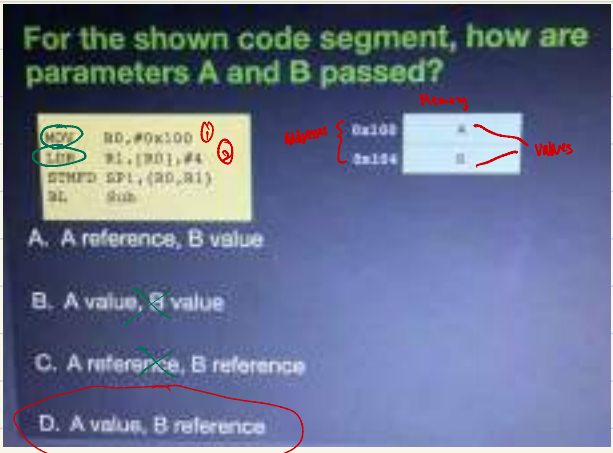
Summary

- Nested subroutines are key for truly modular designs
  - Need to ensure that return address is not overwritten.
- Recursive subroutine is very efficient implementation of subroutines
  - Ensure: 1) stopping condition and 2) return address stored

↳ Stop function from calling itself

↳ to main program

42

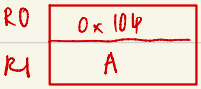


0x104 : reference of B

R1 has the value of A

① R0 has address of A

② post index



← value of 0x100 updated by 4

Mov doesn't mean always pass by ref, it is case paired with LDR, then one value one reference

The segment loads the content of memory address 0x100 to R1 → value of A

we store in the stack then the value of A and the value 0x104 (address of B)

What is the instruction to create a stack frame for 4 local variables (32-bit each)

A. ADD SP, SP, #16

B. SUB SP, SP, #16

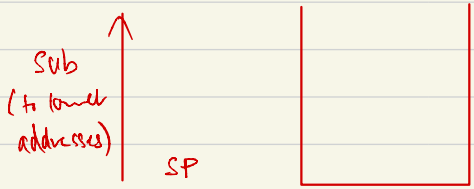
C. SUB SP, SP,#32

D. ADD SP,SP,#32

STR R11, [SP,#-4]!
MOV R11, SP

each is 4 bytes

∴ 4 + 4 = 16 bytes



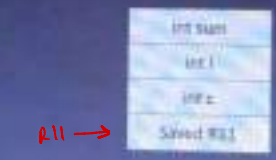
If we are using R11 as frame pointer, what is the instruction to store R4 to local variable c

