Chapter 6: Modular Programming

Mohamed M. Sabry Aly

N4-02c-92

Learning Objectives

- Describe ARM instructions in implement subroutines

- Describe passing parameters to subroutines using:
  - Registers
  - Memory

Modular Program Design

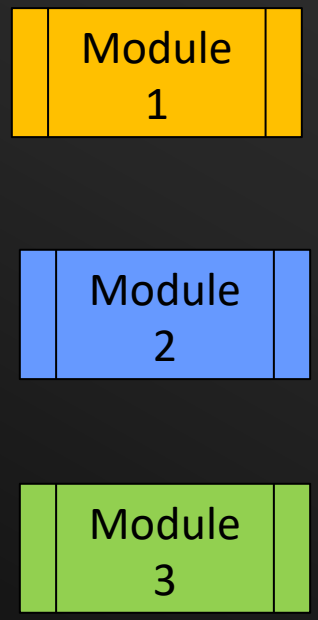
- Real-world applications are very large and complex
  - Google Chrome: ~6.7 Millions line of code
  - Android OS: 12-15 Millions line of code
  - Boeing 787: ~6.5 Millions line of code

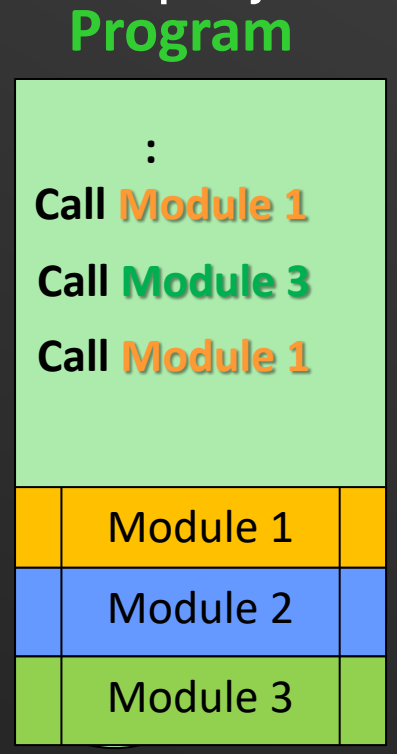
- Large software cannot be a single function

- It is decomposed into several less complex **modules**

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





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**

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;
}
```

$$\sigma = \sqrt{\frac{\sum (x_i - \mu)^2}{N}}$$

Example: Standard Deviation

Case 2: Modular

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

$$\sigma = \sqrt{\frac{\sum (x_i - \mu)^2}{N}}$$

```
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;
}
```

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?

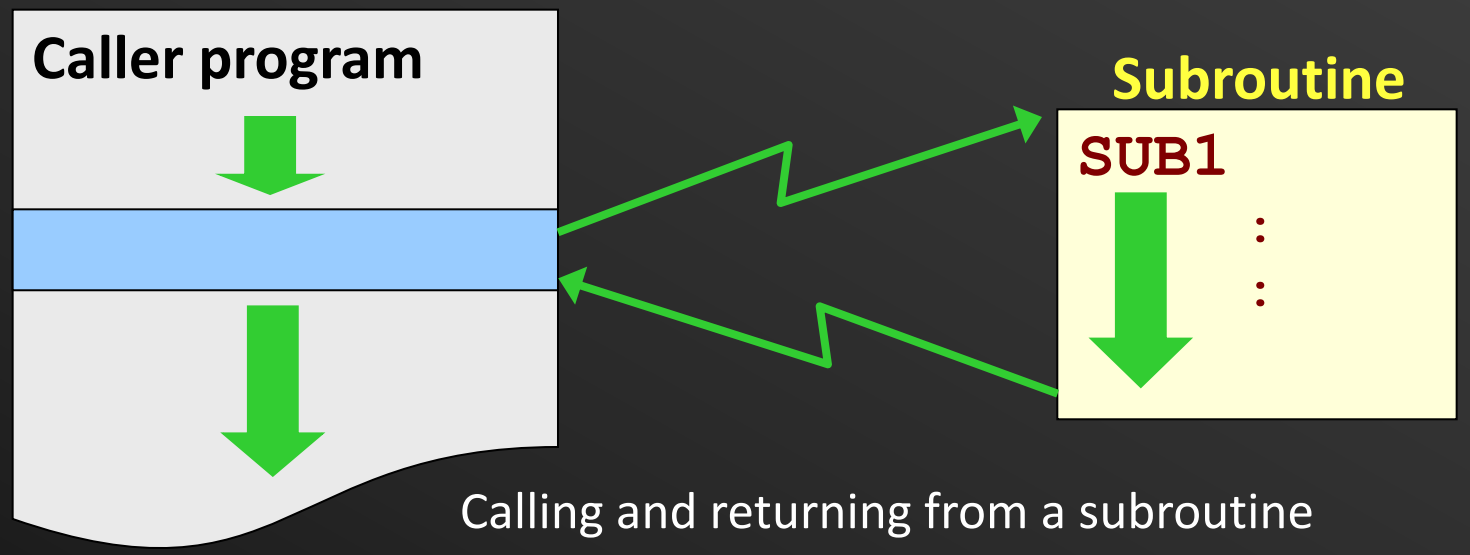
How will ARM go to these two functions and return?

```
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;
}
```

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)



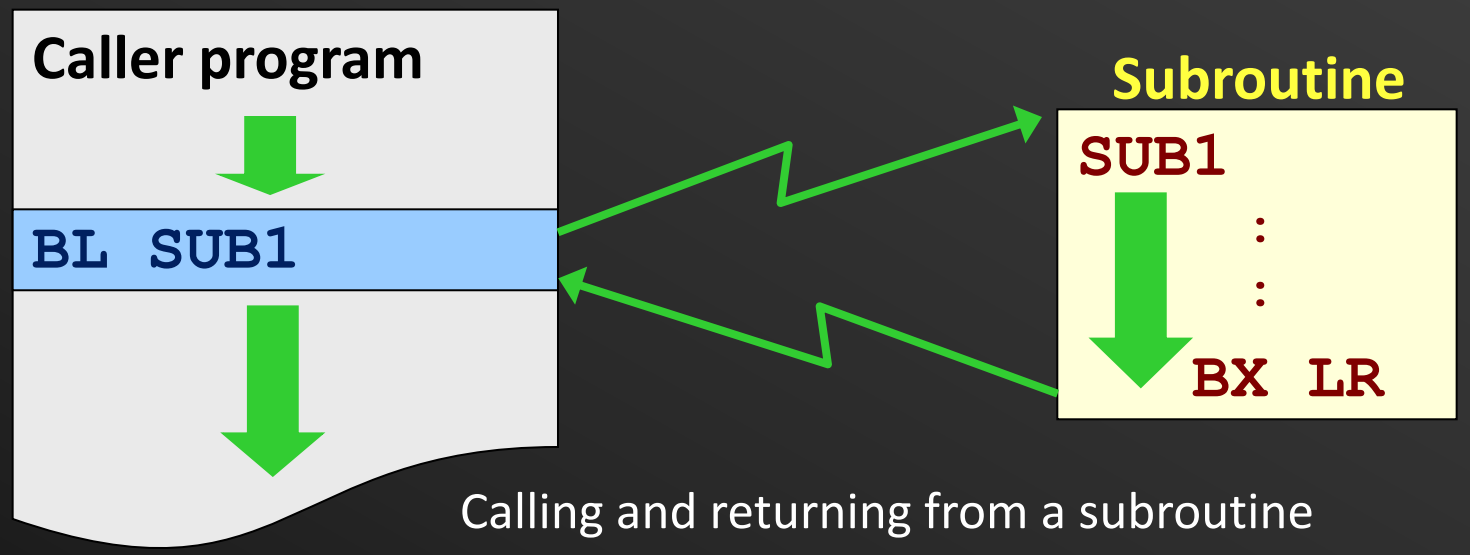
Calling and returning from a subroutine

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`



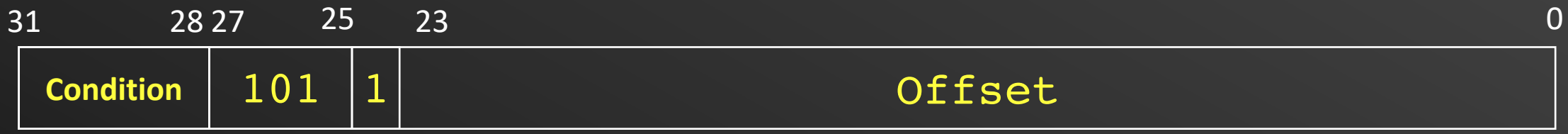
Calling and returning from a subroutine

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

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)

Branch with Link (`BL`)

- **`BL`** used to make subroutine call

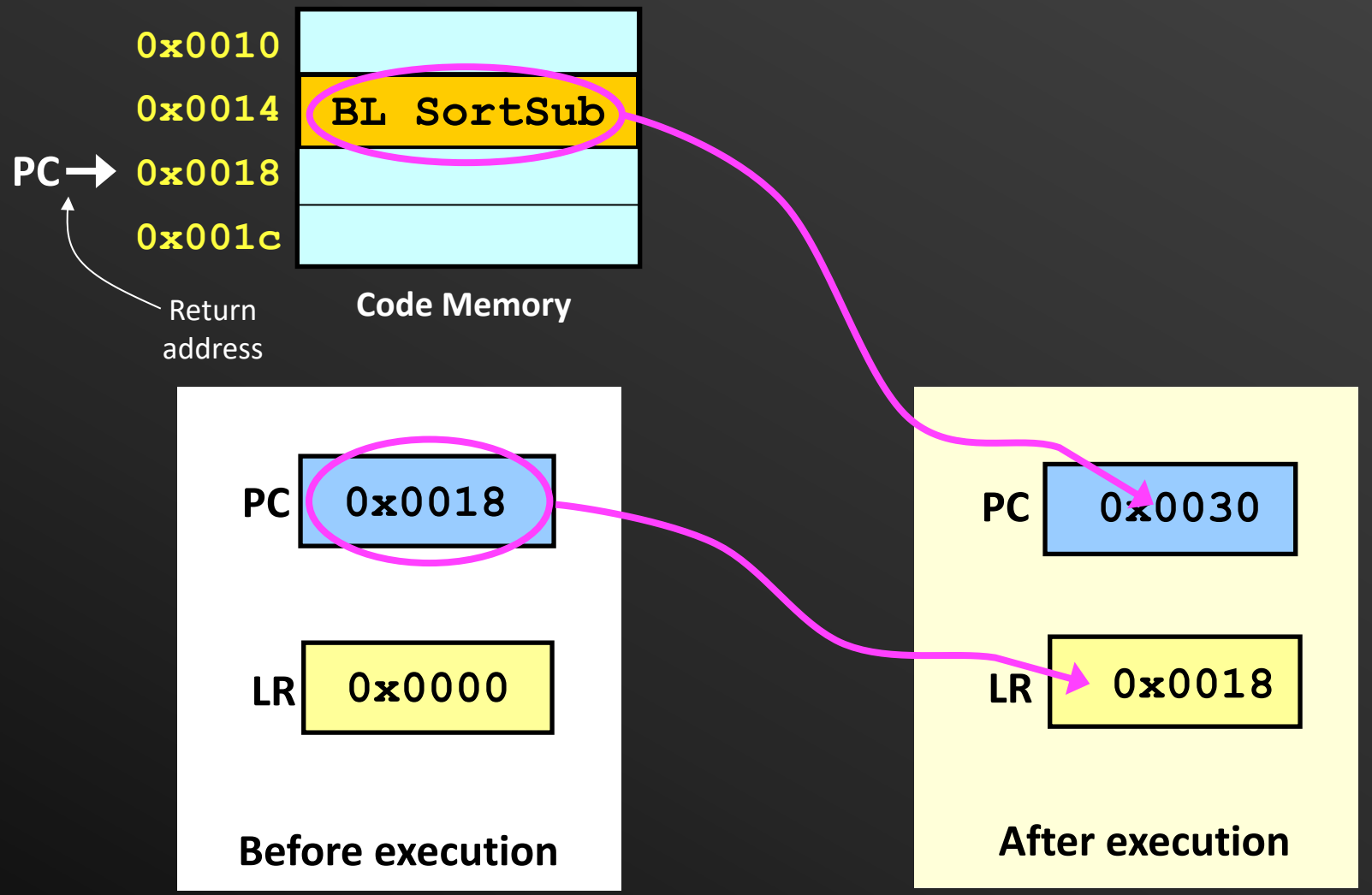
Subroutine Call

```
BL SortSub
```

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

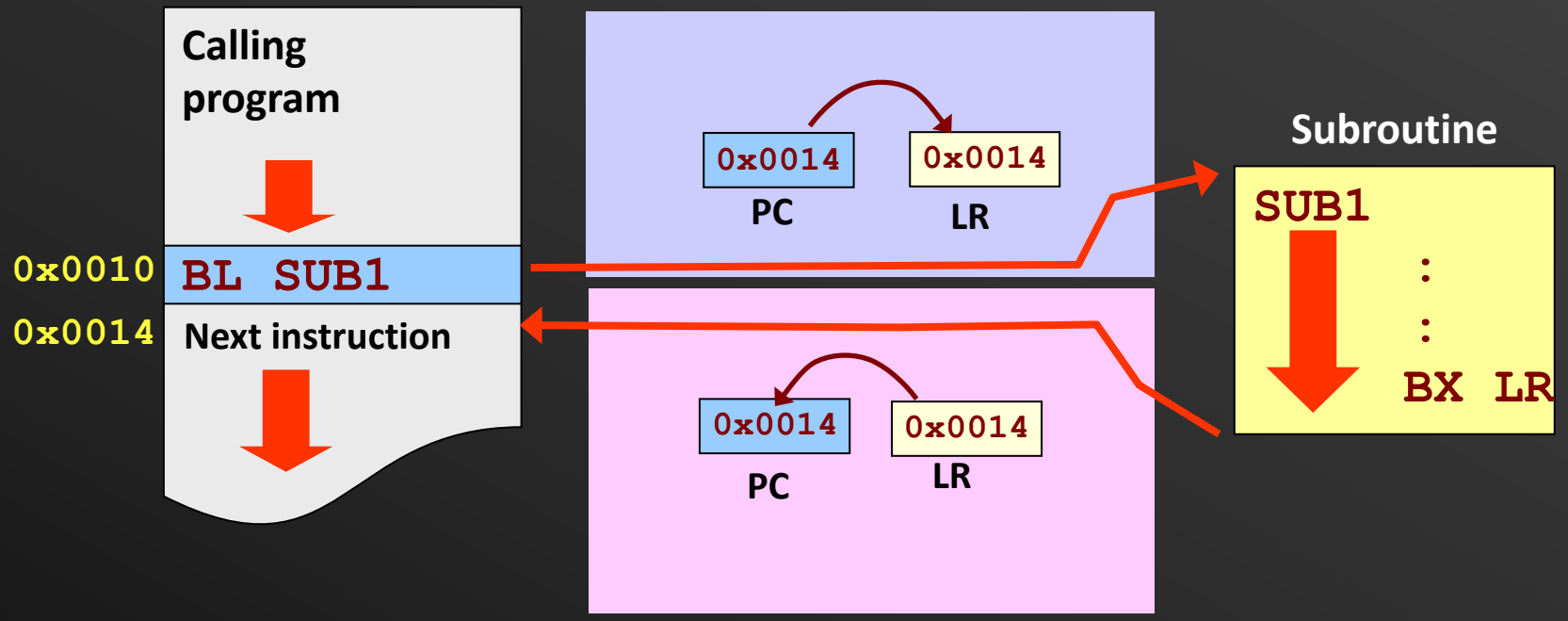
BL (Execution example)

- BL SortSub (assume SortSub is in 0x0030)



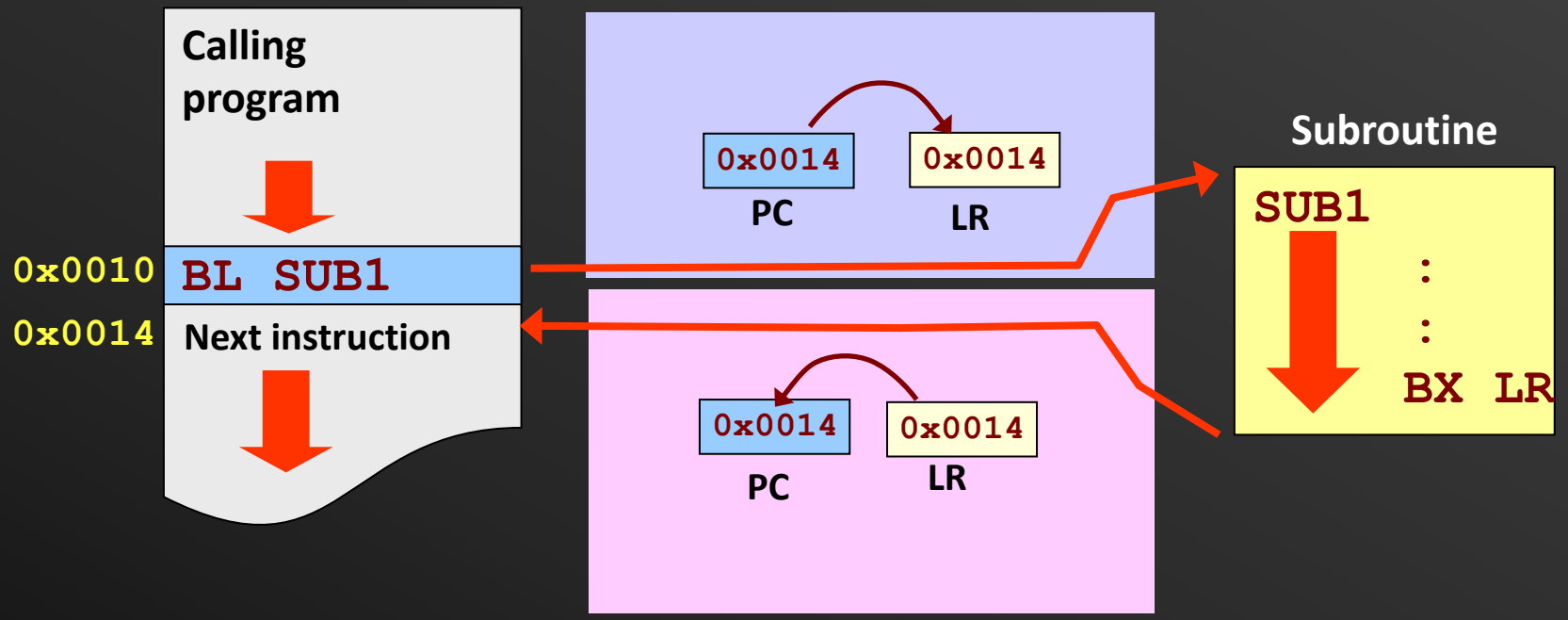
BX instruction

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



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`

Reset to continue editing code

```
1  Num1    DCD    0x20
2  Num2    DCD    0x14
3  Result  DCD    0
4          ADR    SP, 0xFFFFFFFC
5          ADR    r0, Num1
6          ADR    r1, Num2
7          ADR    r2, Result
8          bl     Mean
9          END
10
11
12 Mean    LDR    r4,[r0]
13         LDR    r5,[r1]
14         add    r4,r4,r5
15         lsr    r4,r4,#1
16         STR    r4,[r2]
17         bx     lr
18
```

Syntax Error

Unsupported instruction.

Press F1 for a list of supported instructions.

New | Open | Save | Settings | Tools

Emulation Complete — Line 8 Issues 0

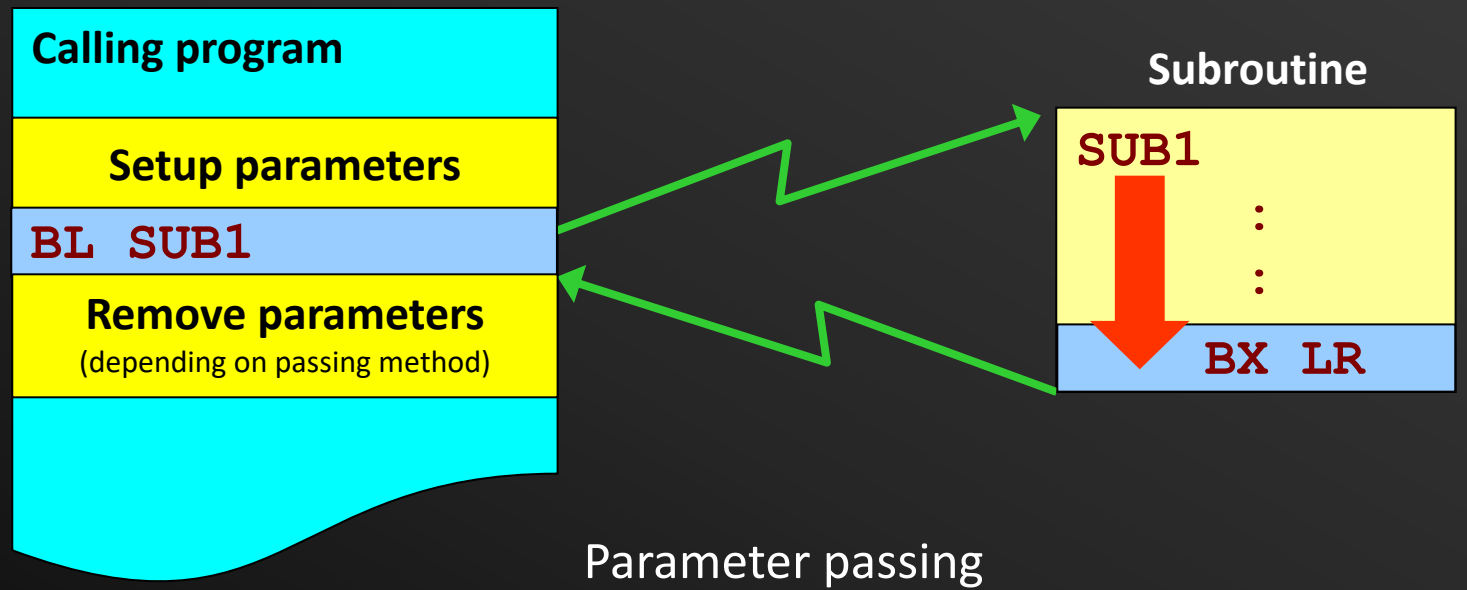
Reset to continue editing code

```
1  Num1    DCD    0x20
2  Num2    DCD    0x14
3  Result  DCD    0
4          ADR    SP, 0xFFFFFFFC
5          ADR    r0, Num1
6          ADR    r1, Num2
7          ADR    r2, Result
8          bl     Mean
9          END
10
11
12 Mean    LDR    r4,[r0]
13         LDR    r5,[r1]
14         add    r4,r4,r5
15         lsr    r4,r4,#1
16         STR    r4,[r2]
17         mov    pc,lr
18
```

R0 R1 R2 R3 R4 R5 R6 R7 R8 R9

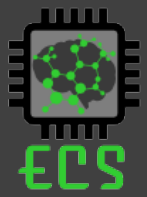
Parameter Passing

- 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**.



Parameter passing

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

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

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

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

R0-R3
Can be used to pass argument values
to return values from subroutine
Subroutine can modify values

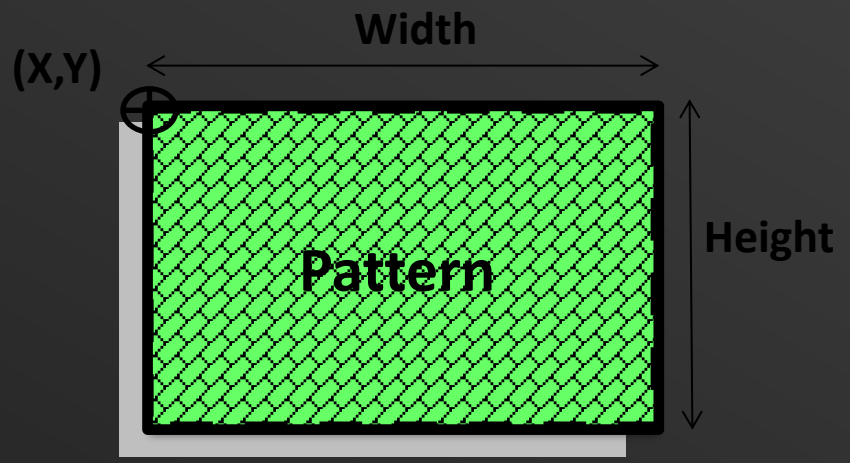
R4-R11
Used to hold local variables
Not for passing arguments
Must be preserved in the subroutine

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

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);
```



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.

