

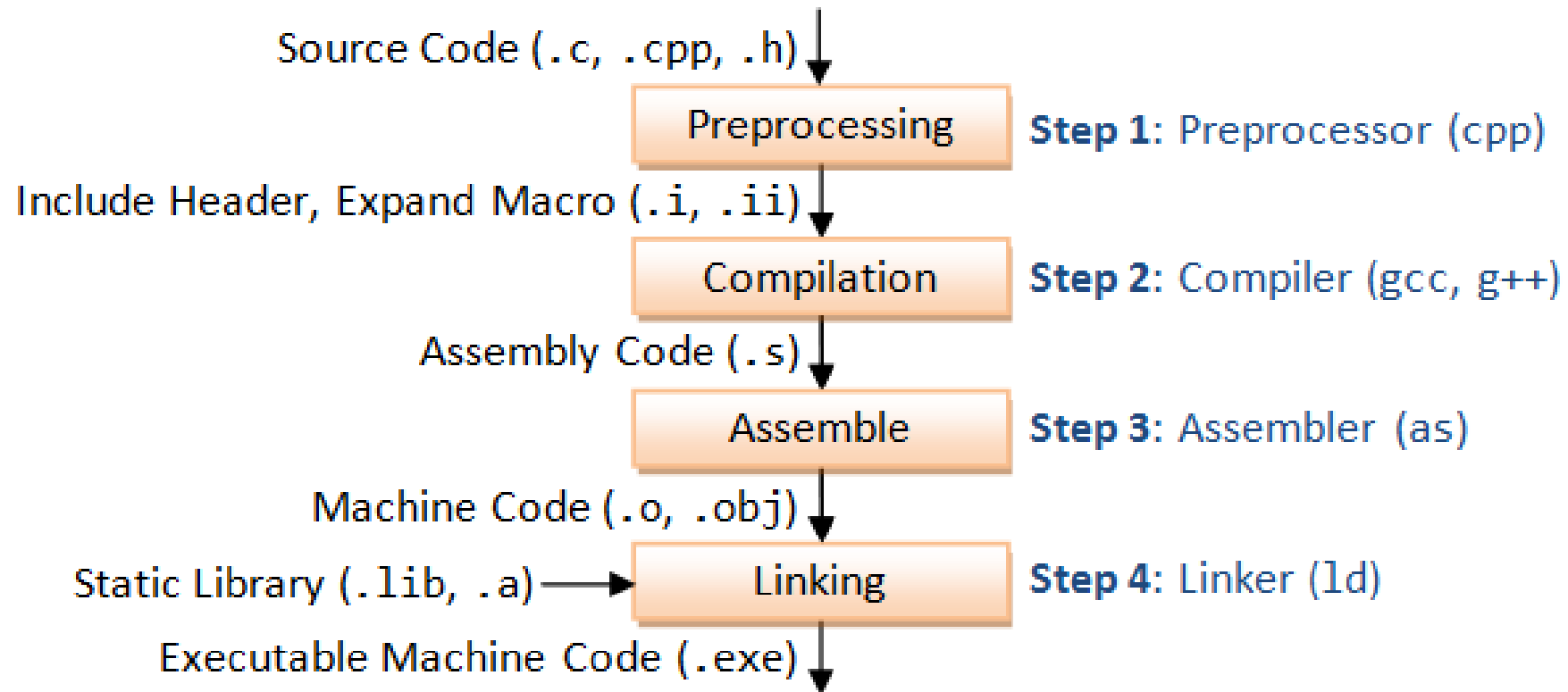
# CSE 211: Introduction to Embedded Systems

## Section 4

# Contact Information

[tasneem.awaad@eng.asu.edu.eg](mailto:tasneem.awaad@eng.asu.edu.eg)

# Software Build Process



# General Layout of Assembly

- {label} {instruction | directive | pseudo-instruction} {;comment}

# Addressing Modes (Immediate Addressing )

- MOV
  - Format: **MOV Rn, Op2**
  - **Op2** can be register or #immediate
  - MOV R0,#0x25
- LDR Rn, =0x123