A. STR R4, [R11,#-1]

B. STR R4, [R11,#-4]

C. STR R4, [R11,#-4]!

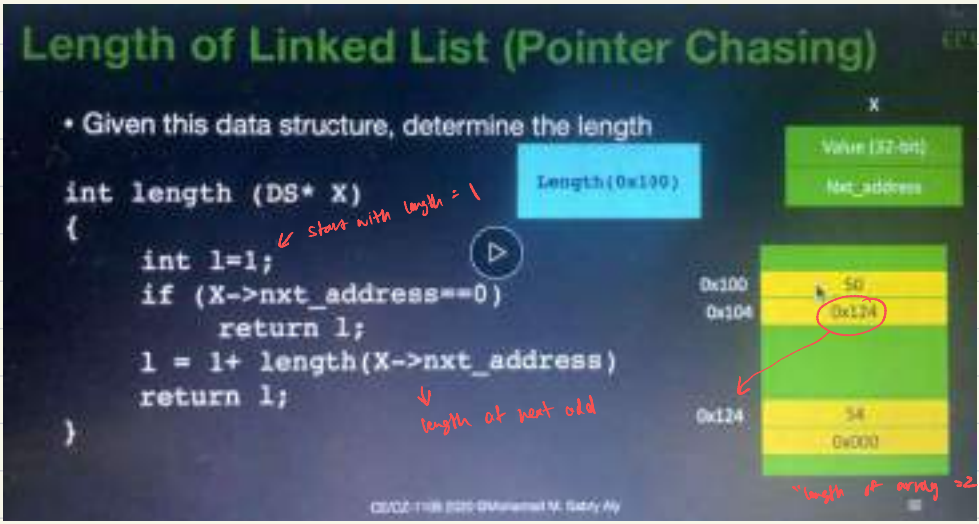
D. STMFD R11!,{R4}

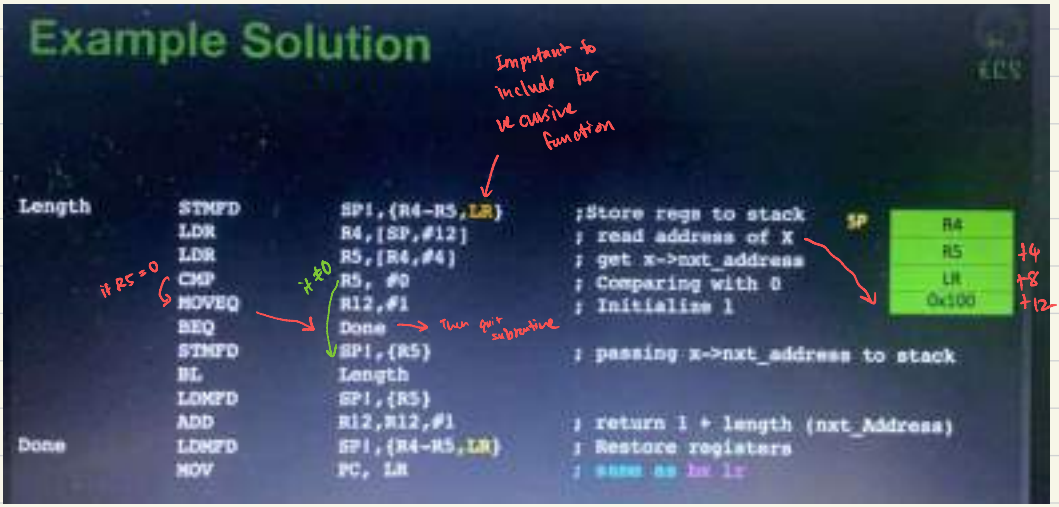


if r11 was a stack pointer, then we would have to update it

10/2 Lecture

Alternative Example of Fibonnaci Sequence:





Main program:

```
MOV   R5, #0x100     ; Put 0x100 to R5
STMFD SP!, {R5}      ; Push to stack
BL    Length         ; go execute length
```


read R4 with offset of 4 and store it into R5
0x100
0x104
0x108
0x10C
0x124
0x140
50
0x124
10
0x000
54
0x140
-1
0x108
SP
R4
R5
LR
0x100
+4
+8
+12
R4
R5
R12
0X100
0X124
①
②


On Compare, R5 is ≠0
0x100
0x104
0x108
0x10C
0x124
0x140
50
0x124
10
0x000
54
0x140
-1
0x108
SP
0x124
R4
R5
LR
0x100
R4
R5
R12
0X100
0X124
③


0x100
0x104
0x108
0x10C
0x124
0x140
50
0x124
10
0x000
54
0x140
-1
0x108
SP
0x124
R4
R5
LR
0x100
R4
R5
R12
0X100
0X124

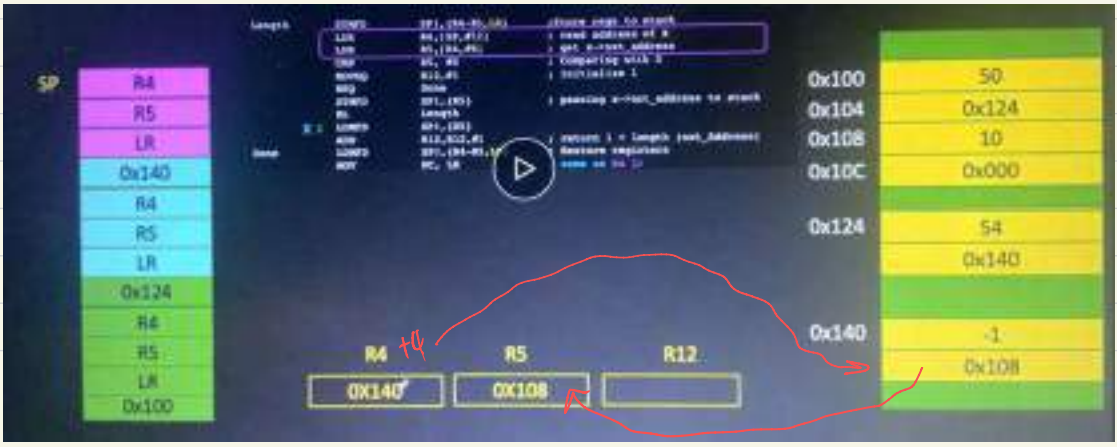

SP
R4
R5
LR
0x124
R4
R5
LR
0x100
Args by value
R4
R5
R12
0X100
0X124
0x100
50
0x104
0x124
0x108
10
0x10C
0x000
0x124
54
0x140
0x140
-1
0x108