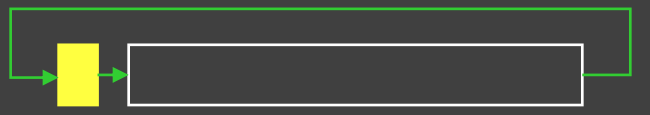
- **Con – lacks generality** due to the limited number of registers.

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**.

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`

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`

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`

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`

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`

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`

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`

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`

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`

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**

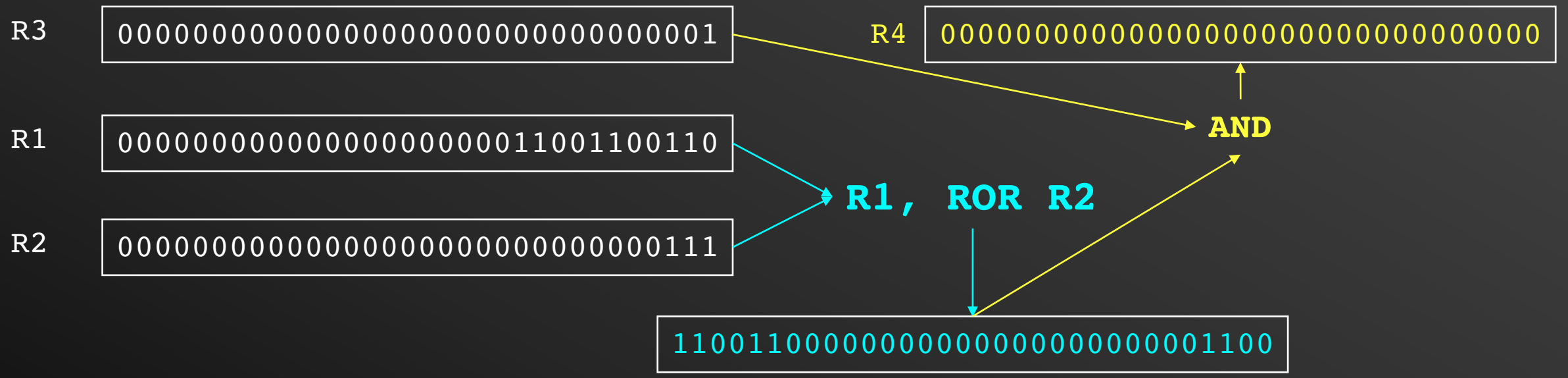
```
         MOV     PC, LR        ; same as bx lr
```

Value passed in R1, return value in R0

VisUAL does not support `bx lr`

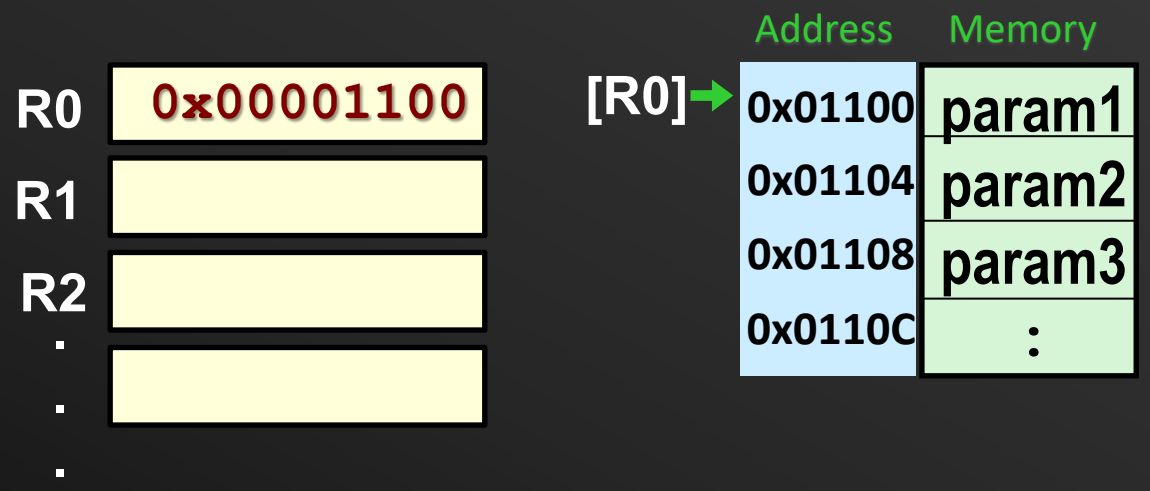
A Deep Dive

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



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**.



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:

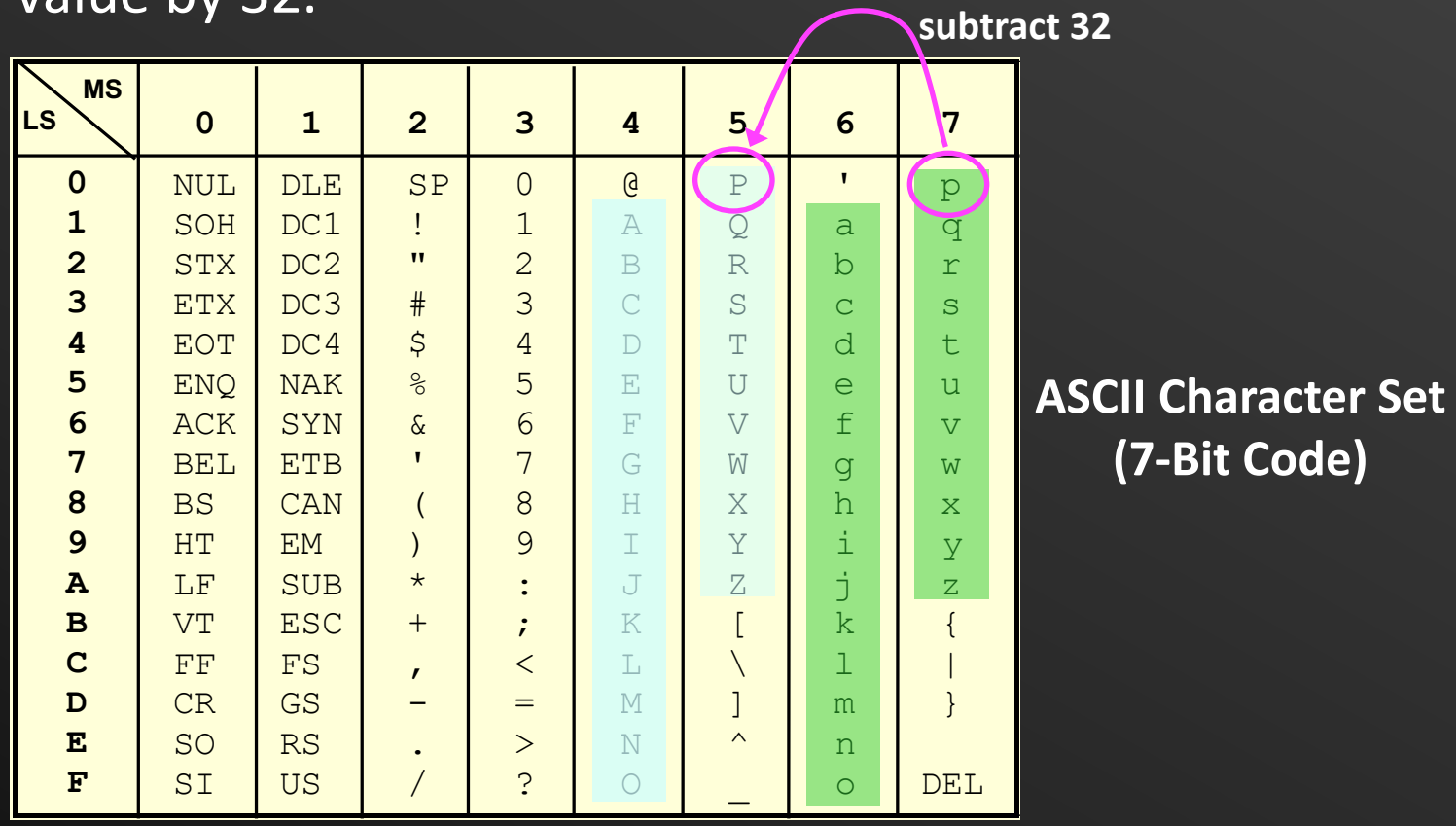
```
;Calling program
    :
MOV    R0,#0x100   ;move start addr. of string to R1
BL     Lo2Up       ;branch to Lo2Up subroutine
    :
```

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

Algorithm Design

- How to convert an ASCII character from lower to upper case?

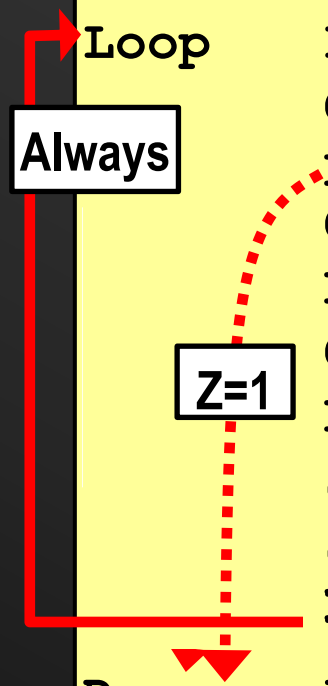
  - Check that the character's value is between 'a' and 'z'.
  - If so, subtract its value by 32.



LS \ MS	0	1	2	3	4	5	6	7
0	NUL	DLE	SP	0	@	P	'	p
1	SOH	DC1	!	1	A	Q	a	q
2	STX	DC2	"	2	B	R	b	r
3	ETX	DC3	#	3	C	S	c	s
4	EOT	DC4	$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	'	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[	k	{
C	FF	FS	,	<	L	\	l	|
D	CR	GS	-	=	M	]	m	}
E	SO	RS	.	>	N	^	n	
F	SI	US	/	?	O	_	o	DEL

ASCII Character Set (7-Bit Code)

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
```

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

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

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
```

R0=0x100

R0=0x104

R0-4=0x100

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

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	:

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	:

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
```

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

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.

Chapter 6: Modular Programming-Cont

Mohamed M. Sabry Aly

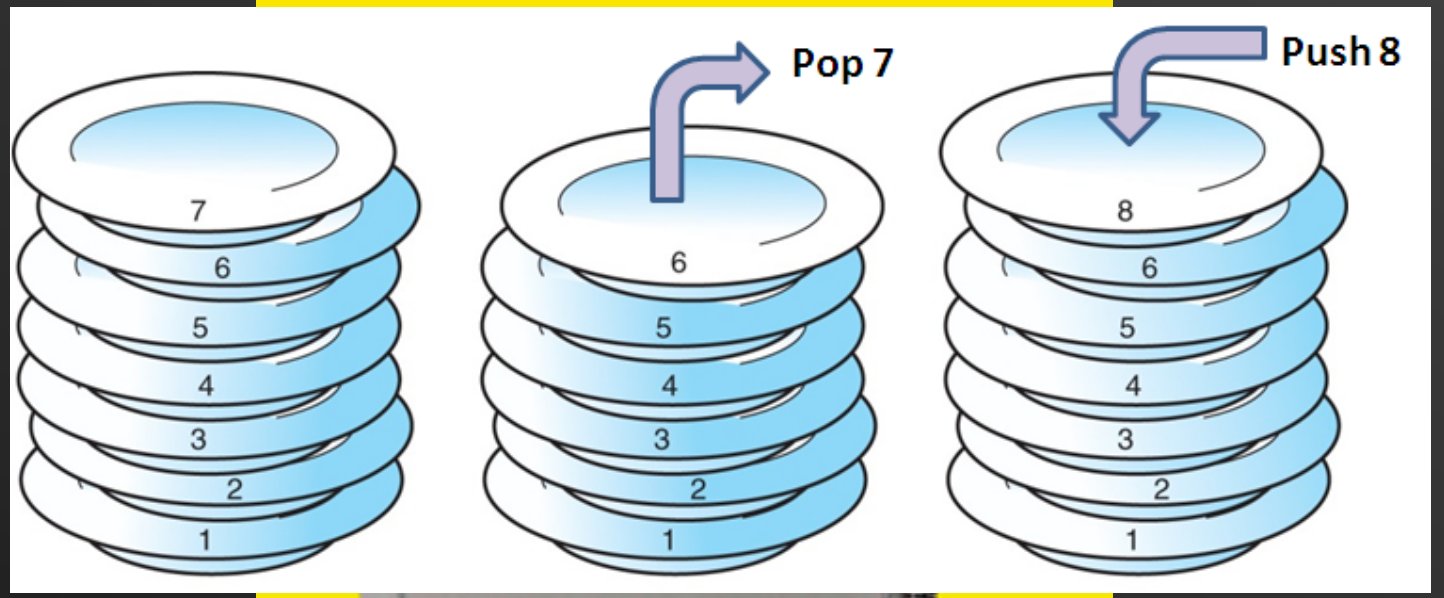
N4-02c-92

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.

System Stack

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

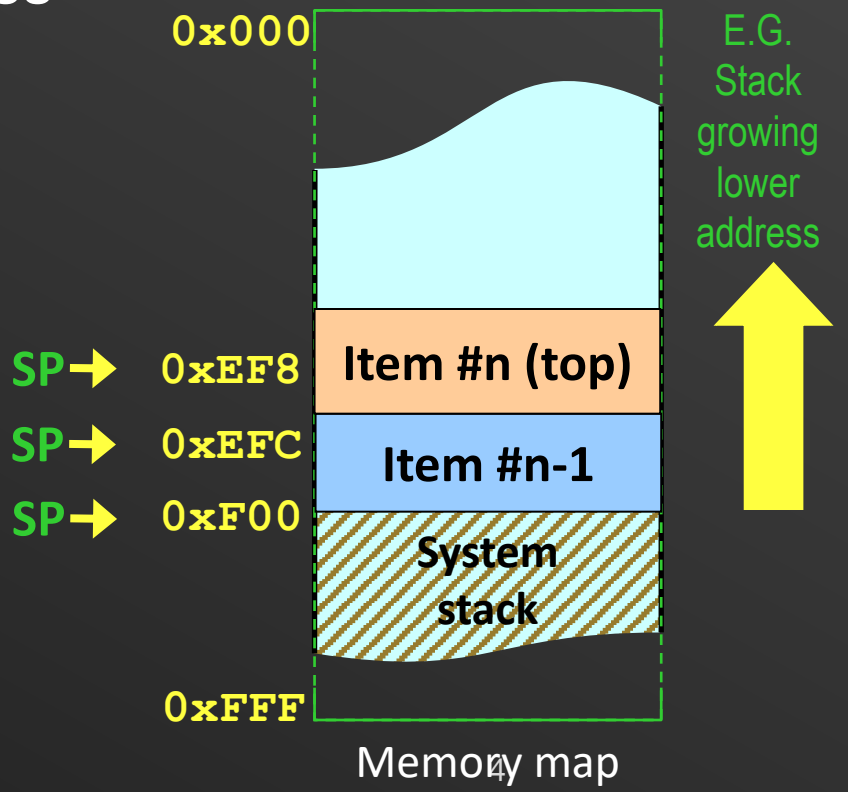


Stack of warm plates on dispenser

System Stack

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

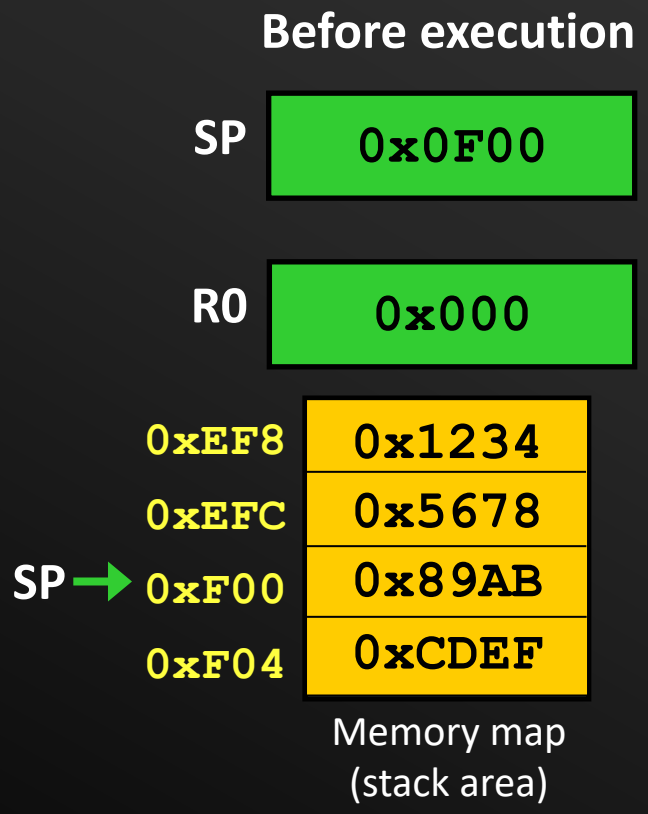
  - The system stack in the ARM processor is maintained by a dedicated stack pointer (**SP, R13**).
  - Stack can grow towards lower or higher memory address
    - Preferred direction is lower, start from max address ➔ Why?
  - The **SP** points to the top item on the system stack.
  - The 3 basic stack operations are **push**, **pop** and **access** items on the stack.

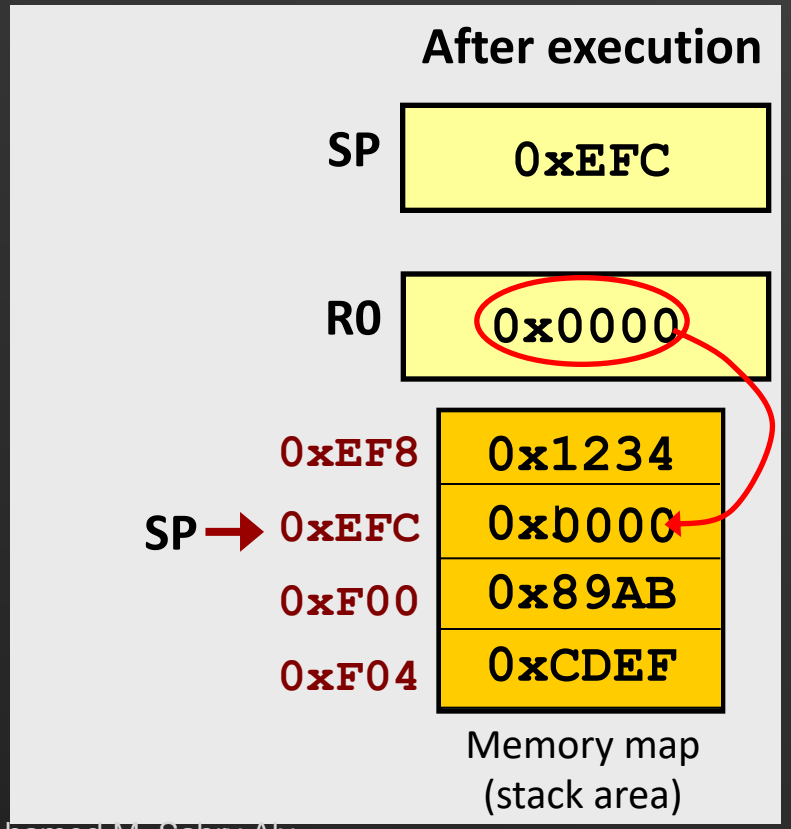


Ref: Null & Lobur (3rd Ed) section A.2.3 – Stacks

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]!`





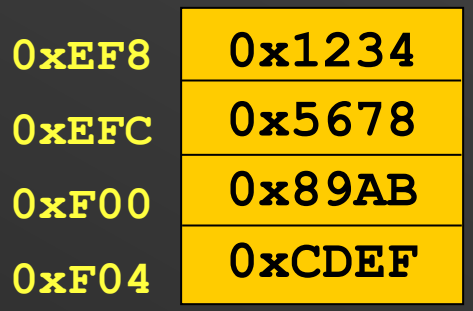
Pop Data off the Stack

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

Removing from stack

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

NO



- Data still resides in memory and can be read