# Addressing Modes

- Load/Store memory
  - Register indirect Addressing Mode
  - PC relative Addressing Mode
  - PUSH and POP Register Addressing Mode

# Addressing Modes

- Load/Store memory
  - Register indirect Addressing Mode
    - **Regular Register indirect**
      - LDR R7, [R5]
      - R5 unchanged, R7 = Mem[R5]
    - **With Immediate Offset:**
      - LDR R7, [R5, #4]
      - R5 unchanged, R7 = Mem[R5 + 4]
    - **With Register Offset:**
      - LDR R7, [R5, R4]
      - R5 Unchanged, R7 = Mem[R5 + R4]

# Addressing Modes

- Load/Store memory
  - Register indirect Addressing Mode
    - **With Pre indexed Immediate Offset:**
      - LDR R7, [R5, #4]!
      - $R5 = R5 + 4$
      - $R7 = \text{Mem}[R5]$
    - **With Post indexed Immediate Offset:**
      - LDR R7, [R5], #4
      - $R7 = \text{Mem}[R5]$
      - $R5 = R5 + 4$
    - **With Shifted Register Offset:**
      - LDR R7, [R5, R4, LSL #2]
      - R5 Unchanged,  $R7 = \text{Mem}[R5 + R4 \ll 2]$



# PC Relative Addressing

- PC-relative: An addressing mode where the effective address is calculated by its position relative to the current value of the program counter.
- PC-relative addressing is indexed addressing mode using the PC as the pointer.
  - LDR Rn, =1234567 to move any 32-bit value into a register
  - BL{cond} label ; branch to subroutine at label
  - It is used for branching, for calling functions
  - B Location ; jump to Location, using PC-relative addressing
  - LDR R1,[PC,#28]

# Steps needed for function call & return

- Transfer control to the function being called (callee) (This is some location in memory different from the current address in pc)
- Pass parameters to the function
- Transfer control back to the caller once function execution is complete
- Make return values available to caller function

```

... sum(a,b);... /* a,b:r4,r5 */
}
C int sum(int x, int y) {
    return x+y;
}

```

Is there something wrong with using the simple branch instruction ?

---

	address	A. Nothing is wrong
	1000 ...	B. It is not sufficient for generic usage scenarios
	1004 ...	Reason: <b>sum</b> might be called by many functions,
A	1008 ...	so we can't return to a fixed place.
R	1012 B sum	The calling proc to <b>sum</b> must be able to say "return
M	1016 return_loc:...	back here" somehow.
	1020 ...	
	2000 sum: ADD r0,r0,r1	
	2004 B return_loc	

**C**

```
... sum(a,b);... /* a,b:r4,r5 */  
}  
int sum(int x, int y) {  
    return x+y;  
}
```

---

**A  
R  
M**

address

1000 ...

1004 ...

1008 MOV lr,1016 ; lr = 1016

1012 B sum ; branch to sum

1016 ...

1020 ...

2000 sum: ADD r0,r0,r1

2004 BX lr ; MOV pc,lr i.e., return


---

```
... sum(a,b);... /* a,b:$s0,$s1 */  
}  
C int sum(int x, int y) {  
    return x+y;  
}
```

---

❖ Question: Why use **BX** here? Why not simply use **B**?

**A** ❖ Answer: **sum** might be called by many functions, so  
**R** we can't return to a fixed place. The calling proc to  
**M** **sum** must be able to say "return here" somehow.

 2000 sum: ADD r0,r0,r1  
2004 **BX lr** ; new instruction

# Instruction Support for Functions

- Single instruction to jump and save return address: jump and link (BL)
- Before:
  - 1008 MOV lr, 1016 ; lr=1016
  - 1012 B sum ; goto sum
- After:
  - 1008 BL sum # lr=1012, goto sum
  - Why have a BL? Make the common case fast:
  - function calls are very common. Also, you don't have to know where the code is loaded into memory with BL.