SP
R4
R5
LR
0x124
R4
R5
LR
0x100
R4
R5
R12
0X124
0X140
0x100
50
0x104
0x124
0x108
10
0x10C
0x000
0x124
54
0x140
0x140
-1
0x108


SP
0x140
R4
R5
LR
0x124
R4
R5
LR
0x100
R4
R5
R12
0X124
0X140
0x100
50
0x104
0x124
0x108
10
0x10C
0x000
0x124
54
0x140
0x140
-1
0x108


SP
0x140
R4
R5
LR
0x124
R4
R5
LR
0x100
R4
R5
R12
0X124
0X140
0x100
50
0x104
0x124
0x108
10
0x10C
0x000
0x124
54
0x140
0x140
-1
0x108

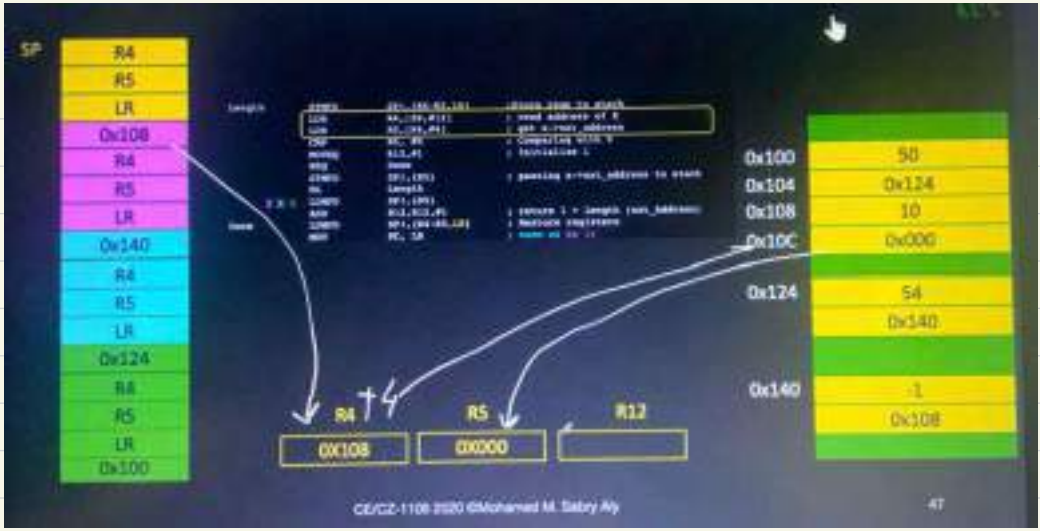

SP
R4
R5
LR
0x140
R4
R5
LR
0x124
R4
R5
LR
0x100
R4
R5
R12
0X124
0X140
0x100
50
0x104
0x124
0x108
10
0x10C
0x000
0x124
54
0x140
0x140
-1
0x108