```
LDR R0, [SP, #-4]
```

SP is not changed

Pushing Registers to Stack

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

Method 1:

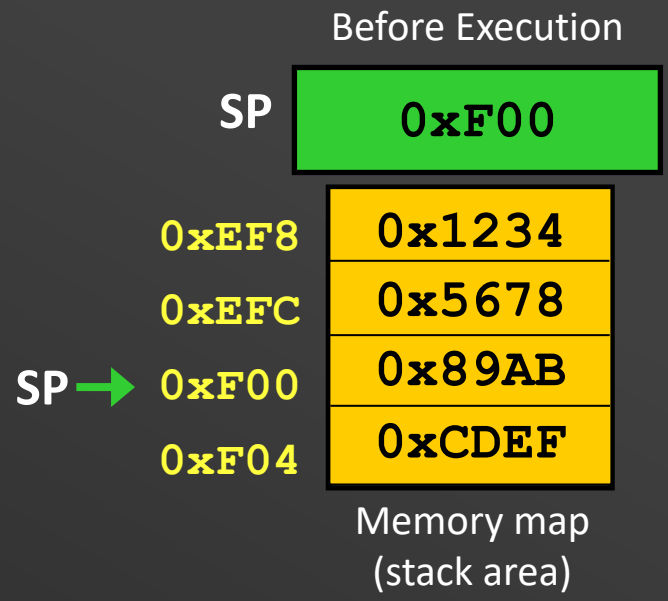
```
STR R0, [SP, #-4]!
STR R1, [SP, #-4]!
```

Pushing Registers to Stack

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

Method 1:

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



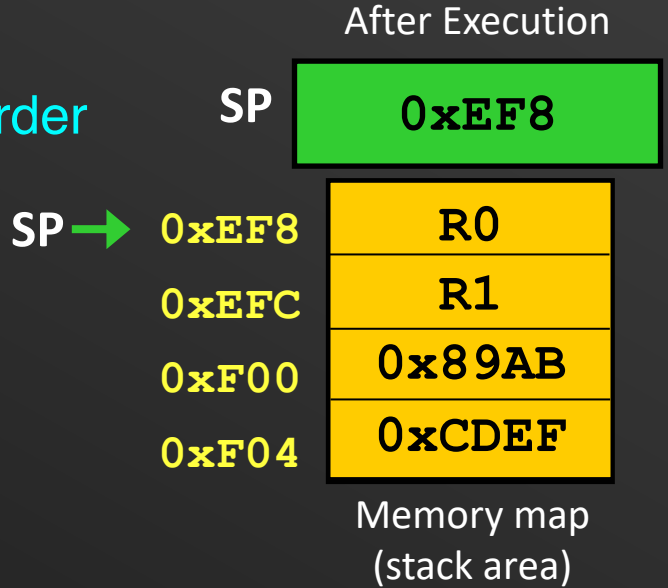
Method 2:

Use Stack pointer and write back updated value after operation

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

Write registers in descending order

Store multiple registers fully descending



Pushing and popping to stack

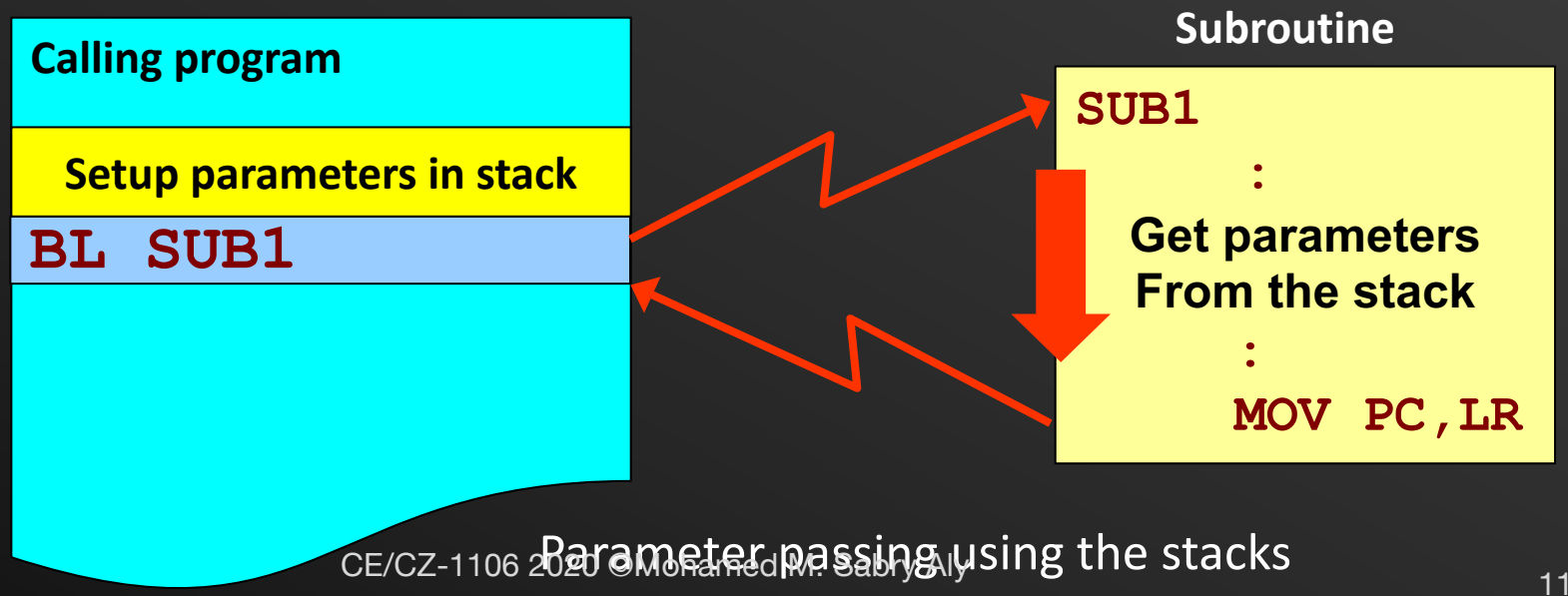
```
STMFD SP!, {list of registers}

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

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

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

Parameter Passing using Stack

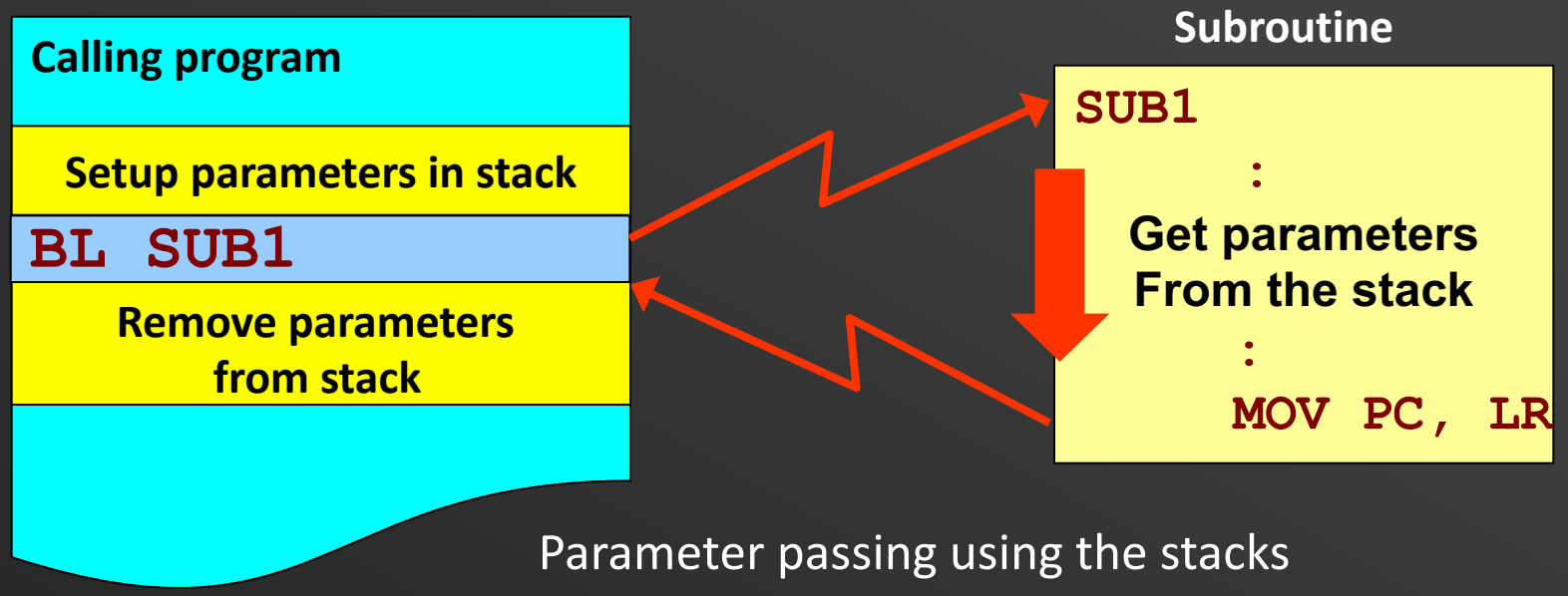
- Parameters are pushed onto the stack before calling the subroutine and retrieved from the stack within the subroutine.

- 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

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.

- If not, repeated pushing of parameters to the stack will lead to a stack overflow.

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:

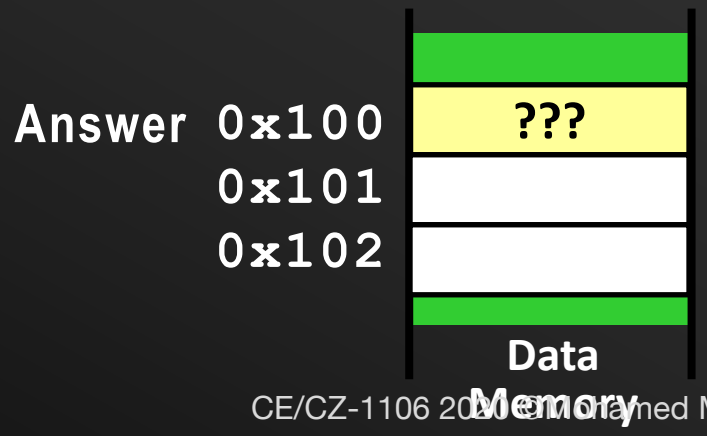
- Push two parameters on stack, the value of **N** and **address** of memory variable **Answer**.

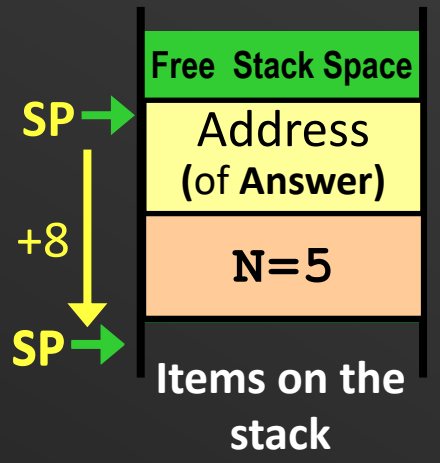
Calling the Subroutine

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

Parameters setup

Remove parameters from the Stack





Calling the Subroutine

Parameters setup

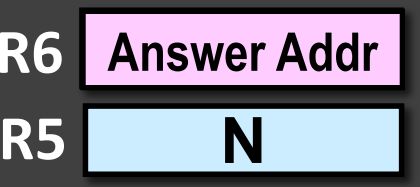
```
      :
MOV   R1, #5          ;Set R0 with #5
MOV   R0, #0x100      ;Set R1 with the address
STMFD SP!, {R1,R0}    ;push R0&R1 to stack
BL    Sum1N           ;call subroutine Sum1N
ADD   SP,SP,#8        ;add 8 to pop the two parameters from stack
      :
```

;find the sum of (1+2+3+4+5), where N=5

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.

Sum from 1 to N Subroutine
Possible Solution




```
Sum1N   STMFD  SP!,{R4,R5,R6}     ;save registers to stack
        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
        BNE    Loop              ;jump back to Loop if R4 not yet zero
        STR    R4,[R6]           ;write sum to Answer
        LDMFD  SP!,{R4,R5,R6}    ;restored saved registers
        MOV PC,LR                ;return from subroutine
```

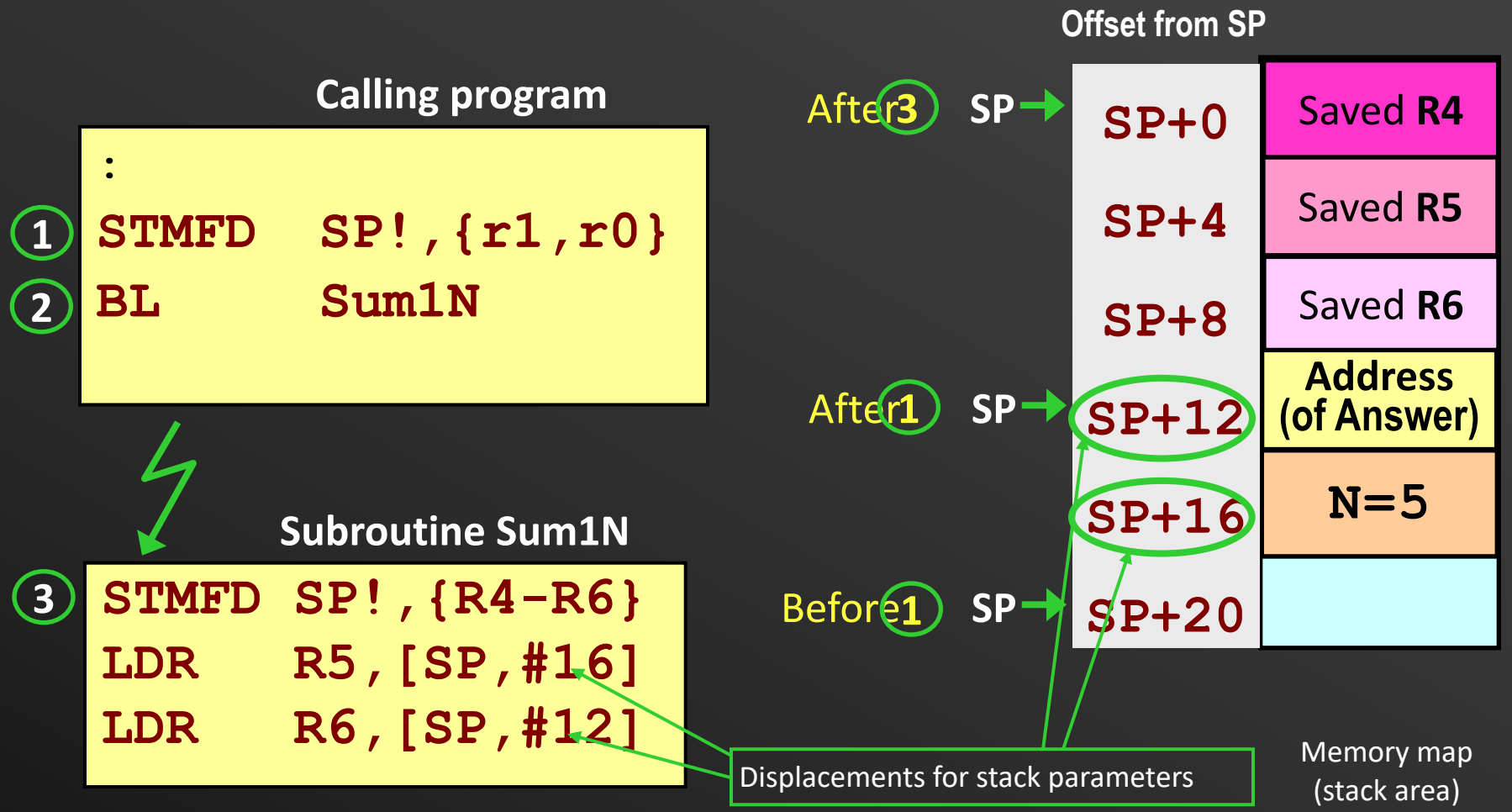
Retrieve stack parameters

Z=0

Z=1

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.

Sum from 1 to N Subroutine
Accessing Stack Parameters

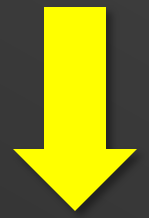


Note: **SP indirect with offset** can be used to access parameters on the stack but knowledge of all items on the stack is needed to compute the correct offset from the current **SP** position.

Transparent Subroutines

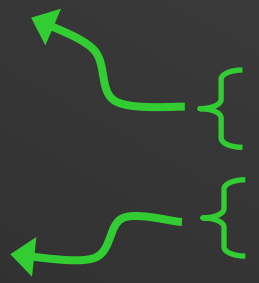
- A transparent subroutine will **not affect any CPU resources** used by the program calling it.
- To achieve this, all local registers used by the subroutine (R4-R11) must be **saved on the stack** on entry and **restored from stack** before returning.

```
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
```



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.
  - **Pass by reference** – the address of the variable is passed to the subroutine .



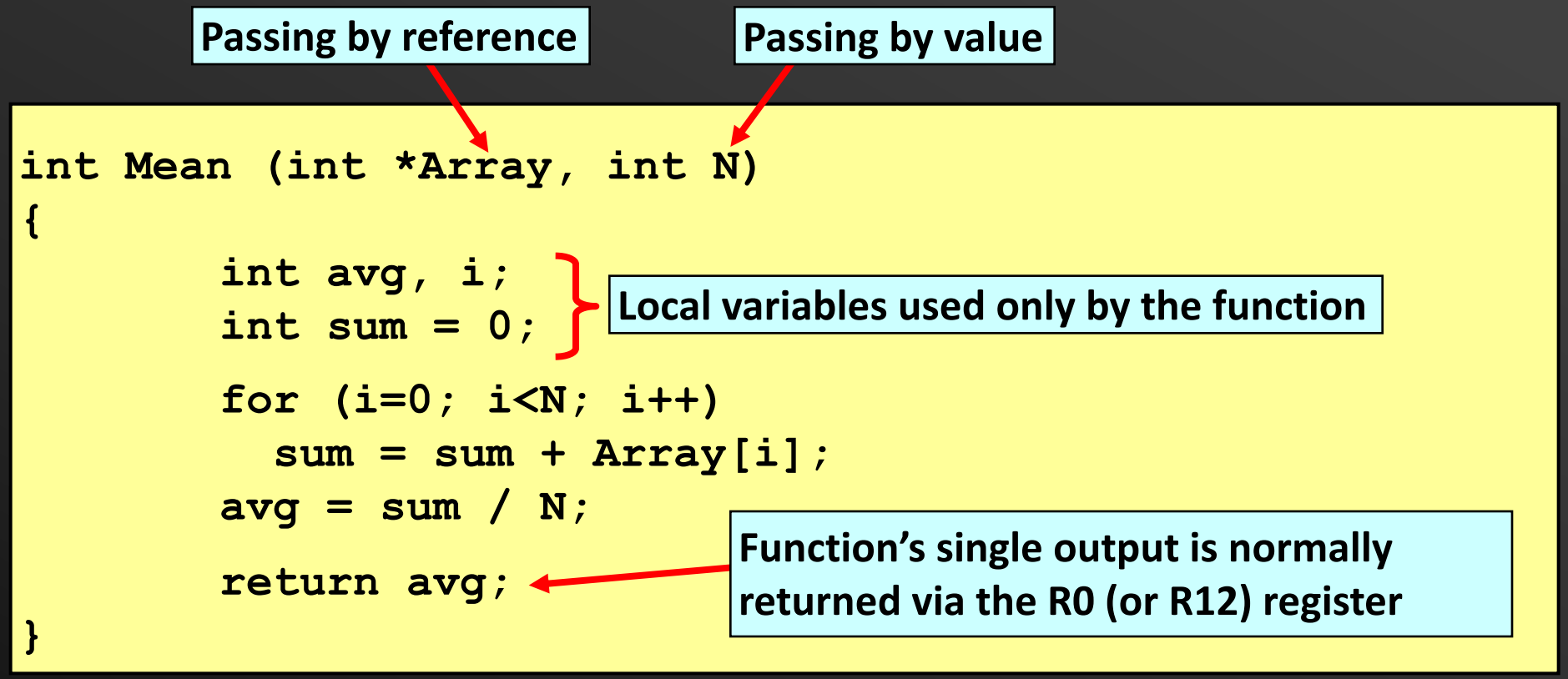
Calling program

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

- 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.

C Function Example

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



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

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;
}
```

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

Local Variables

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

Review

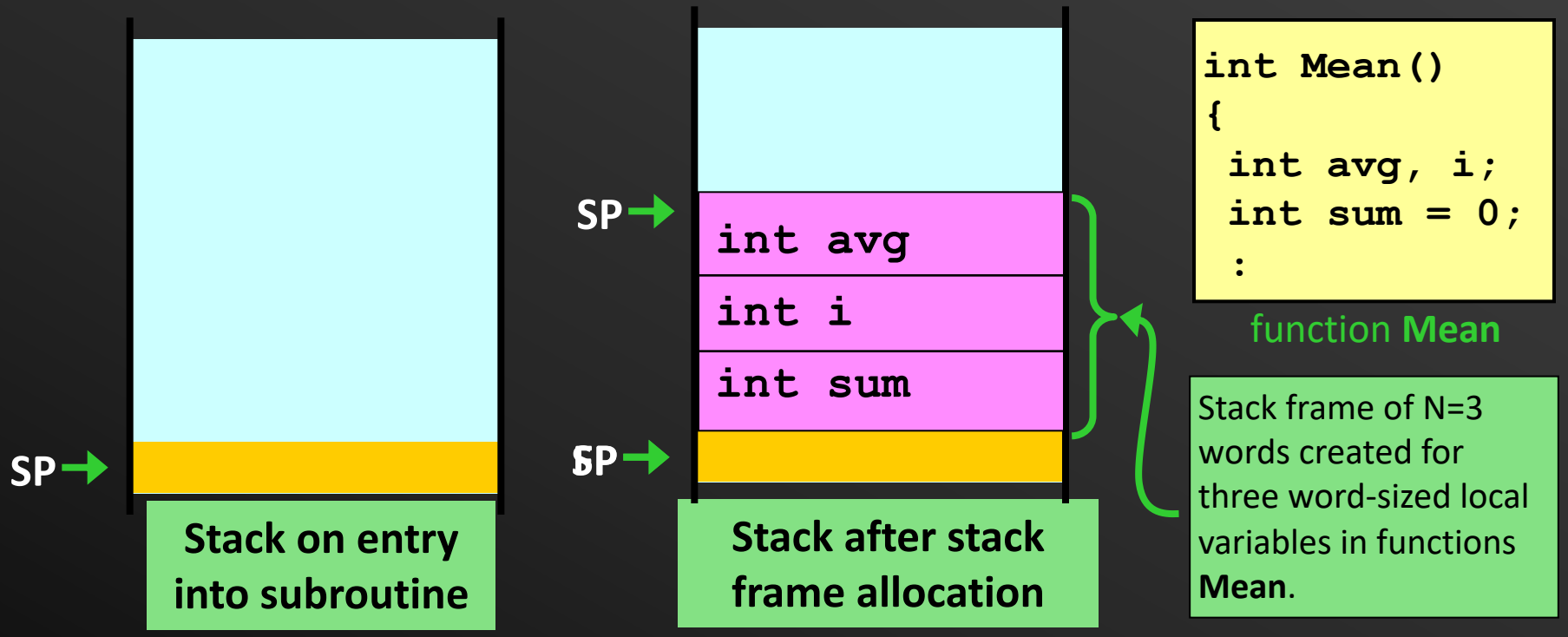
- **Characteristics of a good software module.**
- Loose coupling – **data** within module is entirely **independent** of other modules (local variables).

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**.

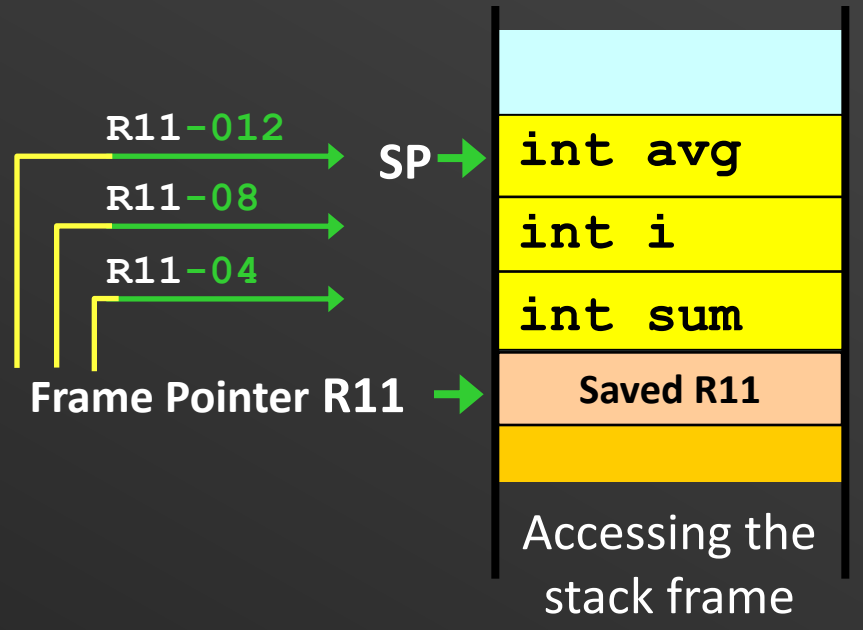
Stack Frame

- 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**



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.



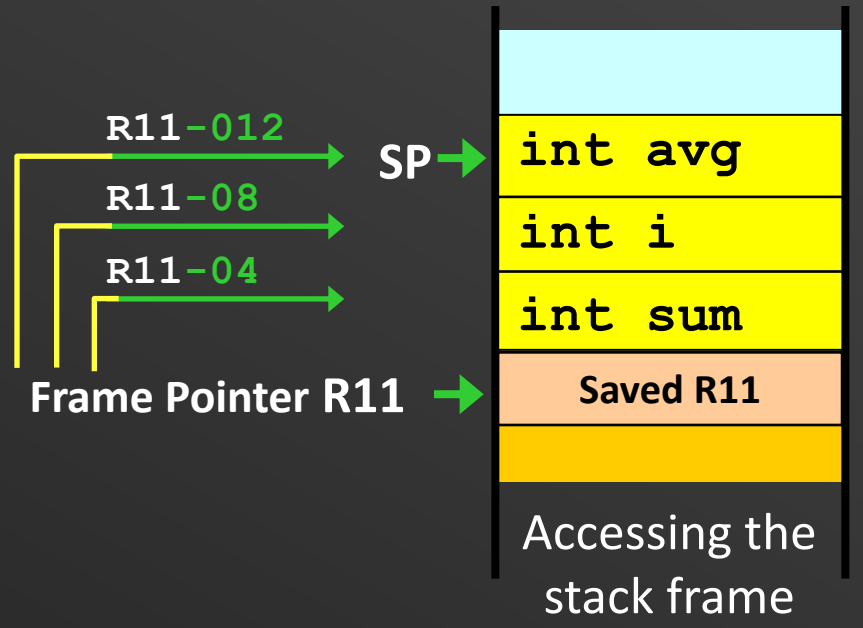
- When exiting the subroutine, the stack frame can be destroyed by adding **4N+4** to **SP** and copying the saved **R11** value on the stack back into **R11**.

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**.

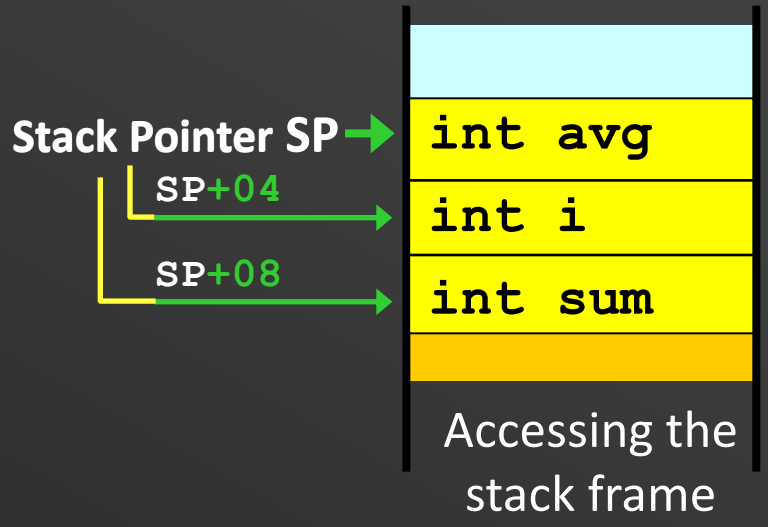
For N=3