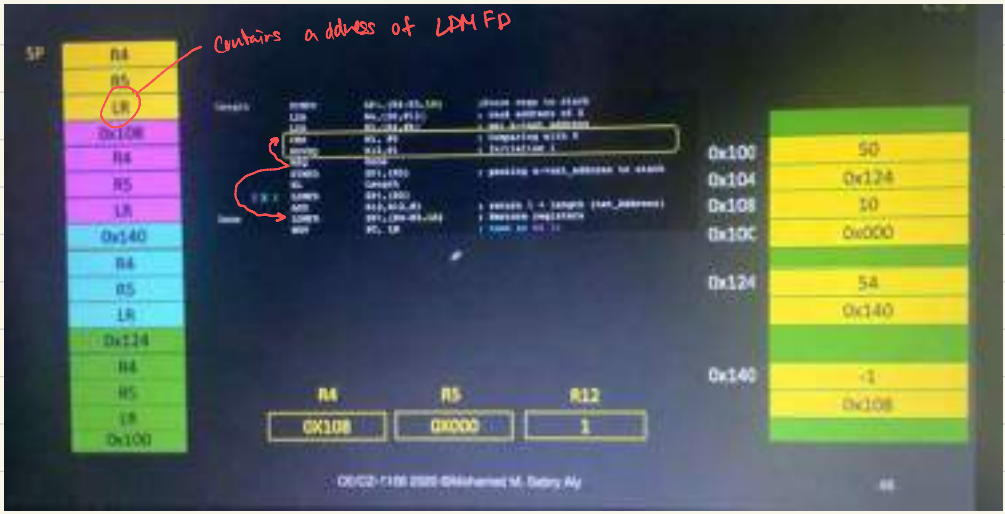

SP
R4
R5
LR
0x140
R4
R5
LR
0x124
R4
R5
LR
0x100
R4 +4
R5
R12
0X140
0X108
0x100
50
0x104
0x124
0x108
10
0x10C
0x000
0x124
54
0x140
0x140
-1
0x108


SP
0x108
R4
R5
LR
0x140
R4
R5
LR
0x124
R4
R5
LR
0x100
0x100
50
0x104
0x124
0x108
10
0x10C
0x000
0x124
54
0x140
0x140
-1
0x108
R4
R5
R12
0X140
0X108


SP
0x108
R4
R5
LR
0x140
R4
R5
LR
0x124
R4
R5
LR
0x100
0x100
50
0x104
0x124
0x108
10
0x10C
0x000
0x124
54
0x140
0x140
-1
0x108
R4
R5
R12
0X140
0X108


SP
R4
R5
LR
0x108
R4
R5
LR
0x140
R4
R5
LR
0x124
R4
R5
LR
0x100
0x100
50
0x104
0x124
0x108
10
0x10C
0x000
0x124
54
0x140
0x140
-1
0x108
R4
R5
R12
0X140
0X108

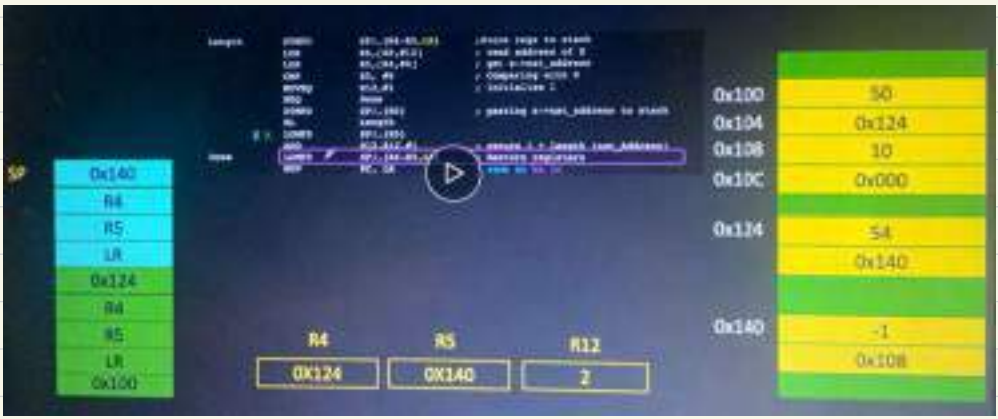

SP
R4
R5
LR
0x108
R4
R5
LR
0x140
R4
R5
LR
0x124
R4
R5
LR
0x100
0x100
50
0x104
0x124
0x108
10
0x10C
0x000
0x124
54
0x140
0x140
-1
0x108
R4
R5
R12
0X108
0X000
CE/CZ-1106 2020 ©Mohamed M. Sabry Aly
47


SP
Contains address of LDM FD
R4
R5
LR
0x108
R4
R5
LR
0x140
R4
R5
LR
0x124
R4
R5
LR
0x100
0x100
50
0x104
0x124
0x108
10
0x10C
0x000
0x124
54
0x140
0x140
-1
0x108
R4
R5
R12
0X108
0X000
1


SP
0x108
R4
R5
LR
0x140
R4
R5
LR
0x124
R4
R5
LR
0x100
0x100
50
0x104
0x124
0x108
10
0x10C
0x000
0x124
54
0x140
0x140
-1
0x108
R4
R5
R12
0X140
0X108
1


SP
R4
R5
LR
0x140
R4
R5
LR
0x124
R4
R5
LR
0x100
0x100 50
0x104 0x124
0x108 10
0x10C 0x000
0x124 54
0x140
0x140 -1
0x108
R4 R5 R12
0X140 0X108 1