```
ADD SP, SP, #16

LDR R11,[R11]
```



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**.



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**.

Chapter 6: Modular Programming-Cont

Mohamed M. Sabry Aly

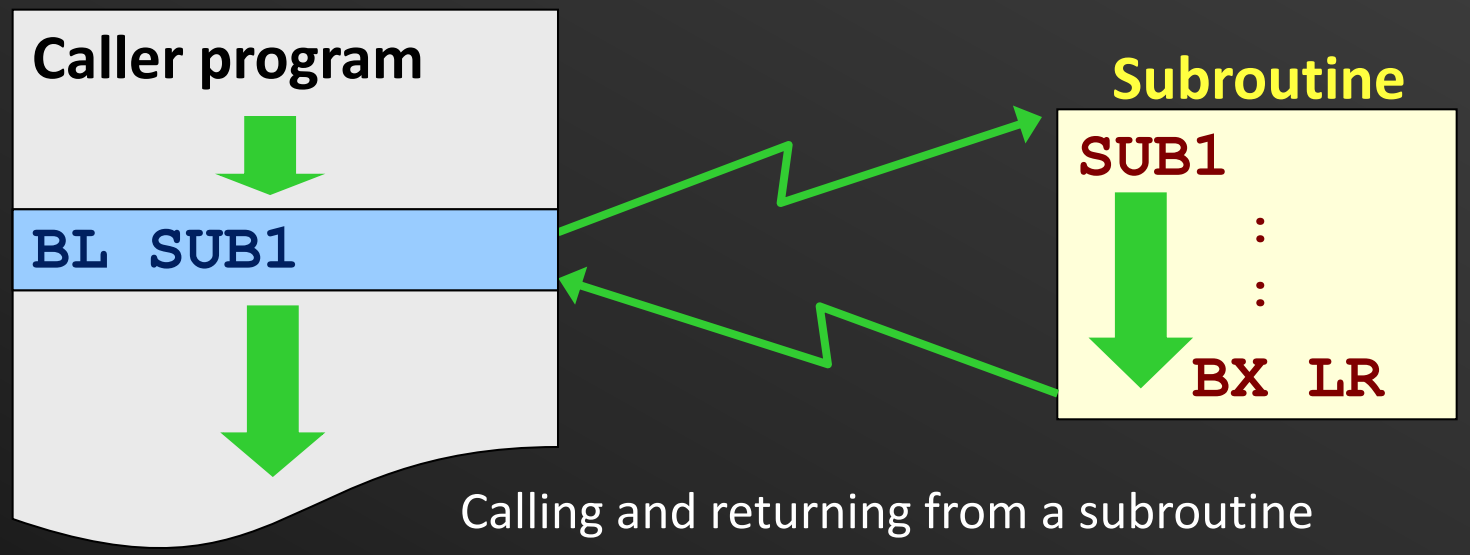
N4-02c-92

Learning Objectives

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

Subroutines (Recap)

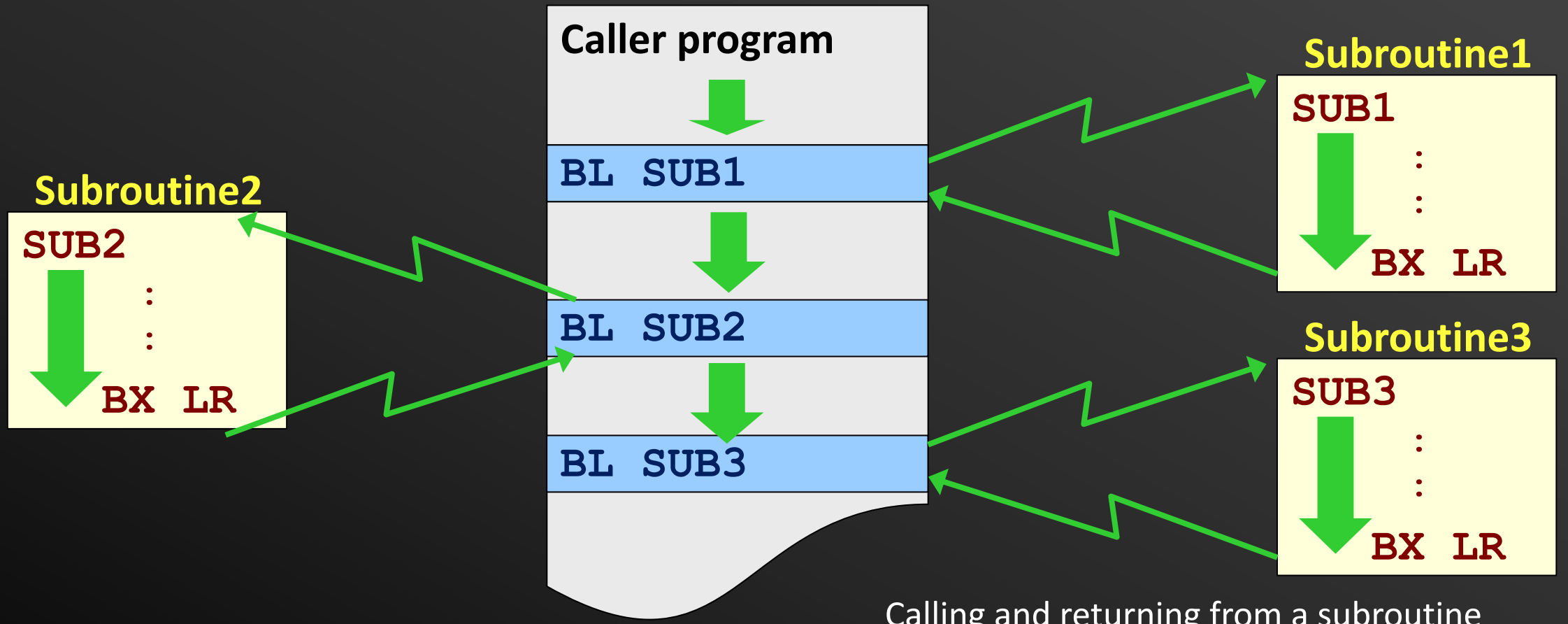
- Modules (e.g., functions in C) are implemented as **subroutines**
- Subroutine can be called from various parts of the program
- Caller and callee
  - Caller: the program that calls subroutine (SUB1)
  - Callee: subroutine (SUB1)



Calling and returning from a subroutine

Multiple Subroutines

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

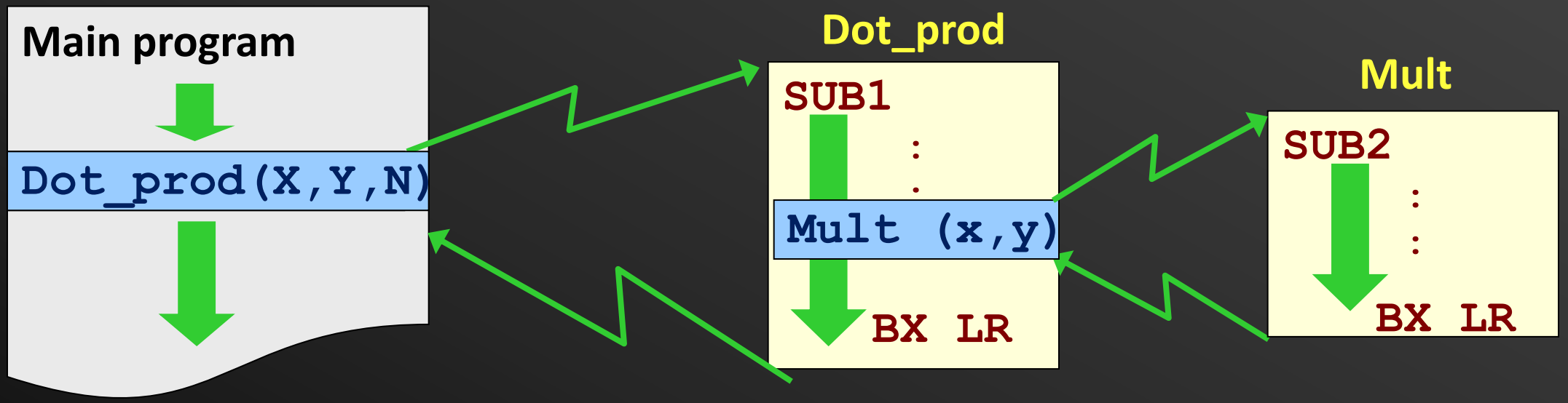


Calling and returning from a subroutine

Do all applications have just 1-level functional call?

Example: Dot product of two arrays

- A subroutine will call another subroutine



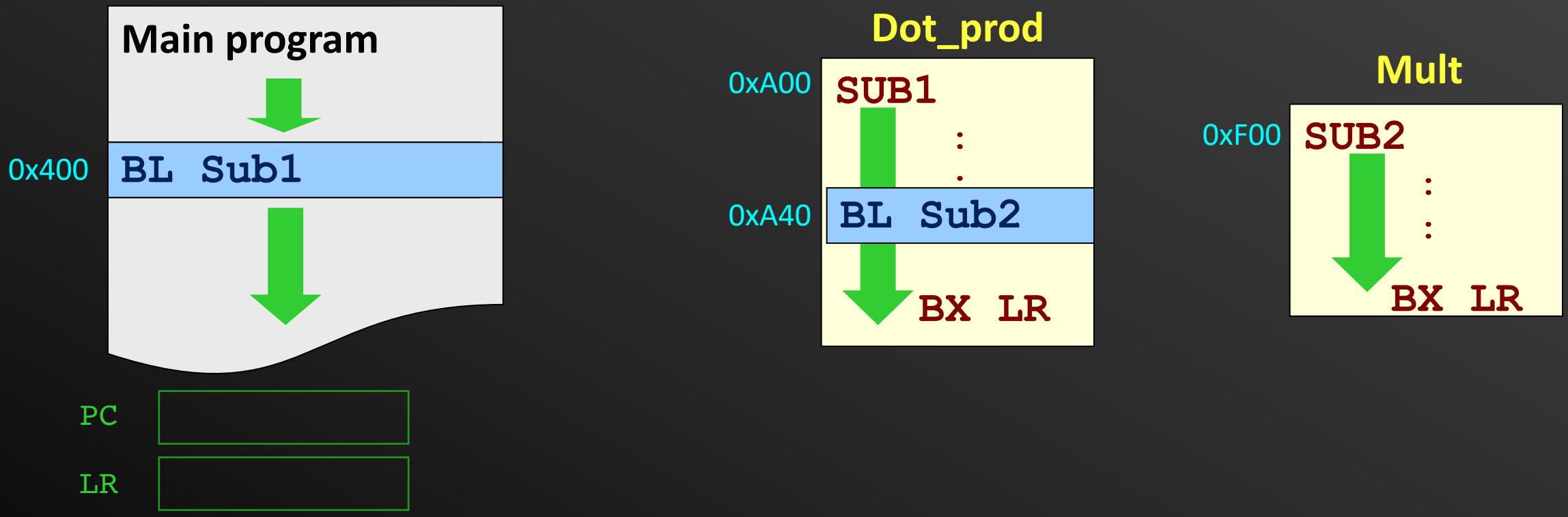
Example: Dot product of two arrays

- A subroutine will call another subroutine

How to ensure right subroutine branch and return?

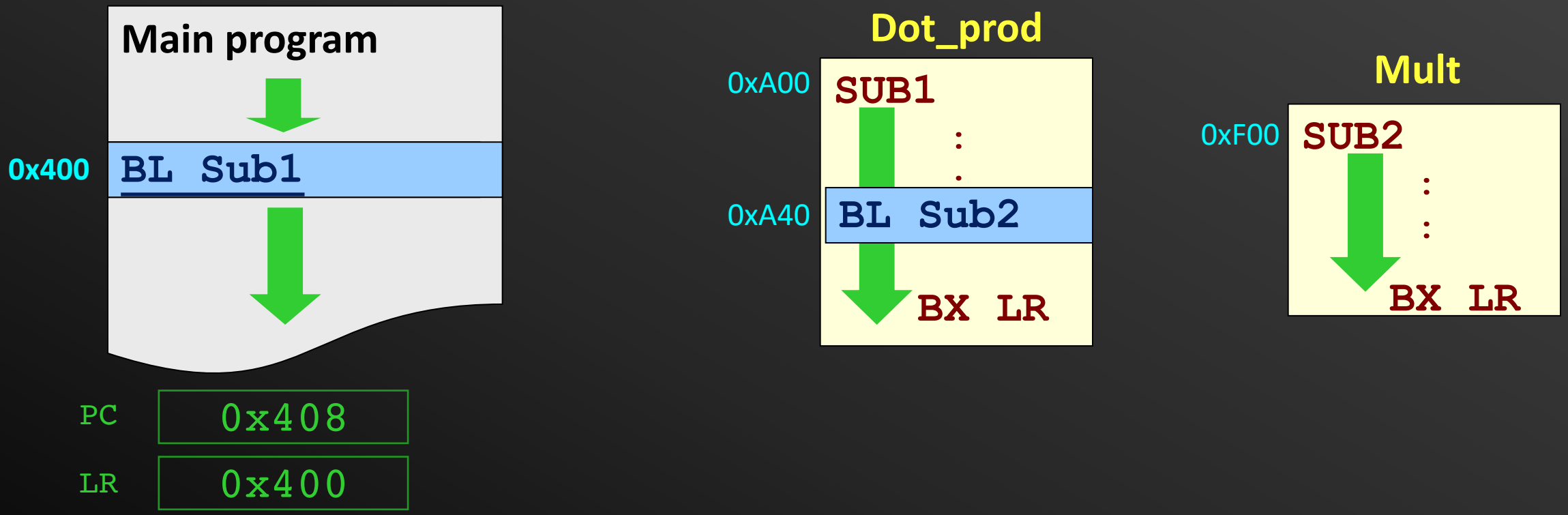
Example: Dot product of two arrays

- A subroutine will call another subroutine



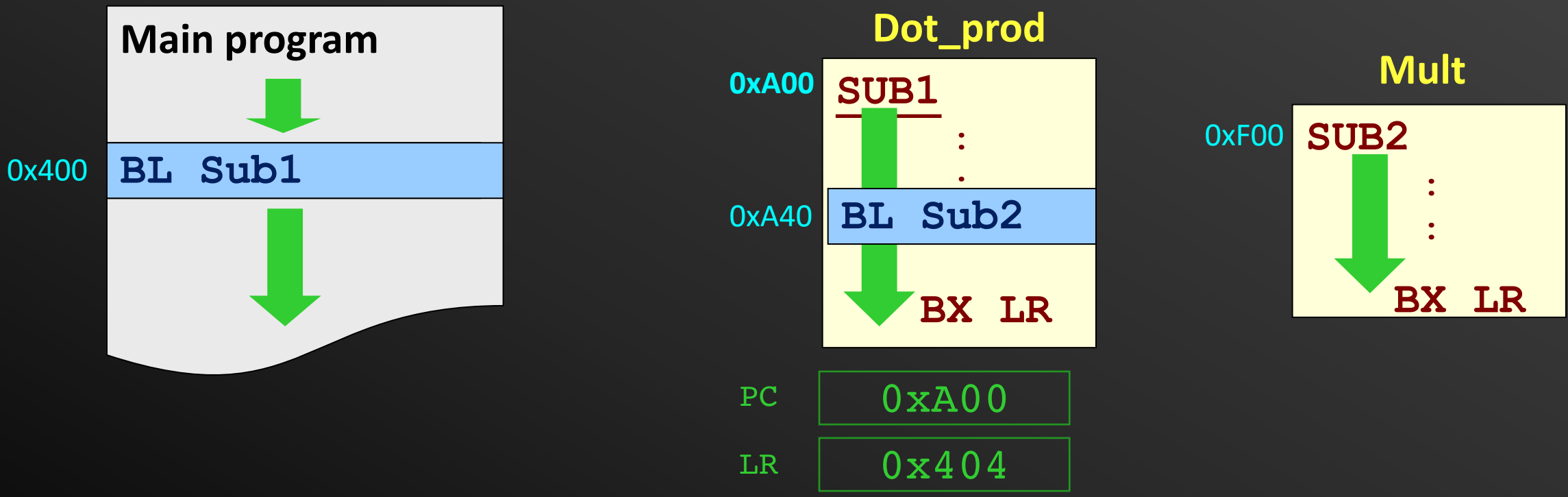
Example: Dot product of two arrays

- A subroutine will call another subroutine



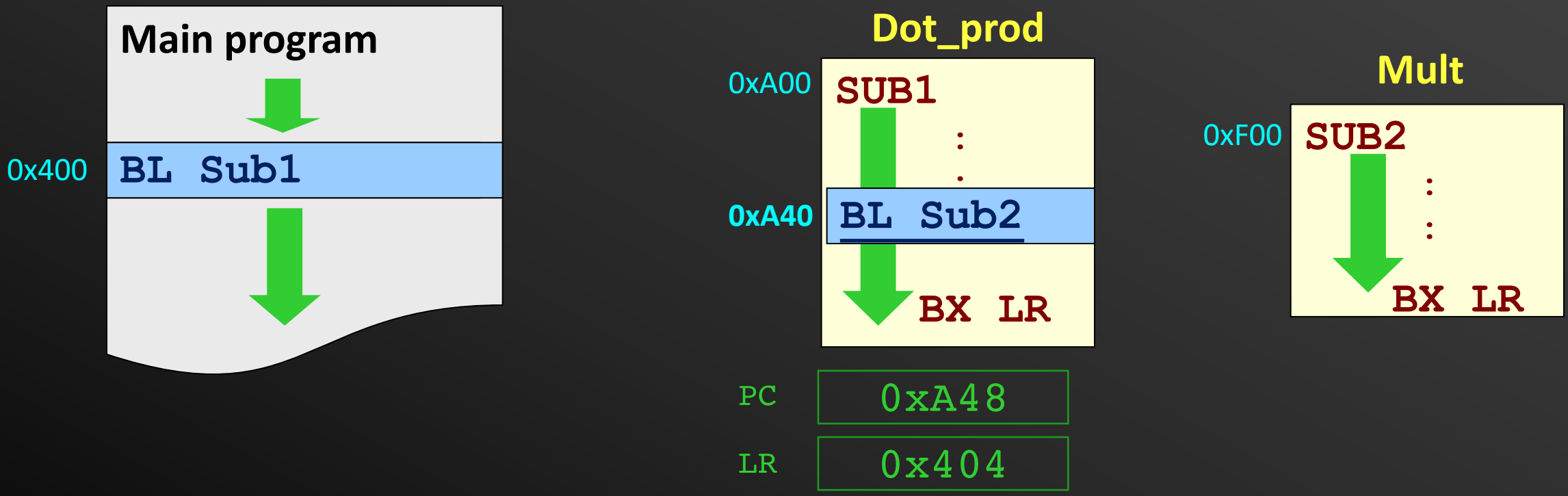
Example: Dot product of two arrays

- A subroutine will call another subroutine



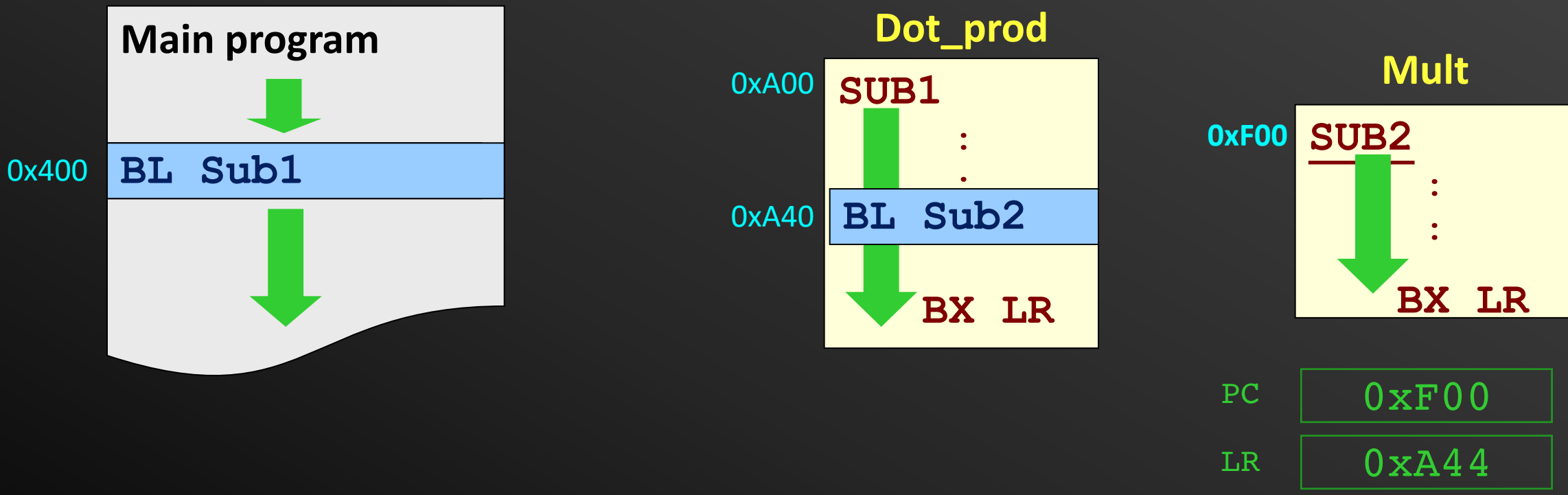
Example: Dot product of two arrays

- A subroutine will call another subroutine



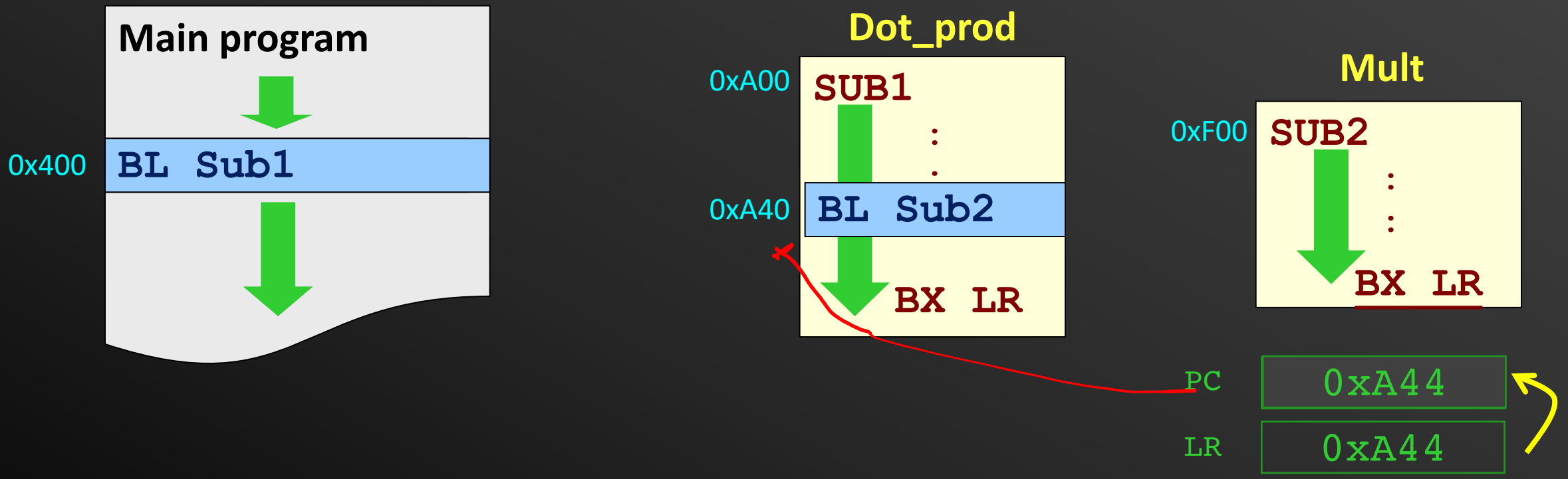
Example: Dot product of two arrays

- A subroutine will call another subroutine



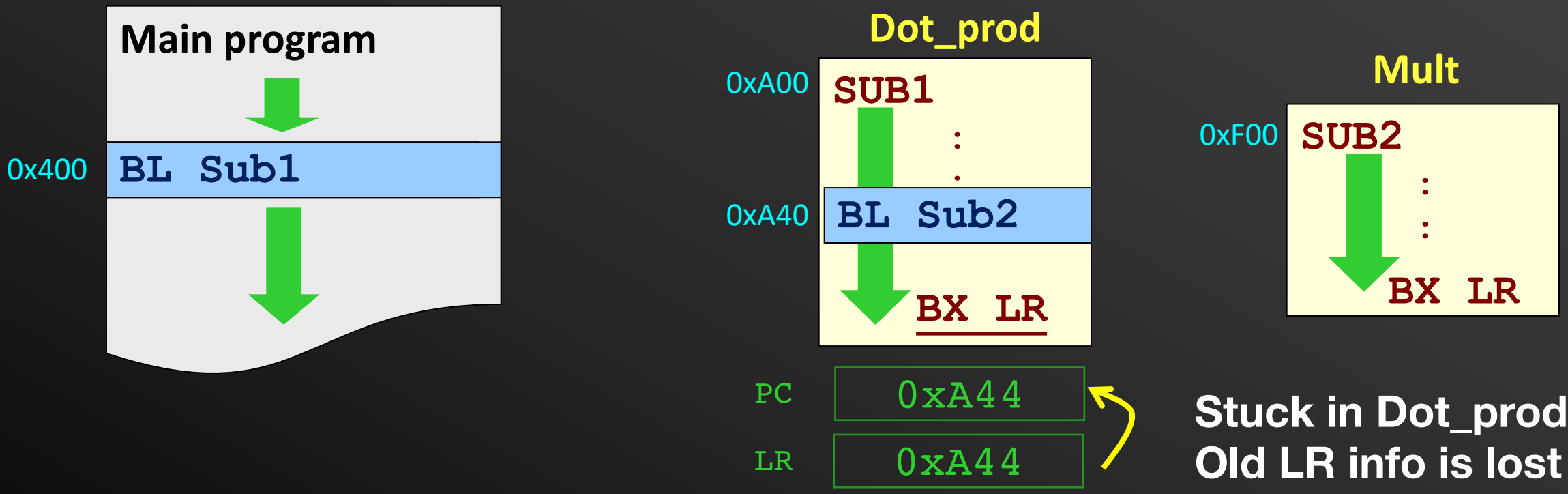
Example: Dot product of two arrays

- A subroutine will call another subroutine



Example: Dot product of two arrays

- A subroutine will call another subroutine



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

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

Multiplication subroutine

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

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

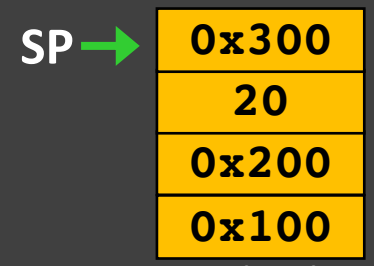
```
        MOV     PC, LR          ; same as bx lr
```

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
```