SP
R4
R5
LR
0x140
R4
R5
LR
0x124
R4
R5
LR
0x100
0x100 50
0x104 0x124
0x108 10
0x10C 0x000
0x124 54
0x140
0x140 -1
0x108
R4 R5 R12
0X140 0X108 2


SP
0x140
R4
R5
LR
0x124
R4
R5
LR
0x100
0x100 50
0x104 0x124
0x108 10
0x10C 0x000
0x124 54
0x140
0x140 -1
0x108
R4 R5 R12
0X124 0X140 2


SP
0x140
R4
R5
LR
0x124
R4
R5
LR
0x100
R4
R5
R12
0X124
0X140
2
0x100
50
0x104
0x124
0x108
10
0x10C
0x000
0x124
54
0x140
0x140
-1
0x108


SP
R4
R5
LR
0x124
R4
R5
LR
0x100
R4
R5
R12
0X124
0X140
2
0x100
50
0x104
0x124
0x108
10
0x10C
0x000
0x124
54
0x140
0x140
-1
0x108
CE/CZ-1106 2020 ©Muhammad Ali Sabry Aly


SP
R4
R5
LR
0x124
R4
R5
LR
0x100
R4
R5
R12
0X124
0X140
3
0x100
50
0x104
0x124
0x108
10
0x10C
0x000
0x124
54
0x140
0x140
-1
0x108

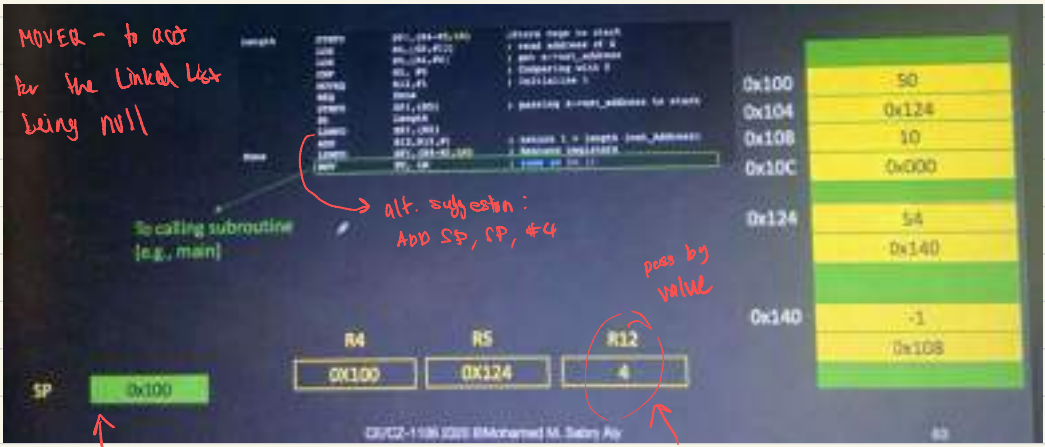

SP
0x124
R4
R5
LR
0x100
R4
R5
R12
0X100
0X124
3
0x100
50
0x104
0x124
0x108
10
0x10C
0x000
0x124
54
0x140
0x140
-1
0x108


SP
0x124
R4
R5
LR
0x100
R4
R5
R12
0X100
0X124
3
0x100
50
0x104
0x124
0x108
10
0x10C
0x000
0x124
54
0x140
0x140
-1
0x108


SP
R4
R5
LR
0x100
R4
R5
R12
0X100
0X124
3
0x100
50
0x104
0x124
0x108
10
0x10C
0x000
0x124
54
0x140
0x140
-1
0x108


0x100
0x104
0x108
0x10C
0x124
0x140
50
0x124
10
0x000
54
0x140
-1
0x108
SP
R4
R5
LR
0x100
R4
R5
R12
0X100
0X124
4


0x100
0x104
0x108
0x10C
0x124
0x140
50
0x124
10
0x000
54
0x140
-1
0x108
R4
R5
R12
0X100
0X124
4
SP
0x100


MOVER - to act for the Linked List being NULL
To calling subroutine (e.g., main)
alt. suggestion : ADD SP, SP, #4
pass by value
0x100
0x104
0x108
0x10C
0x124
0x140
50
0x124
10
0x000
54
0x140
-1
0x108
R4
R5
R12
0X100
0X124
4
SP
0x100
left with what was passed from main program
length