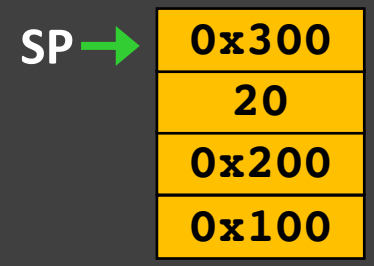
The Dot product Subroutine



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


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

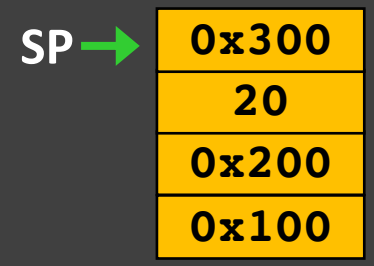
The Dot product Subroutine



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
```

The Dot product Subroutine

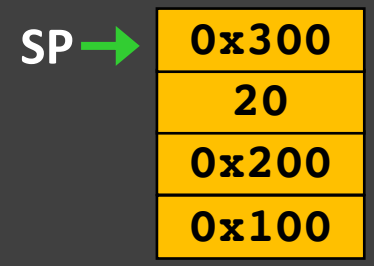


System stack when entering DotProd

```
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 (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
```

```
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
```

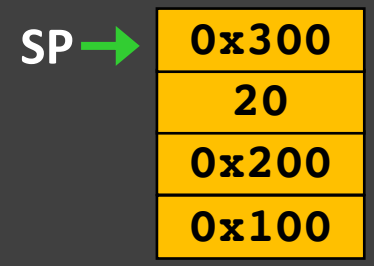
The Dot product Subroutine



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 (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
```

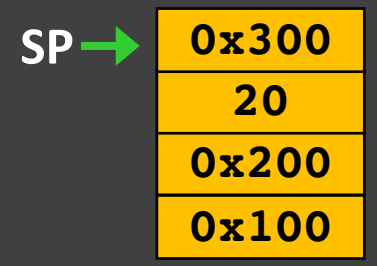
The Dot product Subroutine



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 (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
```

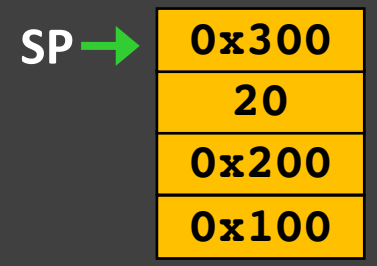
The Dot product Subroutine



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 (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
```

The Dot product Subroutine



System stack when entering `DotProd`

✓ Straight forward, no other impact on the code

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

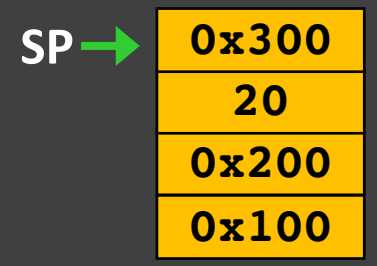
✗ Different instruction structure

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

The Dot product Subroutine



System stack when entering DotProd

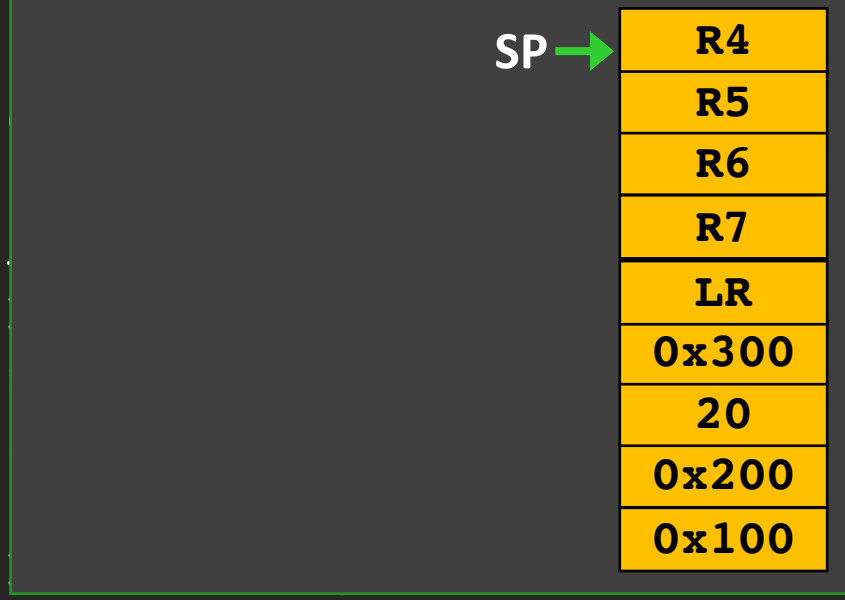
```
DotProd     STMFD       SP!,{R4-R7,LR}    ;Store regs to stack
            LDR         R4 ,[SP,#28]      ;Read location of X
            LDR         R5 ,[SP,#24]      ;Read location of Y
            LDR         R6 ,[SP,#20]      ;Read arrays length
            MOV         R7, #0            ; Clear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

            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,LR}    ; Restore registers
            MOV         PC, LR            ; same as bx lr
```

The Dot product Subroutine

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

System stack when entering DotProd

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

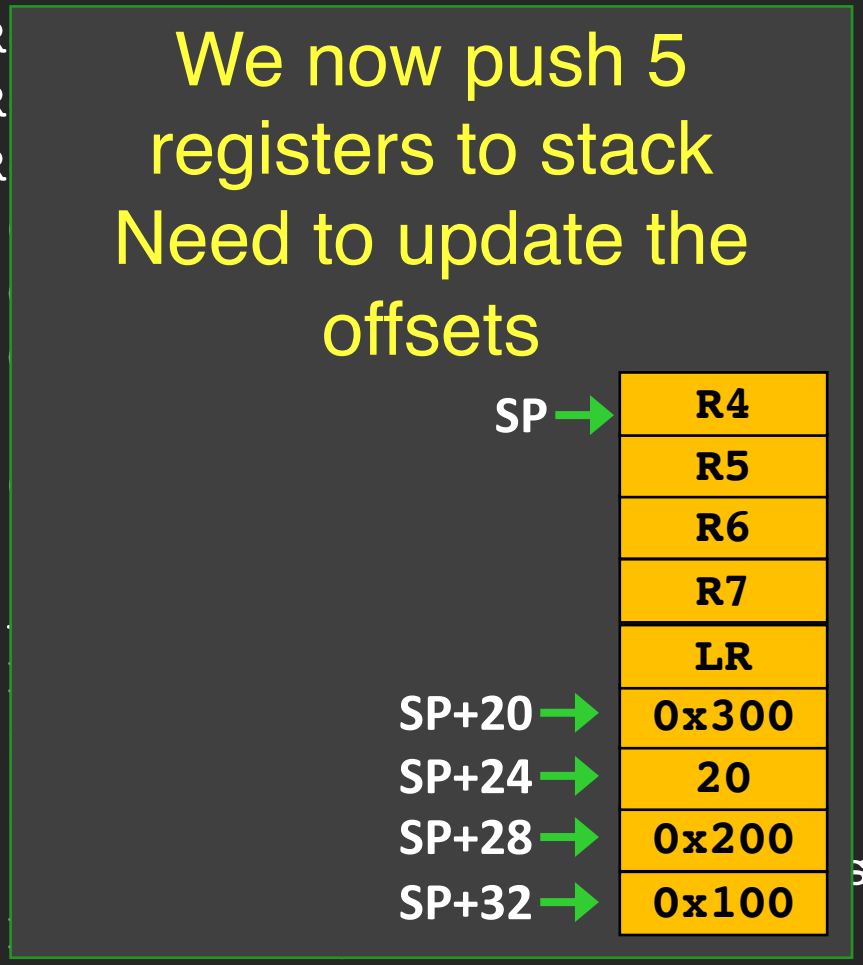


The Dot product Subroutine

```
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
```

System stack when entering DotProd

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



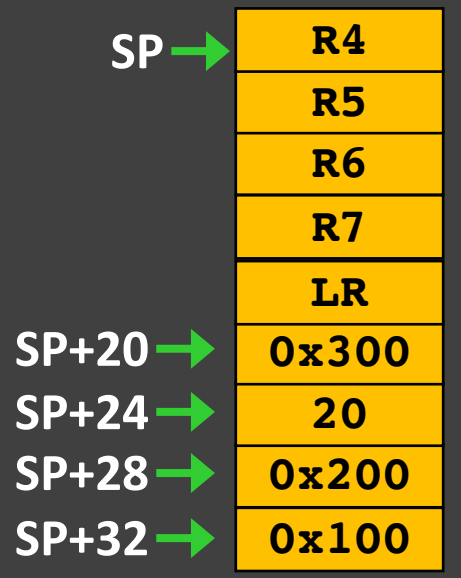
The Dot product Subroutine

```
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?

System stack when entering DotProd

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



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);
}
```

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

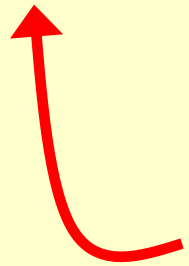
```
Recur   :                ; Subroutine Recur
        BL  Recur        ; call Recur with Recur routine
        :
        BX  LR           ; return to calling program
```

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

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
```

Example: Fibonacci Sequence

- Number sequence

  0,1,1,2,3,5,8,13,21,34,55

- Starting from index 1

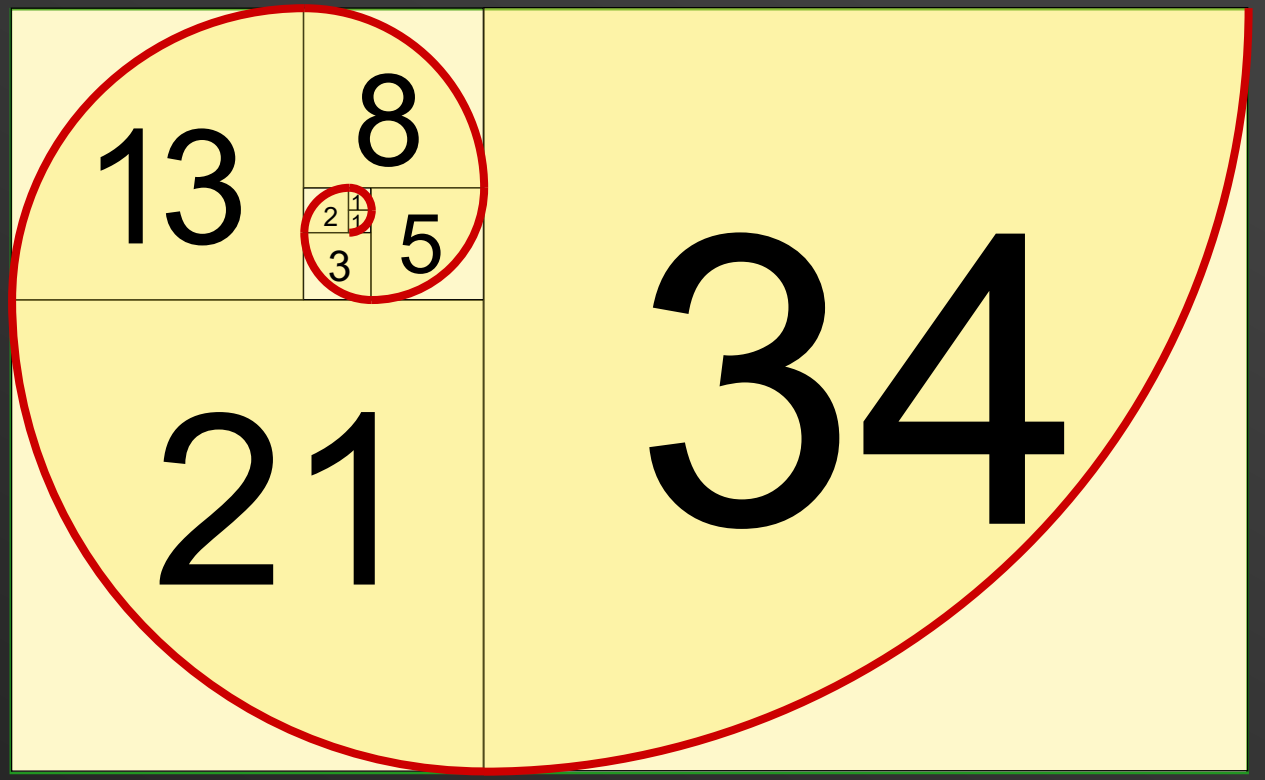
  $F(n) = F(n-1) + F(n-2)$

  $F(2) = 1$

  $F(1) = 0$

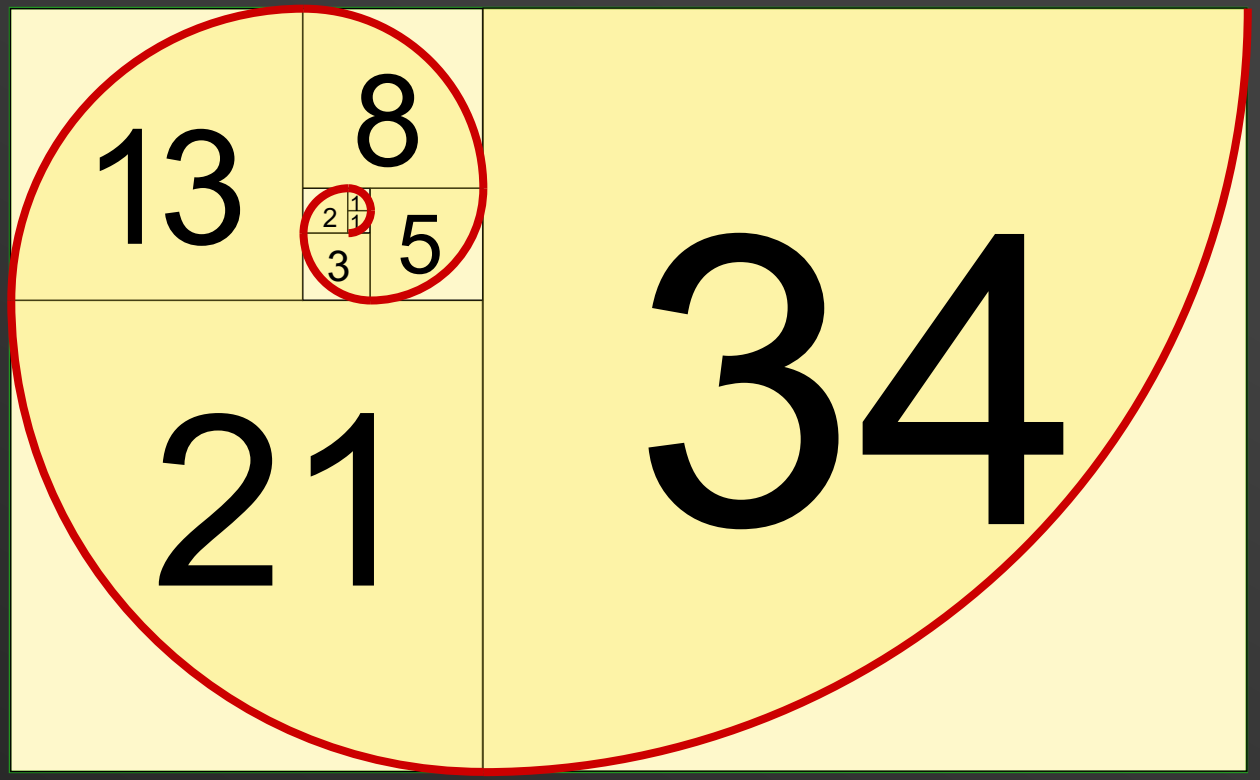
  Pass parameter in stack

  Return result in R12



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;
}
```

Example Solution

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
```

Example Solution

System stack when entering `Fib`

```
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
```

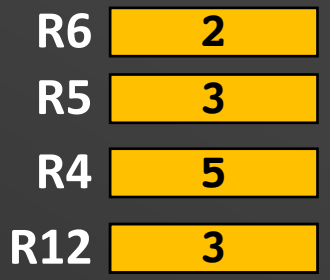
Example Solution

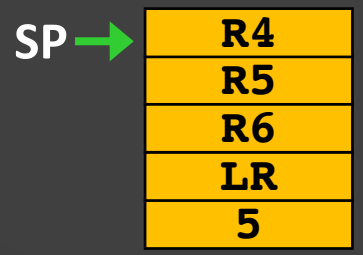

```
Fib     STMFD   SP!,{R4-R6,LR}   ;Store regs to stack
        LDR     R4 ,[SP,#16]     ;Read Value of n
        CMP     R4 ,#1           ;Compare with 1 (if n==1)
        MOVEQ   R12,#0           ;Assign result with 0
        CMP     R4 ,#2           ;Compare with 2 (if n==2)
        MOVEQ   R12,#1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

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
```

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
```

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
```

Chapter 7: Flow-control Constructs

Mohamed M. Sabry Aly

N4-02c-92

Learning Objectives

- Be able to convert a high level IF and IF-ELSE construct to its low-level equivalent.

- Describe compute-efficient considerations for compound AND and OR constructs.

- Describe and analyze simple branchless logic constructs.

IF Statements

- How is the **IF** construct implemented?
- **Imitating** the high-level test condition does not result in very efficient assembly-level implementation.

```
if (a > b)
  {S1}
```

Convert to assembly code

```
        CMP a,b
        BGT DoIF
        B Skip
DoIF    {S1}
Skip
```

- High-level test condition is **reversed** in assembly-level to avoid the need for an additional unconditional jump.

```
if (a > b)
  {S1}
```

Convert to assembly code

```
        CMP a,b
        BLE Skip
        {S1}
Skip
```

Note: The reverse condition test of **HS** is **LO**, reverse of **LT** is **GE**

Find Largest Number

```
        MOV   R0,#0x100      ;setup pointer to first array element
        MOV   R1,#9
        LDR   R3,[R0]
Loop    LDR   R2,[R0,#4
        CMP   R2,R3          ;compare R3 and R2 (i.e. R2-R3)
        BLO   Skip           ;branch if R2 <  current max (i.e. R3)
        MOV   R3,R2          ;update current max. in R3 with R2
Skip    SUBS  R1,R1,#1       ;decrement 1 from counter register
        BNE   Loop           ;jump back to Loop if not zero
```

```
if R2 >= R3   // R2 larger or equal to current max
{
   R3 = R2;   // then update R3 with new max R2
}
```

R0 = Address pointer for current array element.
R1 = Loop counter register
R2 = Temporary register holding current no.
R3 = Current maximum value (i.e. the result).

Can We Avoid Reversion?

```
        MOV   R0,#0x100      ;setup pointer to first array element
        MOV   R1,#9
        LDR   R3,[R0]
Loop    LDR   R2,[R0,#4
        CMP   R2,R3
        BLO   Skip
        MOV   R3,R2
Skip    SUBS  R1,R1,#1
        JNE   Loop
```

```
if R2 >= R3   // R2 larger or equal to current max
{
   R3 = R2;   // then update R3 with new max R2
}
```

```
        CMP     R2,R3
        BHS     Assign
        SUBS    R1,R1,#1
        BNE     Loop
Assign  MOV     R3,R2
        SUBS    R1,R1,#1
        BNE     Loop
```

Redundant

Conditional Execution

- In the 32-bit ARM ISA, instructions can be conditionally executed based on the CC flags.
- Consider the following 32-bit ARM code segment.

```
; C code
if (r0 == 1)
 r1 = 3;
```

```
        CMP   r0, #1        ; set CC based on r0 –1
        BNE   Skip          ; if (r0 == 1)
        MOV   r1, #3        ; then { r1 := 3}
SKIP      ......            ;
```

- It can be replaced using conditional execution instructions.

```
        CMP    r0, #1       ; if (r0 == 1)
        MOVEQ  r1, #3       ; then { r1 := 3}
SKIP    ......              ;
```

Largest with Conditional Execution

- Significant reduction in Code size

```
        MOV   R0,#0x100      ;setup pointer to first array element
        MOV   R1,#9
        LDR   R3,[R0]
Loop    LDR    R2,[R0,#
        CMP    R2,R3         ;compare R3 and R2 (i.e. R2-R3)
        MOVGE  R3,R2         ;update current max. in R3 with R2
        SUBS   R1,R1,#1      ;decrement 1 from counter register
        JNE    Loop          ;jump back to Loop if not zero
```

```
if R2 >= R3   // R2 larger or equal to current max
{
   R3 = R2;   // then update R3 with new max R2
}
```

Does Conditional Execution work with Just 1 Instruction?

No, as long as the status registers are not altered from the comparison instruction

Find Largest Number, and Store Its Index

```
R3= X[0];
R4=0;
R0= &X[0]
For (R1=9;R1>0;R1--){
        R0+=4;R2=X[R0];
       if(R2>=R3){
              R3=R2;
              R4=10-R1;
       }
}
```

No conditional execution

```
        MOV   R0,#0x100      ;setup pointer to first array element
        MOV   R1,#9          ;load 9 into counter register
        MOV   R4,#0          ;load 0 into index_max register
        LDR   R3,[R0]        ; 1st no. in array is current max
Loop    LDR   R2,[R0,#4]!    ;get next no. in array
        CMP   R2,R3          ;compare R3 and R2 (i.e. R2-R3)
        BLO   Skip           ;branch if R2 <  current max (i.e.
        MOV   R3,R2          ;update current max. in R3 with R2
        RSUB  R4,R1,#10      ;Store the index in R4
Skip    SUBS  R1,R1,#1       ;decrement 1 from counter register
        BNE   Loop           ;jump back to Loop if not zero
```

Find Largest Number, and Store Its Index

```
R3= X[0];
R4=0;
R0= &X[0]
For (R1=9;R1>0;R1--){
        R0+=4;R2=X[R0];
       if(R2>=R3){
              R3=R2;
              R4=10-R1;
       }
}
```

With conditional execution

```
        MOV   R0,#0x100      ;setup pointer to first array element
        MOV   R1,#9          ;load 9 into counter register
        MOV   R4,#0          ;load 0 into index_max register
        LDR   R3,[R0]        ; 1st no. in array is current max
Loop    LDR     R2,[R0,#4]!  ;get next no. in array
        CMP     R2,R3        ;compare R3 and R2 (i.e. R2-R3)
        MOVGE   R3,R2        ;update current max. in R3 with R2
        RSUBGE  R4,R1,#10    ;Store the index in R4
        SUBS    R1,R1,#1     ;decrement 1 from counter register
        BNE   Loop           ;jump back to Loop if not zero
```

IF-ELSE Statements

- How is the **IF-ELSE** construct implemented?
- Combination of conditional and unconditional jumps used for the **IF-ELSE** construct.
- Reversing high-level test condition does not improve efficiency unless
- the ELSE code segment {S2} is more likely to execute.

```
        CMP a,#3
        BNE DoElse
        {S1}
        B Skip
DoElse  {S2}
Skip    :
```

Reversing test condition

```
if (a == 3)
   {S1}
else
   {S2}
```

Convert to assembly code

```
        CMP a,#3
        BEQ DoIf
        {S2}
        B Skip
DoIf    {S1}
Skip    :
```

IF-ELSE implementation in low-level assembly

Conditional Execution for IF-ELSE

- Higher efficient coding
- In the shown example, Skip will always be the 4th instruction

```
; C code
if (r0 == 1)
 r1 = 3;
else
 r1 = 5;
```

```
        CMP   r0, #1        ; set CC based on r0 –1
        BNE   ELSE          ; if (r0 == 1)
        MOV   r1, #3        ; then { r1 := 3}
        B     SKIP          ; skip over else code seg
ELSE    MOV   r1, #5        ; else  { r1 := 5}
SKIP      ......            ;
```

- With conditional execution

```
        CMP  r0, #1         ; if (r0 == 1)
        MOVEQ r1, #3        ; then { r1 := 3}
        MOVNE r1, #5        ; else  { r1 := 5}
SKIP    ......              ;
```

Compound AND Conditions

- How are compound AND conditions handled?
- Logical AND can bind multiple basic relational conditions.

e.g.

```
if ((a==b) && (b>0)) {S1}
```

order of compound AND test

- Compilers resolve compound conditions into simpler ones.

```
    if (a!=b) then Skip
    if (b<=0) then Skip
       {S1}
Skip    :
```

- Elementary conditions bound by the logical AND are tested from **left-to-right**, in the order given in the C program.
- The first false condition means the remaining conditions are not computed. This is called the **fast Boolean operation**.
- Keep the **least likely** condition **leftmost** in your program for more efficient execution.

Compound Condition Example

- Not all cases will be executed correctly

```
if ((a==b) && (b>0)) {S1}
```

What if a>b and b>0

```
CMP     R1,R2  ;R1<-a, R2<-b
CMPEQ   R2,#0
XXXGT   XXXX   ; S1
```

Compound OR Conditions

- How are compound OR conditions handled?

e.g.

```
if ((a == 1) || (a == 2)) {S1}
```

- Most compilers eliminate an unconditional jump at the end of the compound OR series by **reversing** the **last conditional** test.

```
        if (a == 1) then DoIf
        if (a != 2) then Skip
DoIf    {S1}
Skip      :
```

- The conditional test that is **most likely** to be **true** should be kept leftmost.

Compound Condition Example

```
if ((a == 1) || (a == 2)) {S1}
```

With conditional assignment
S1 needs to be replicated.
Not suitable for multiple instructions

```
CMP     R1,#1  ;R1<-a
XXXEQ   XXXX   ;  S1
CMPNE   R1,#2
XXXEQ   XXXX   ;  S1
```

A more holistic approach

```
        CMP     R1,#1
        BEQ     doIF
        CMPNE   R1,#2
doIF    XXXEQ   XXXX   ;  S1
```

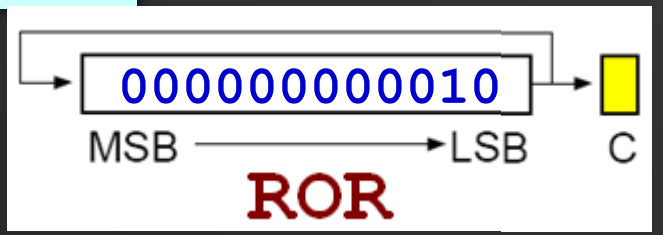
Branchless Logic

- Branchless logic avoid using conditional jump instructions when implementing logical constructs.
- `Bcc` instructions may result in costly **flushing** operations when the wrong next instruction is pre-fetched into the CPU's pipeline.
- How is branchless logic implemented?
- **Exploit arithmetic relationship** to transform the test condition into the corresponding desired outcome. Can only be applied in special cases and desired outcomes are usually Boolean values.

e.g.

```
if ((X & 2) == 2)
  X = true;
else
  X = false;
```

```
AND R1,R1,#0x02
ROR R1,R1,#1
```



- **Conditional execution** can be used to avoid branching.

Summary

- The **IF** and **ELSE** constructs are implemented using one or more conditional jump (**`Bcc`**).
  - Using the **reverse** conditional test can help the IF construct execute more efficiently.
- **Conditional execution in ARM** greatly improves flow-control efficiency.

- Sequencing the **least likely** or **most likely** conditional test can help improve execution speed of compound **AND** and **OR** respectively .
- **Branchless logic** and **conditional execution** techniques can help keep the CPU pipeline efficient by maintaining **sequential** execution.

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

Chapter 7: Flow-control Constructs

Mohamed M. Sabry Aly

N4-02c-92

Learning Objectives

- Describe how SWITCH constructs can be implemented efficiently for consecutive narrow and random wide cases.

- Contrast the implementations of pre and post-test loop constructs like WHILE, DO-WHILE and FOR.

Switch Statement

- How is the **SWITCH** construct implemented?
- The assembly code produced varies between compilers and depends on the nature and range of the case values.
- Two different SWITCH scenarios are examined:

```
switch(x)  {
 case 0 : {S0};
          break;

 case 1 : {S1};
          break

 case 2 : {S2};
          break;

 case 3 : {S3};
}
```

Running values narrow range

```
switch(x)  {
 case 1    : {S0};
             break;

 case 10   : {S1};
             break

 case 100  : {S2};
             break;

 case 1000 : {S3};
}
```

Random values wide range

Switch – Running & Narrow Values

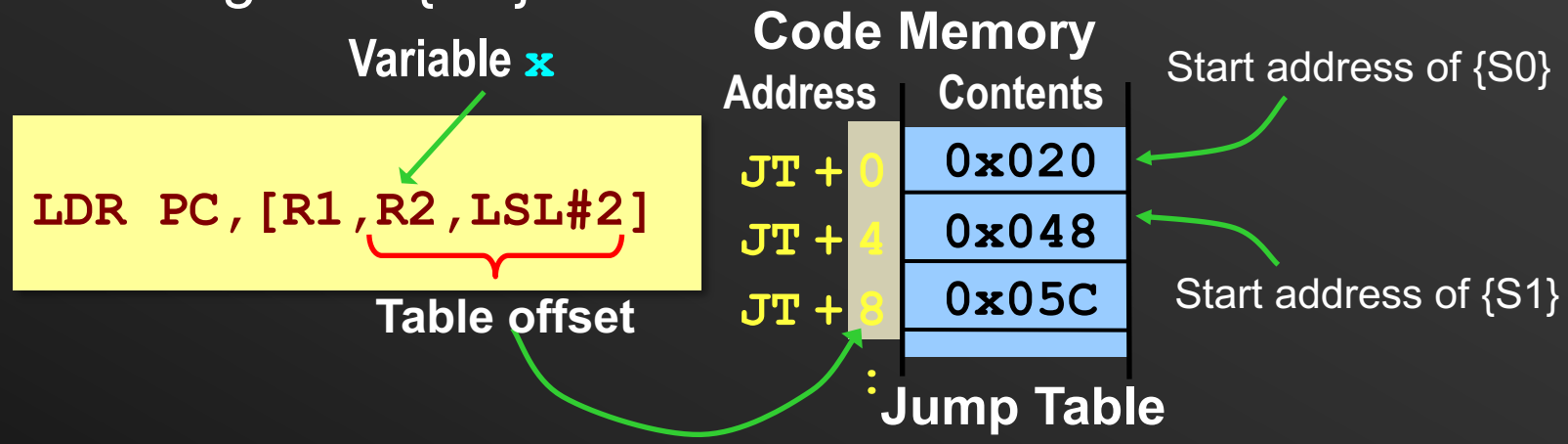
```
switch(x)
{
case 0:
 {S0};
 break;

case 1:
 {S1};
 break;

case 2:
 {S2};
 break;

case 3:
 {S3};
}
```

Switch – Running & Narrow Values

- If cases are consecutive narrow range values, a **Jump Table** is used to avoid testing each case in turn.

```
switch(x)
{
case 0:
  {S0};
 break;

case 1:
  {S1};
 break;

case 2:
 {S2};
 break;

case 3:
 {S3};
}
```

- Jump table contains list of **start addresses** for the code segments that is associated with each case values.
- Var **x** on which the switch is decided, acts as an **offset** into the table to access the corresponding start address.
- Start address is loaded into **PC** to execute the required code segment {S**x**}.



Note: Time taken is on average less than the equivalent if-else-if cascade and is independent of number of cases in the switch construct.

Jump Table Example

```
switch(x)
{
case 0:
 G='F';
 break;

case 1:
 G='D';
 break;

case 2:
 G='C';
 break;

case 3:
 G='B';
}
```

Cascaded if-else

```
      LDR R1, [R2]
      CMP R1, #0
      BEQ C1
      CMP R1, #1
      BEQ C2
      CMP R1, #2
      BEQ C3
      MOV R0, #66 ;'B'
RES   STR R0, [R2,#4]
      .
      .
      .
C1    MOV R0, #70 ;'F'
      B RES
C2    MOV R0, #68 ;'D'
      B RES
C3    MOV R0, #67 ;'C'
      B RES
```

Jump Table

```
      LDR R1, [R2]
      ADR R3, TBL
      LDR PC, [R3,R1, LSL #2]
RES   STR R0, [R2,#4]
      .
      .
0xA00 C1   MOV R0, #70 ;'F'
      B RES
C2    MOV R0, #68 ;'D'
      B RES
C3    MOV R0, #67 ;'C'
      B RES
```

Address	Contents
TBL + 0	0xA00
TBL + 4	0xA08
TBL + 8	0xA10
:	

Jump Table

Jump Table Example (Conditional Execution)

```
switch(x)
{
case 0:
 G='F';
 break;

case 1:
 G='D';
 break;

case 2:
 G='C';
 break;

case 3:
 G='B';
}
```

Cascaded if-else

```
      LDR R1, [R2]
      CMP R1, #0
      MOVEQ R0,#70
      CMP R1, #1
      MOVEQ R0,#68
      CMP R1, #2
      MOVEQ R0,#67
      CMP R1, #3
      MOVEQ R0, #66 ;'B'
RES   STR R0, [R2,#4]
      .
      .
      .
```

Jump Table

```
      LDR R1, [R2]
      ADR R3, TBL
      LDR PC, [R3,R1, LSL #2]
RES   STR R0, [R2,#4]
      .
      .
0xA00 C1   MOV R0, #70 ;'F'
      B RES
C2    MOV R0, #68 ;'D'
      B RES
C3    MOV R0, #67 ;'C'
      B RES
```

Address	Contents
TBL + 0	0xA00
TBL + 4	0xA08
TBL + 8	0xA10
:	

Jump Table

Jump Table Example (Default)

```
switch(x)
{
case 0:
 G='F';
 break;

case 1:
 G='D';
 break;

case 2:
 G='C';
 break;

case 3:
 G='B';
Default:
G='X';
}
```

Cascaded if-else

```
        LDR R1, [R2]
        CMP R1, #0
        BEQ C1
        CMP R1, #1
        BEQ C2
        CMP R1, #2
        BEQ C3
        CMP R1,#3
        BEQ C4
        MOV R0, #88 ;'X'
RES     STR R0,[R2,#4]
        .
        .
C1      MOV R0, #70 ;'F'
        B RES
C2      MOV R0, #68 ;'D'
        B RES
C3      MOV R0, #67 ;'C'
        B RES
C4      MOV R0, #66; 'B'
        B RES
```

Jump Table

```
        LDR R1, [R2]
        MOV R0, #88; 'X'
        ADR R3, TBL
        LDR PC, [R3,R1, LSL #2]
RES     STR R0, [R2,#4]
        .
0xA00 C1  MOV R0, #70 ;'F'
        B RES
C2      MOV R0, #68 ;'D'
        B RES
C3      MOV R0, #67 ;'C'
        B RES
```

Address	Contents
TBL + 0	0xA00
TBL + 4	0xA08
TBL + 8	0xA10
:	

Jump Table

Jump Table Example (Default)

```
switch(x)
{
case 0:
 G='F';
 break;

case 1:
 G='D';
 break;

case 2:
 G='C';
 break;

case 3:
 G='B';
Default:
G='X';
}
```

Cascaded if-else

```
        LDR R1, [R2]
        CMP R1, #0
        BEQ C1
        CMP R1, #1
        BEQ C2
        CMP R1, #2
        BEQ C3
        CMP R1,#3
        BEQ C4
        MOV R0, #88 ;'X'
RES     STR R0,[R2,#4]
        .
        .
C1      MOV R0, #70 ;'F'
        B RES
C2      MOV R0, #68 ;'D'
        B RES
C3      MOV R0, #67 ;'C'
        B RES
C4      MOV R0, #66; 'B'
        B RES
```

Jump Table

```
        LDR R1, [R2]
        MOV R0, #88; 'X'
        CMP R1, #3
        BGT RES
        ADR R3, TBL
        LDR PC, [R3,R1, LSL #2]
0xA00 RES  STR R0, [R2,#4]
        .
C1      MOV R0, #70 ;'F'
        B RES
C2      MOV R0, #68 ;'D'
        B RES
C3      MOV R0, #67 ;'C'
        B RES
```

Address	Contents
TBL + 0	0xA00
TBL + 4	0xA08
TBL + 8	0xA10
:	

Jump Table

Example

- Create a subroutine that estimates the grading statistics of grades
  - Input: array of grades: ‘A’,’B’,’C’,’D’, passes in characters
  - The array terminates with a grade of 0
- Calculate statistics
  - Number of top grading marks
  - Number of B-grading marks
  - Other passing grades
- Pass address of grade array in stack, return three parameters also in stack

Example

- Create a subroutine
  grades
  - Input: array of grade
  - The array terminates
- Calculate statistics
  - Number of top gradi
  - Number of B-gradin
  - Other passing grade

```
Cal_grades (char* arr, int* top, int* Bgrading, int *other)
{
        *top=0;
        *Bgrading=0;
        *other=0;
        int done =0;
        int index=0;
        do{
                char val=arr[index++];
                switch (val):
                {
                   case ('A'):
                         *top++;
                         break;
                   case ('B'):
                         *Bgrading++;
                         break;
                   case ('C'):
                         *other++;
                         break;
                   case ('D'):
                         *top++;
                         break;
                   default:
                        done=1;
        }while (done==0);
}
```

```
Cal_grades      STMFD SP!,{R4-R9}
                ADR   R9 ,Base
                MOV   R4 ,#0
                MOV   R5 ,#0
                MOV   R6 ,#0




                LDMFD SP!,{R4-R9}
                MOV   PC, LR
```

```
Cal_grades (char* arr, int* top, int* Bgrading, int *other)
{
        *top=0;
        *Bgrading=0;
        *other=0;
        int done =0;
        int index=0;
        do{
                char val=arr[index++];
                switch (val):
                {
                   case ('A'):
                         *top++;
                         break;
                   case ('B'):
                         *Bgrading++;
                         break;
                   case ('C'):
                         *other++;
                         break;
                   case ('D'):
                         *top++;
                         break;
                   default:
                        done=1;
        }while (done==0);
}
```

```
Cal_grades      STMFD SP!,{R4-R9}
                ADR   R9 ,Base
                MOV   R4 ,#0
                MOV   R5 ,#0
                MOV   R6 ,#0
                LDR   R8,[SP,#36]
Loop            LDRB  R7,[R8],#1
                CMP   R7,#0
                BEQ   Done


Done            LDR   R8,[SP,#32]
                STR   R4,[R8]
                LDR   R8,[SP,#28]
                STR   R5,[R8]
                LDR   R8,[SP,#24]
                STR   R6,[R8]
                LDMFD SP!,{R4-R9}
                MOV   PC, LR
```

```
Cal_grades (char* arr, int* top, int* Bgrading, int *other)
{
        *top=0;
        *Bgrading=0;
        *other=0;
        int done =0;
        int index=0;
        do{
                char val=arr[index++];
                switch (val):
                {
                   case ('A'):
                         *top++;
                         break;
                   case ('B'):
                         *Bgrading++;
                         break;
                   case ('C'):
                         *other++;
                         break;
                   case ('D'):
                         *top++;
                         break;
                   default:
                              done=1;
        }while (done==0);
}
```

```
Cal_grades      STMFD SP!,{R4-R9}
                ADR   R9 ,Base
                MOV   R4 ,#0
                MOV   R5 ,#0
                MOV   R6 ,#0
                LDR   R8,[SP,#36]
Loop            LDRB  R7,[R8],#1
                CMP   R7,#0
                BEQ   Done
                SUB   R7, R7, #65
                LDR   PC,[R9,R7,LSL #2]
Return          B     Loop
Done            LDR   R8,[SP,#32]
                STR   R4,[R8]
                LDR   R8,[SP,#28]
                STR   R5,[R8]
                LDR   R8,[SP,#24]
                STR   R6,[R8]
                LDMFD SP!,{R4-R9}
                MOV   PC, LR

Cond1           ADD   R4,R4,#1
                B     Return
Cond2           ADD   R5,R5,#1
                B     Return
Cond3           ADD   R6,R6,#1
                B     Return
Cond4           ADD   R6,R6,#1
                B     Return
```

```
Cal_grades (char* arr, int* top, int* Bgrading, int *other)
{
        *top=0;
        *Bgrading=0;
        *other=0;
        int done =0;
        int index=0;
        do{
                char val=arr[index++];
                switch (val):
                {
                   case ('A'):
                          *top++;
                          break;
                   case ('B'):
                          *Bgrading++;
                          break;
                   case ('C'):
                          *other++;
                          break;
                   case ('D'):
                          *top++;
                          break;
                   default:
                                done=1;
        }while (done==0);
}
```

```
Cal_grades      STMFD SP!,{R4-R9}
                ADR   R9 ,Base
                MOV   R4 ,#0
                MOV   R5 ,#0
                MOV   R6 ,#0
                LDR   R8,[SP,#36]
Loop            LDRB  R7,[R8],#1
                CMP   R7,#0
                BEQ   Done
                SUB   R7, R7, #65
                LDR   PC,[R9,R7,LSL #2]
Return          B     Loop
Done            LDR   R8,[SP,#32]
                STR   R4,[R8]
                LDR   R8,[SP,#28]
                STR   R5,[R8]
                LDR   R8,[SP,#24]
                STR   R6,[R8]
                LDMFD SP!,{R4-R9}
                MOV   PC, LR

Cond1           ADD   R4,R4,#1
                B     Return
Cond2           ADD   R5,R5,#1
                B     Return
Cond3           ADD   R6,R6,#1
                B     Return
Cond4           ADD   R6,R6,#1
                B     Return
```

```
Cal_grades (char* arr, int* top, int* Bgrading, int *other)
{
        *top=0;
        *Bgrading=0;
        *other=0;
        int done =0;
        int index=0;
        do{
                char val=arr[index++];
                switch (val):
                {
                   case ('A'):
                         *top++;
                         break;
                   case ('B'):
                         *Bgrading++;
                         break;
                   case ('C'):
                         *other++;
                         break;
                   case ('D'):
                         *top++;
                         break;
                   default:
                        done=1;
        }while (done==0);
}
```

Switch – Random & Wide Values

- If cases are random wide range values, a **fork algorithm** is used to speed up the average search time and avoid testing every case (e.g. when **x** = 1000).
- Due to the wide value spread, the **jump table size** will be **too large**. A cascade of if-else-if comparisons is more efficient.

```
switch(x)
{
case 1:
  {S0};
 break;

case 10:
  {S1};
 break;

case 100:
 {S2};
 break;

case 1000:
 {S3};
 break;
}
```

```
if(x == 1)
  {S0};
else if(x == 10)
  {S1};
else if(x == 100)
  {S2};
else if(x == 1000)
  {S3};
```

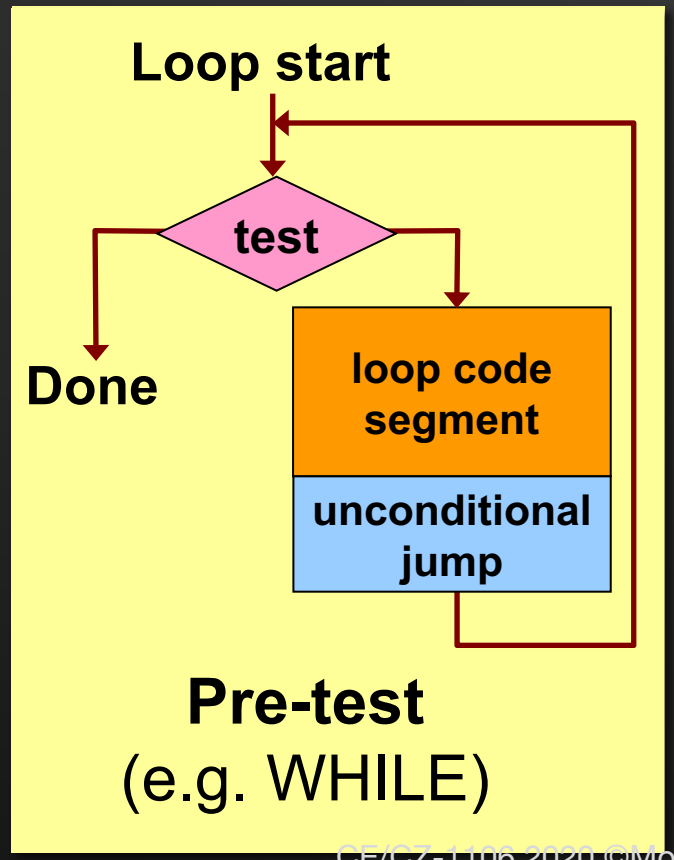
standard if-else-if implementation

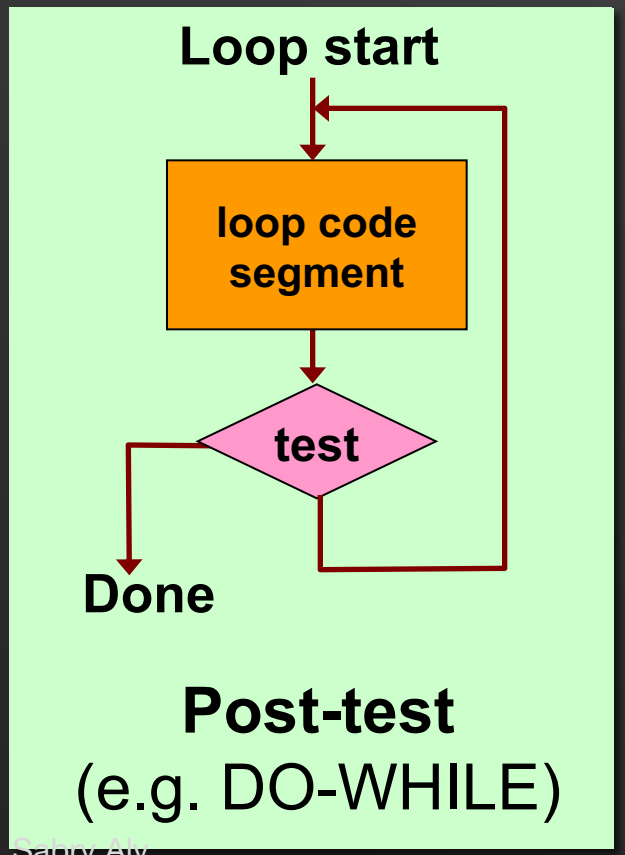
```
if(x <= 10) {
    if(x == 1)
      {S0};
    else if(x == 10)
      {S1};
}
else {
    if(x == 100)
      {S2};
    else if(x == 1000)
      {S3};
}
```

forked if-else-if implementation

Loops

- Loop constructs are distinguished by the position of their conditional test.
- Pre-test loop may **never execute** its loop code segment.
- Post-test loop executes the loop segment **at least once**.





WHILE Implementation

- Implementation of the **WHILE** loop constructs:
- This is an example of a **pre-test** loop.
- If the condition **(VarX > 0)** is false, the loop segment is not executed at all.

```
WHILE (VarX > 0)
{
    Loop segment
}
```

```
Back   CMP "VarX",#0
       BLE Exit
        :
        :   } Loop segment
        :
       B Back
Exit    :
```

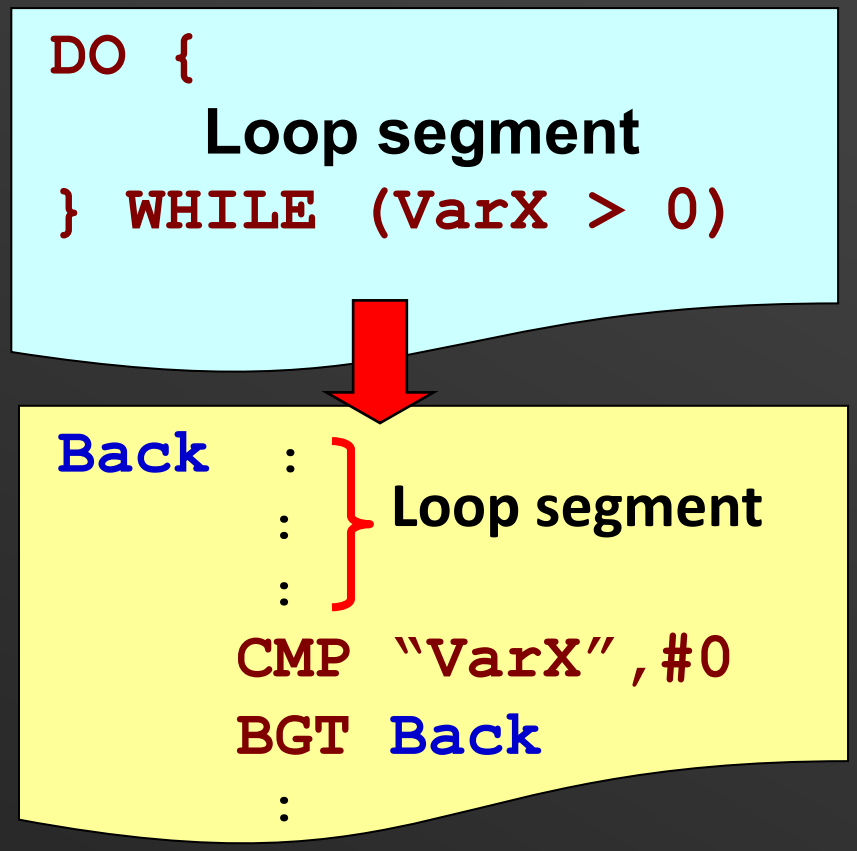
Implementation of WHILE in ARM assembly language

Note: **VarX** is variable. you will need to load it to a register.

DO-WHILE Implementation

- Implementation of the **DO-WHILE** loop constructs
- This is an example of a **post-test** loop.
- The loop segment is executed at least once before condition is tested.
- Post-test loop construct is **more efficient** than the pre-test as there is no need for an additional unconditional jump.

Note: `VarX` is variable. you will need to load it to a register.



Implementation of DO- WHILE in ARM assembly language

FOR Implementation

- Implementation of **FOR** loop constructs:

- The FOR loop is a **pre-test** loop that evaluates the condition first before executing loop segment.

- If loop segment is executed and count **N** is not used in loop segment, some optimizing compilers implement the **FOR** loop using a **post-test** with **decrement** & **test for zero**.

```
FOR (N=0; N<5; N++)
{
    Loop segment (x5)
}
```

```
        MOV R0,#0
Back    CMP R0,#5
        BGE Exit
         :
         :   } Loop segment
        ADD R0,R0,#1
        B   Back
Exit     :
```

Implementation of FOR in ARM assembly language

Summary

- **SWITCH** constructs can be implemented efficiently depending nature of the case values.
  - Narrow consecutive values can benefit from a jump table.
  - Forked if-else-if can be used with wide ranged values.
- **Post-test** loops are **more efficient** that pre-test loops for the same loop segment.
- With optimised compilers, the low-level code produced may not tally directly with the high-level operations. (e.g. loop increments may be implemented as decrements for better code efficient).

Chapter 8: Instruction Encoding and ISA

Mohamed M. Sabry Aly

N4-02c-92

Learning Objectives

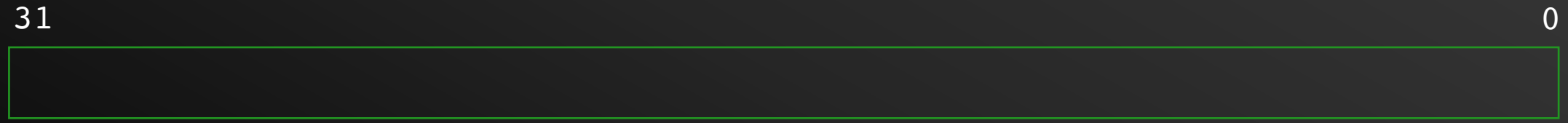
- Describe the instruction format of ARM instruction-set architecture (ISA)

- Describe differences between fixed and variable-length ISA

Instruction Encoding

- For CPU to identify each unique assembly instruction, they must be encoded with unique binary patterns.
  - ARM 32-bit machine encode instructions in a 32-bit word

- What is included?
  - Instruction type ➔ encoded
  - Operand(s) ➔ encoded
  - Condition for conditional execution ➔ encoded
  - Other flags?

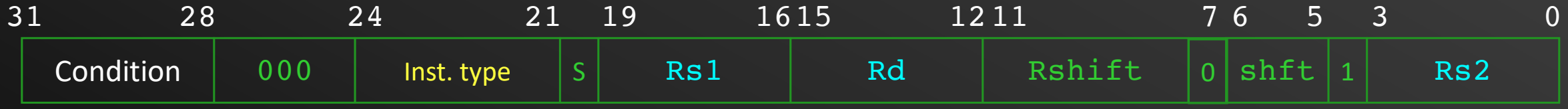


Overall Instruction Format

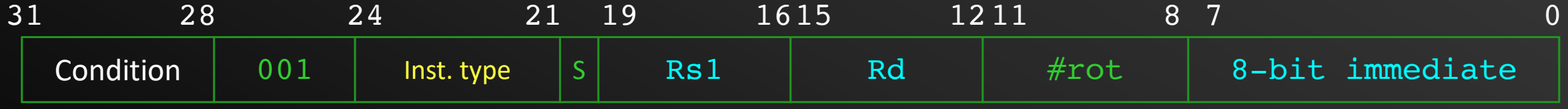
- Arithmetic & Logic instructions
  - ADD, ADC, SUB, SBC, RSB, AND, EOR, RSC, ORR, BIC

XXXCCS Rd, Rs1, Rs2 , {shft #}

XXXCCS Rd, Rs1, Rs2 , {shft Rshift}



XXXCCS Rd, Rs1, #immediate (rotated)

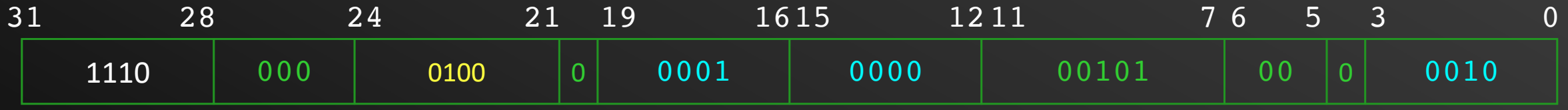


Overall Instruction Format

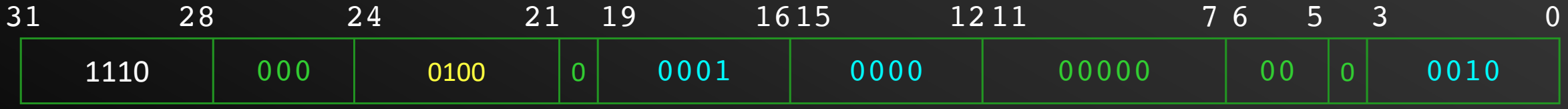
- Arithmetic & Logic instructions
  - ADD, ADC, SUB, SBC, RSB, AND, EOR, RSC, ORR, BIC

XXXCCS Rd, Rs1, Rs2 , {shft #}

ADD R0,R1,R2,LSL #5



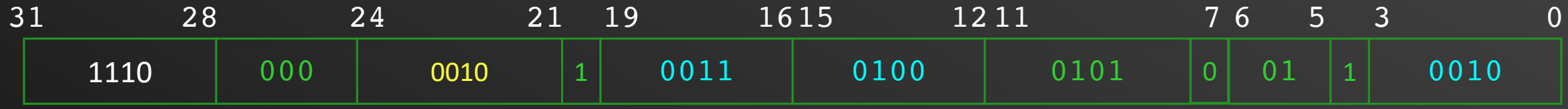
ADD R0,R1,R2



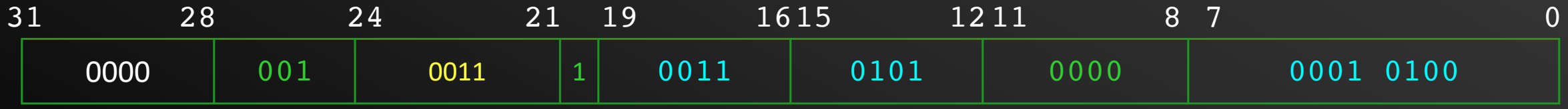
Overall Instruction Format

- Arithmetic & Logic instructions
  - `ADD, ADC, SUB, SBC, RSB, AND, EOR, RSC, ORR, BIC`

`SUBS R4, R3, R2, LSR R5`



`RSBEQS R5,R3,#20`

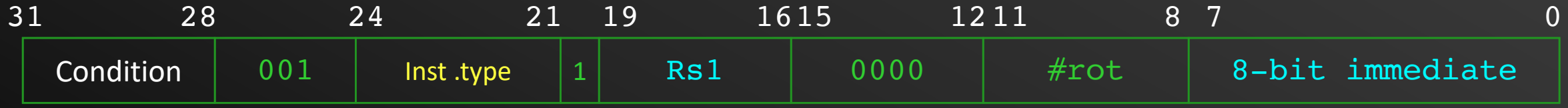


Overall instruction format

- Comparison instructions
  - `TST, TEQ, CMP, CMN`

`XXXCC Rs1, Rs2 , {shft #}`

`XXXCC Rs1, #immediate (rotated)`

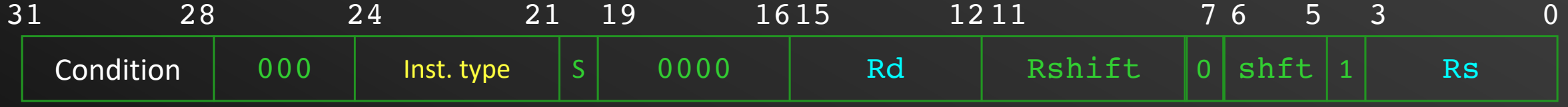


Overall instruction format

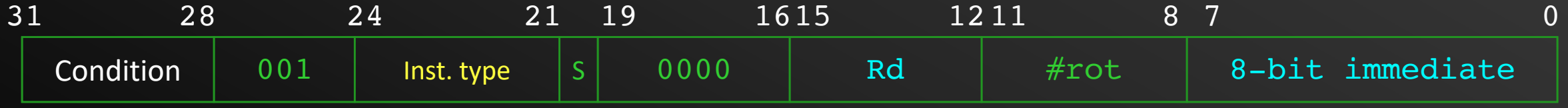
- MOV instructions
  - MOV, MOVN

XXXCCS Rd, Rs , {shft #}

XXXCCS Rd, Rs , {shft #}



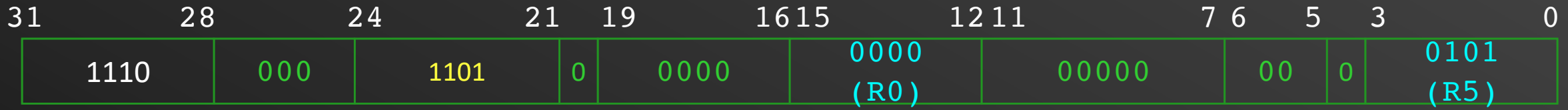
XXXCC Rd, #immediate (rotated)



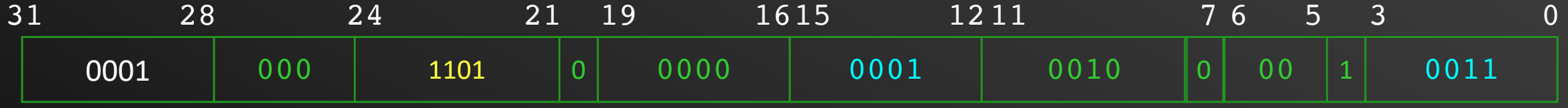
Overall instruction format

- MOV instructions
  - MOV, MOVN

MOV R0,R5



MOVNE R1, R3, LSL R2



MOV R1,#0x40000000

Inst. Type field Field

- 4 bits for instructions ➔ up to 16 combinations
- 4 bits for source and destination registers ➔ 16 registers

Code	Instruction
0000	AND
0001	EOR
0010	SUB
0011	RSB
0100	ADD
0101	ADC
0110	SBC
0111	RSC

Code	Instruction
1000	TST
1001	TEQ
1010	CMP
1011	CMN
1100	ORR
1101	MOV
1110	BIC
1111	MVN

Condition Field

- 4 bits for condition ➔ up to 16 difference combinations

Code	Condition	Flags	Meaning
0000	EQ	Z = 1	Equal
0001	NE	Z = 0	Not equal
0010	CS or HS	C = 1	Higher or same, unsigned
0011	CC or LO	C = 0	Lower, unsigned
0100	MI	N = 1	Negative
0101	PL	N = 0	Positive or zero
0110	VS	V = 1	Overflow
0111	VC	V = 0	No overflow
1000	HI	C = 1 and Z = 0	Higher, unsigned
1001	LS	C = 0 or Z = 1	Lower or same, unsigned
1010	GE	N = V	Greater than or equal, signed
1011	LT	N != V	Less than, signed
1100	GT	Z = 0 and N = V	Greater than, signed
1101	LE	Z = 1 and N != V	Less than or equal, signed
1110	AL		Always.
1111	NV		Reserved (unused)

What about shift instructions, what's there instruction format?

LSL, LSR, ASR, ROR, and RRX has no dedicated instructions

Shift/Rotate instructions

- Instructions are encoded as MOV instructions

  E.g. LSL R1, R2, #5 ≡ MOV R1, R2, LSL #5

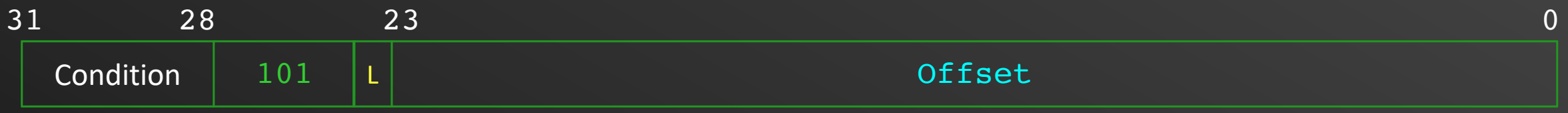
- Specify the shift type in bits 5 and 6

Code	Shift type
00	LSL
01	LSR
10	ASR
11	ROR/RRX

Branch Instructions

- For conditional branch and branch with link

```
B(L)CC   Offset
```



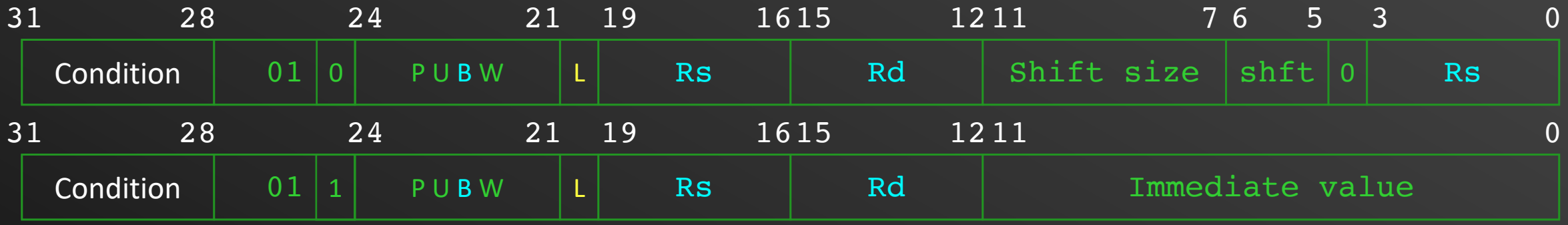
- 26 bits offset shifted to the right by 2 (branching to word addresses) ➔ calculated by the assembler

- Branch range: ±32 MBytes

Load/Store Instructions

- Load/Store word bye

LDR/STR{B}CC Rs,[RD,offset pre or post]W

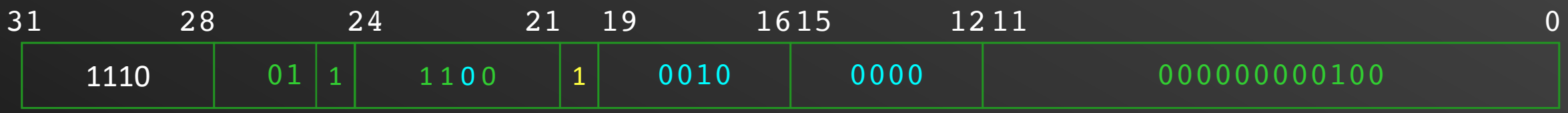


- L bit determines load (1) or store (0)
- P bit determines indexing pre (1) or post (0)
- U determines offset direction add (1) or subtract (0)offset
- B determines byte (1) or word (0) access
- W is for write back (!) of the offset

Load/Store Instructions

- Load/Store word bye

```
LDR   R0,[R2,#4]
```

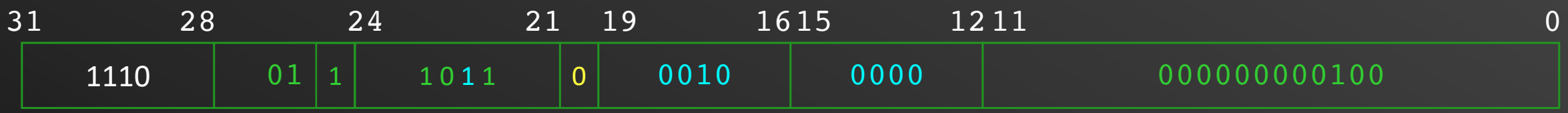


- L bit determines load (1) or store (0)
- P bit determines indexing pre (1) or post (0)
- U determines offset direction add (1) or subtract (0)offset
- B determines byte (1) or word (0) access
- W is for write back (!) of the offset

Load/Store Instructions

- Load/Store word bye

```
STRB R0,[R2,#-4]!
```

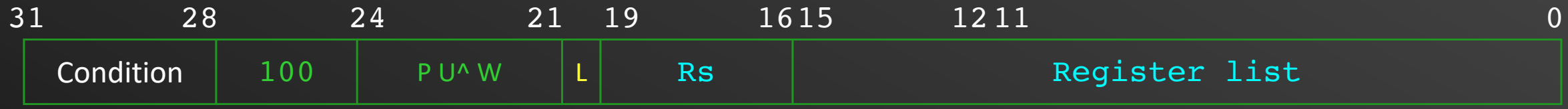


- L bit determines load (1) or store (0)
- P bit determines indexing pre (1) or post (0)
- U determines offset direction add (1) or subtract (0)offset
- B determines byte (1) or word (0) access
- W is for write back (!) of the offset

Load/Store Instructions

- Load/Store Multiple instructions

```
LDM/STMFXCC   RsW,{register list}
```

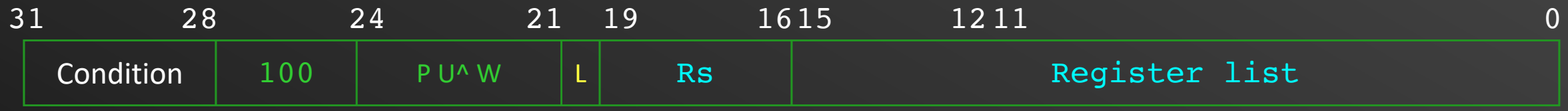


- L bit determines load (1) or store (0)
- P bit determines indexing before (1) or after (0)
- U determines offset direction add (1) or subtract (0)
- ^ is “don’t care”
- W is for write back (!) after operation

Load/Store Instructions

- Load/Store Multiple instructions

```
LDM/STMFXCC   RsW,{register list}
```



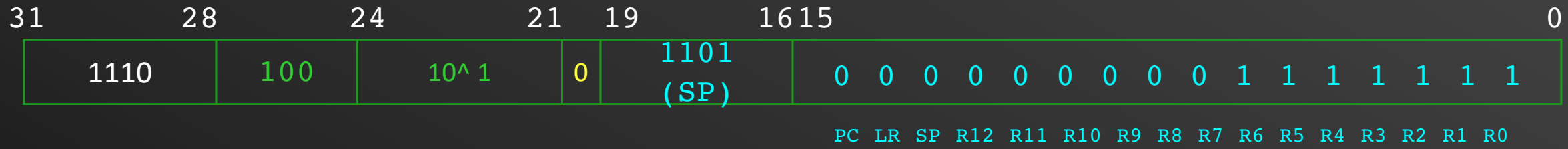
- L bit determines load (1) or store (0)
- P bit determines indexing before (1) or after ((
- U determines offset direction add (1) or subtr
- ^ is “don’t care”
- W is for write back (!) after operation

PU	Instruction
10	STMFD
01	LDMFD
00	STMFA
11	LDMFA

Load/Store Instructions

- Load/Store Multiple instructions

```
STMFD SP!,{R0-R6}
```



- L bit determines load (1) or store (0)
- P bit determines indexing before (1) or after (0)
- U determines offset direction add (1) or subtract (0)
- ^ is “don’t care”
- W is for write back (!) after operation

Fixed vs. Variable-Length instruction

- ARM is fixed-length
  - Operands had to be registers or immediate values

- Variable length instructions
  - Intel 80x86: Instructions vary from 1 to 17 bytes long.
  - Digital VAX: Instructions vary from 1 to 54 bytes long.
  - Require multi-step fetch and decode.
  - Allow for a more flexible (but complex) and compact instruction set.

Variable-Length: Intel Instruction Format

- Complex instructions = complex hardware
- Higher variety for smaller code and faster execution

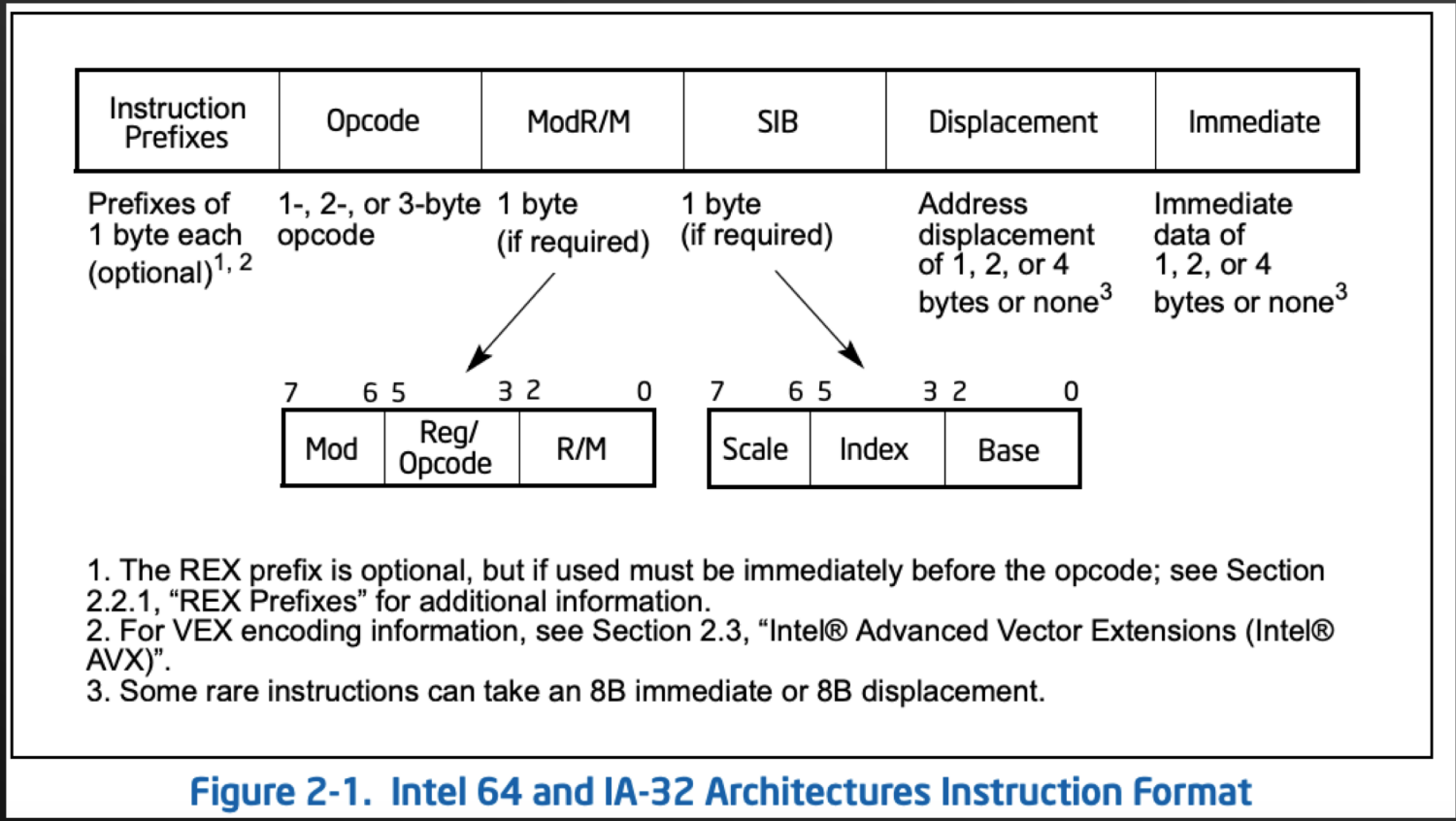
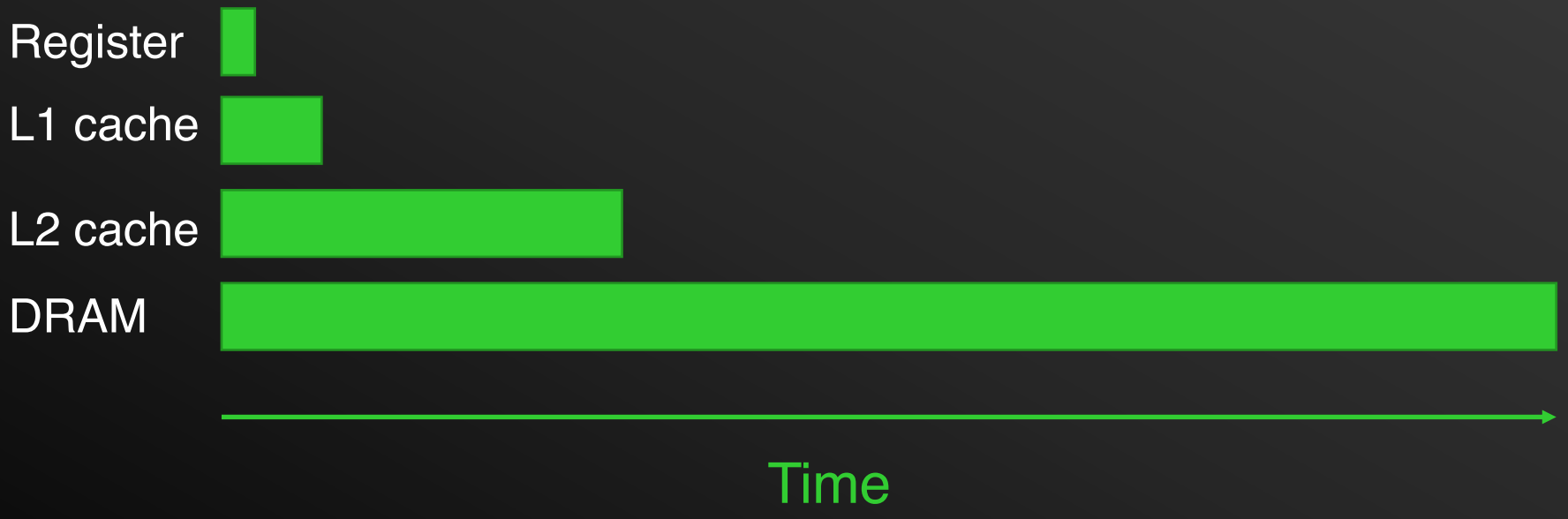


Figure 2-1. Intel 64 and IA-32 Architectures Instruction Format

Using Registers

- Operations using registers are **faster** than those involving memory.
  - Data transfer between registers and ALU is faster due to it physically proximity.
  - To speed up code execution, keep as much of your computations within **registers**.



Implication of Register Count

- Implementing many registers within the CPU will incurs increasing overheads.
  - They take up precious **space** in the silicon die.
  - Connections to various busses increase the **routing** and **multiplexing complexity**.
  - More registers increases the **operand size** during instruction encoding.

ADD (3)

15	14	13	12	11	10	9	8 – 6	5 – 3	2 – 0
0	0	0	1	1	0	0	Rm	Rn	Rd

This form of ADD adds the value of one register to the value of a second register, and stores the result in a third register. The condition code flags are updated, based on the result.

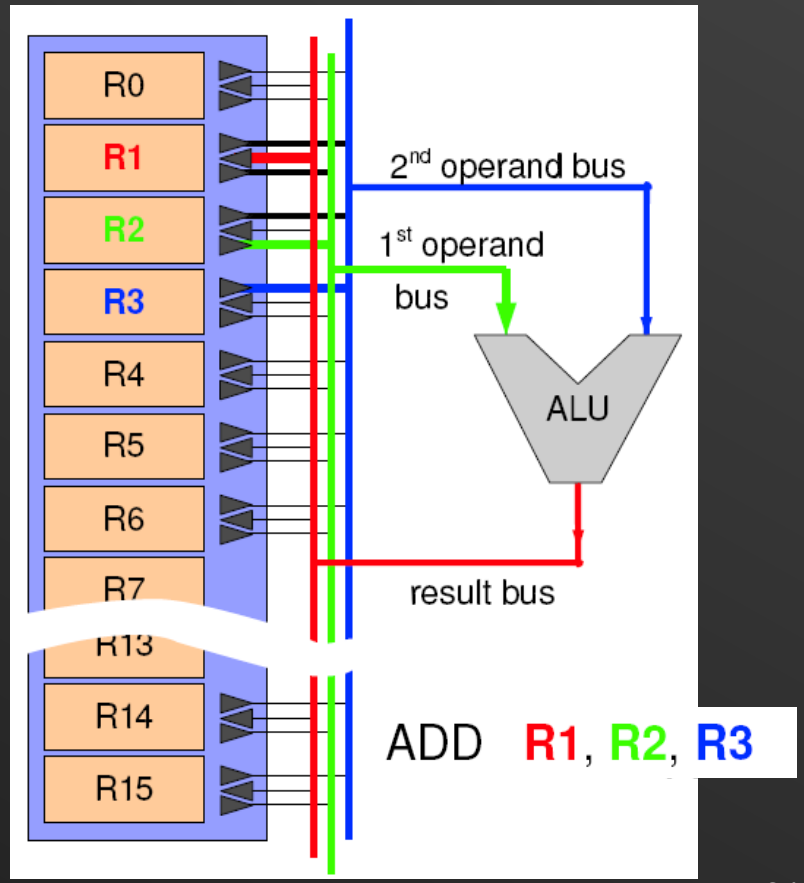
Syntax

`ADD <Rd>, <Rn>, <Rm>`

where:

`<Rd>`	Is the destination register for the completed operation.
`<Rn>`	Specifies the register containing the first value for the addition.
`<Rm>`	Specifies the register containing the second value for the addition.

Instruction Encoding for ADD in Thumb-2



Orthogonality of ISA

- In a truly orthogonal ISA, every instruction is able to use every available addressing modes.
- A truly orthogonal ISA does not restrict certain instruction from using only specific registers.
- Difficult to achieve complete orthogonality due to the **large number of bit patterns** required to express all combinations of operations and addressing modes.
- Increase instruction length will increases the memory size required by the program.

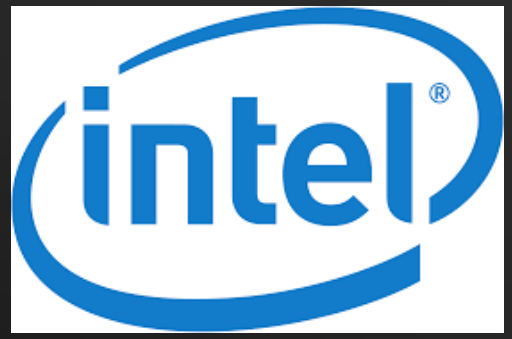
Summary - Why is Good Instruction Set Architecture (ISA) Design Important?

- Good ISA design can yield benefits such as:

  - Increases the execution performance of the processor.
  - Increases the code density (i.e. how much memory a piece of code will occupy).
  - Reduces the power consumption of the CPU.
  - Reduces manufacturing cost of processor.
  - Easy for programmers to write efficient programs.
  - Efficient mapping of high-level programming requirements into low-level instruction sets.

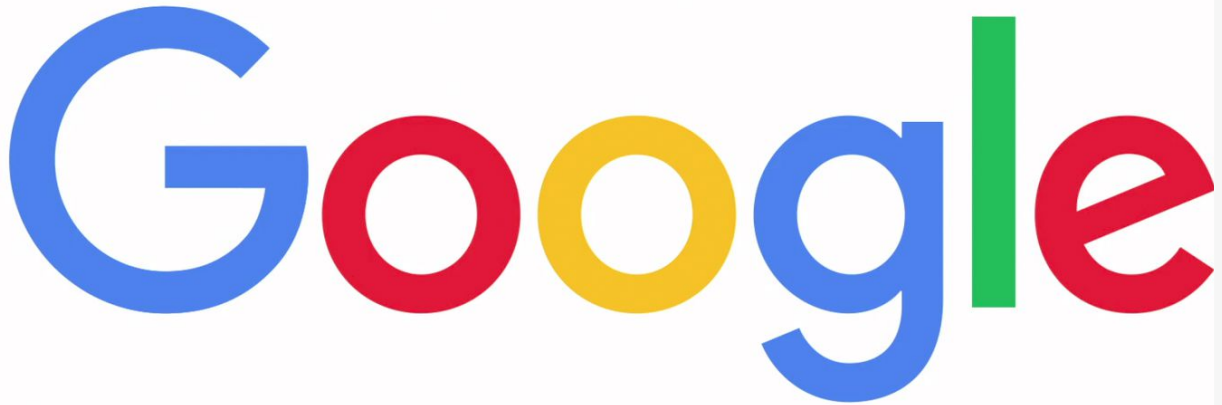
Why Should I Care About Processor Designs and ISA?

- Designing a processor (or a co-processor) is no longer a hardware company job



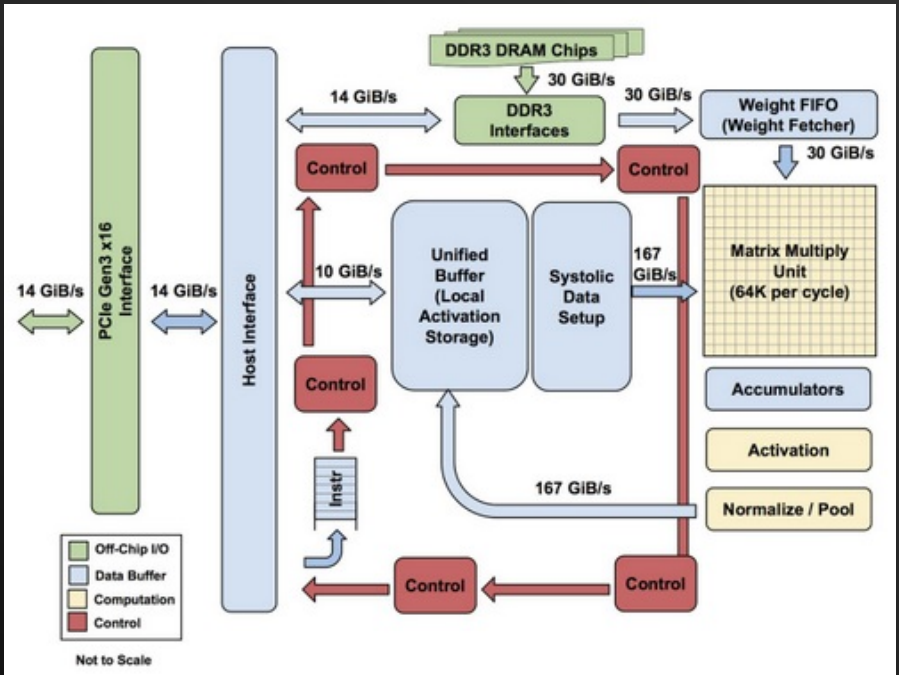
Why Should I Care About Processor Designs and ISA?

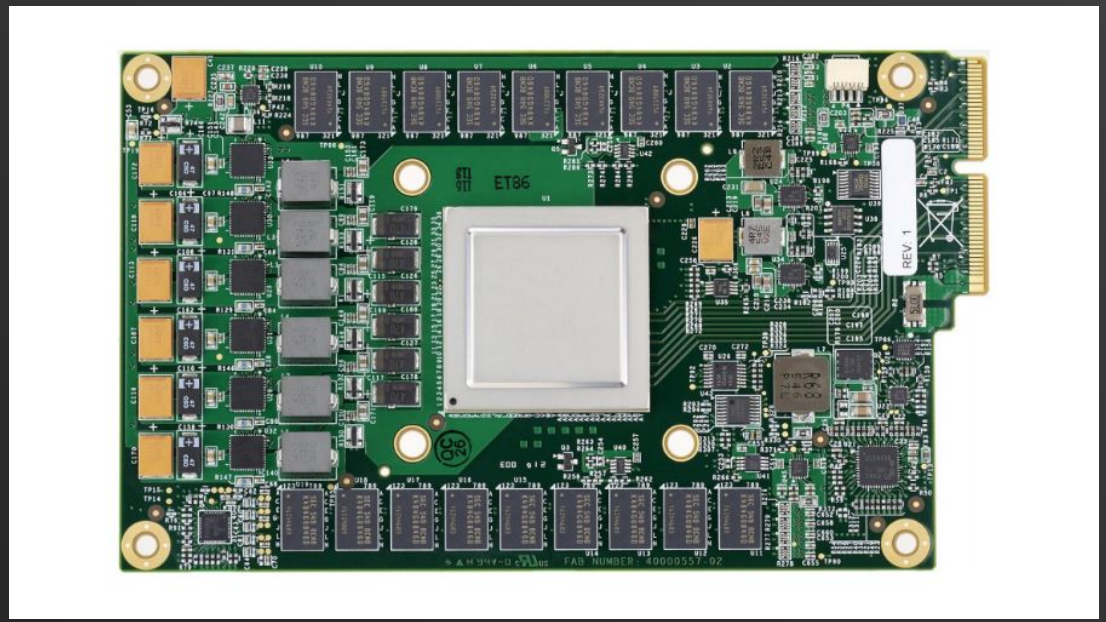
- Designing a processor (or a co-processor) is no longer a hardware company job



The Google TPU

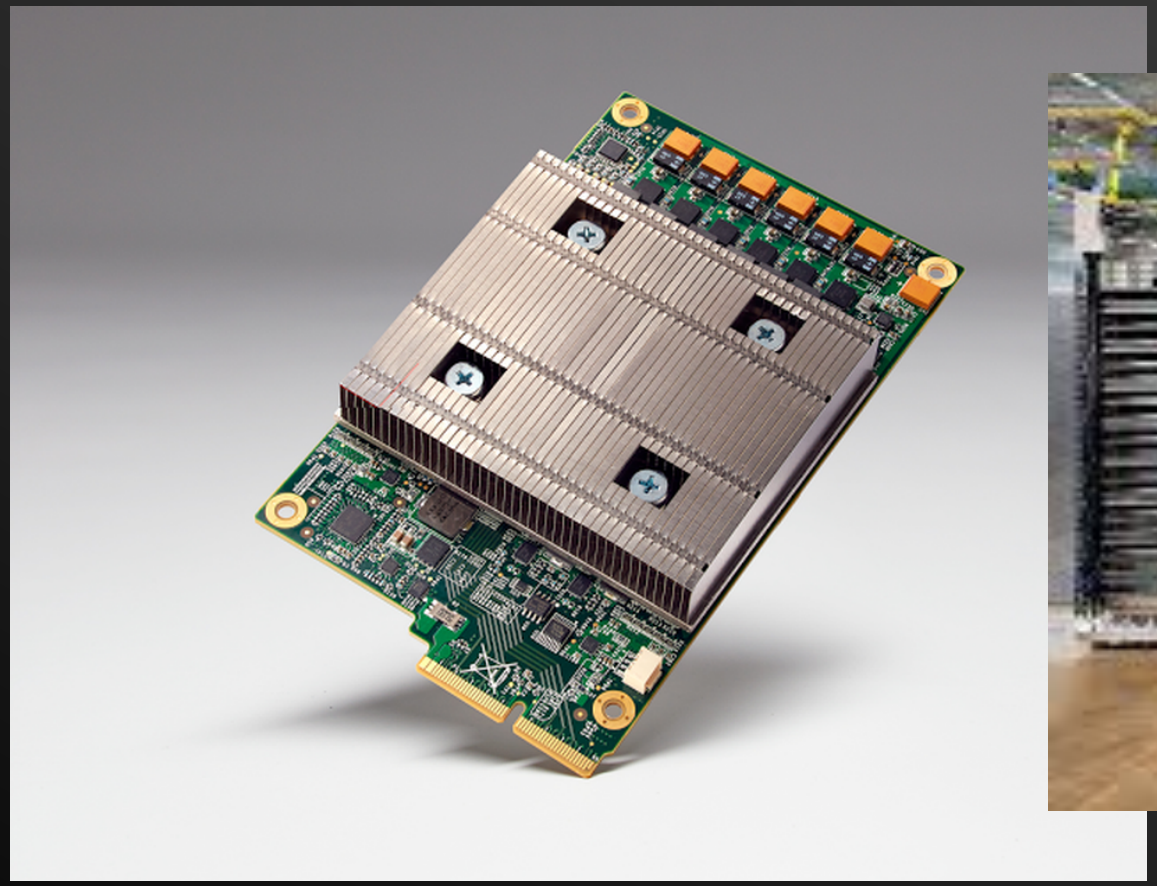
- Tensor Processing Unit
- Designed by Engineers in Google
- Tailored to make AI applications run faster





The Google TPU

- Design by Engineers in Google to make AI applications run faster



Summary

- ARM is fixed-length load/store architecture
  - Operands reside in registers
- Other architectures can use variable-length format
  - E.g, Intel architecture
- Use of registers is key towards higher performance