# Instruction Support for Functions

- Syntax for BL (branch and link) is same as for B (branch):
  - BL label
- BL functionality:
  - Step 1 (link): Save address of next instruction into lr (Why next instruction? Why not current one?)
  - Step 2 (branch): Branch to the given label

# Instruction Support for Functions

- Syntax for BX (branch and exchange):
  - BX register
- Instead of providing a label to jump to, the BX instruction provides a register which contains an address to jump to
- Only useful if we know exact address to jump
- Very useful for function calls:
  - BL stores return address in register (lr)
  - BX lr jumps back to that address



# Functions

- How do we pass arguments?
  - Use registers?
- How do we share registers between caller and callee
  - Need to follow a standard on the usage of registers by caller and callee  
We have a std: The ARM Application Procedure Call Std. (AAPCS)
  - Need to interact with memory (done via stack)

# Steps for Making a Procedure Call

- Save necessary values onto stack
- Assign argument(s), if any
- BL call
- Restore values from stack
- Must follow register conventions

# Rules for Procedures

1. Called with a BL instruction, returns with a BX lr (or MOV pc, lr)
2. Accepts up to 4 arguments in r0, r1, r2 and r3
3. Return value is always in r0 (and if necessary, in r1, r2, r3)

# Registers

Register	Synonym	Role in Procedure Call Standard
r0-r1	a1-a2	Argument/Result/Scratch Register
r2-r3	a3-a4	Argument/Scratch Register
r4-r8	v1-v5	Variable Register
r9	v6	Variable Register
r10-r11	v7-v8	Variable Register
r12	ip	Intra-Procedure Call Scratch Register
r13	sp	Stack Pointer
r14	lr	Link Register
r15	pc	Program Counter

Register
r0
r1
r2
r3
r4
r5
r6
r7
r8
r9
r10
r11
r12
r13 'sp'
r14 'lr'
r15 'pc'

<p>Scratch Registers: r0-r3, r12  r0-r3 used to pass parameters  r12 intra-procedure scratch  will be overwritten by subroutines</p>
<p>Preserved Registers: r4-r11  stack before using  restore before returning</p>
<p>Stack Pointer:  not much use on the stack</p>
<p>Link Register:  set by BL or BLX on entry of routine  overwritten by further use of BL or BLX</p>
<p>Program Counter</p>

Register Use in the ARM Procedure Call Standard

**Arguments into function**  
**Result(s) from function**  
**otherwise corruptible**  
**(Additional parameters**  
**passed on stack)**

**Register**

<b>r0</b>
<b>r1</b>
<b>r2</b>
<b>r3</b>

**Register variables**  
**Must be preserved**

<b>r4</b>
<b>r5</b>
<b>r6</b>
<b>r7</b>
<b>r8</b>
<b>r9/sb</b>
<b>r10/sl</b>
<b>r11</b>

**Scratch register**  
**(corruptible)**

<b>r12</b>
------------

The compiler has a set of rules known as a Procedure Call Standard that determine how to pass parameters to a function (see **AAPCS**)

CPSR flags may be corrupted by function call.  
Assembler code which links with compiled code must follow the AAPCS at external interfaces

- **Stack base**

- **Stack limit if software stack checking selected**

# Registers

- `lr`: Can Change. Caller needs to save on stack if nested call.
- `R0-R3 (a1-a4)`: Can change. These are volatile argument registers. Caller needs to save if they'll need them after the call. E.g., `r0` will change if there is a return value.
- `R12`: may be used by a linker as a scratch register between a routine and any subroutine it calls. It can also be used within a routine to hold intermediate values between subroutine calls.

# Stack

- The stack is a contiguous area of memory block used for storage of local variables, for passing additional arguments to subroutines when there are insufficient argument registers available, for supporting nested routine calls, and for handling processor interrupts.
- The stack has a Last In First Out (LIFO) behavior. As new data comes in, it pushes down the older one. The most recently entered request always resides at the top of the stack. Any item that is stored in the stack can only be retrieved after all other items that were stored in the stack later were removed.
- When data is added to the stack, it is said to be pushed onto the stack. When data is removed from the stack, it is said to be popped off the stack. These operations are done on the top of the stack.



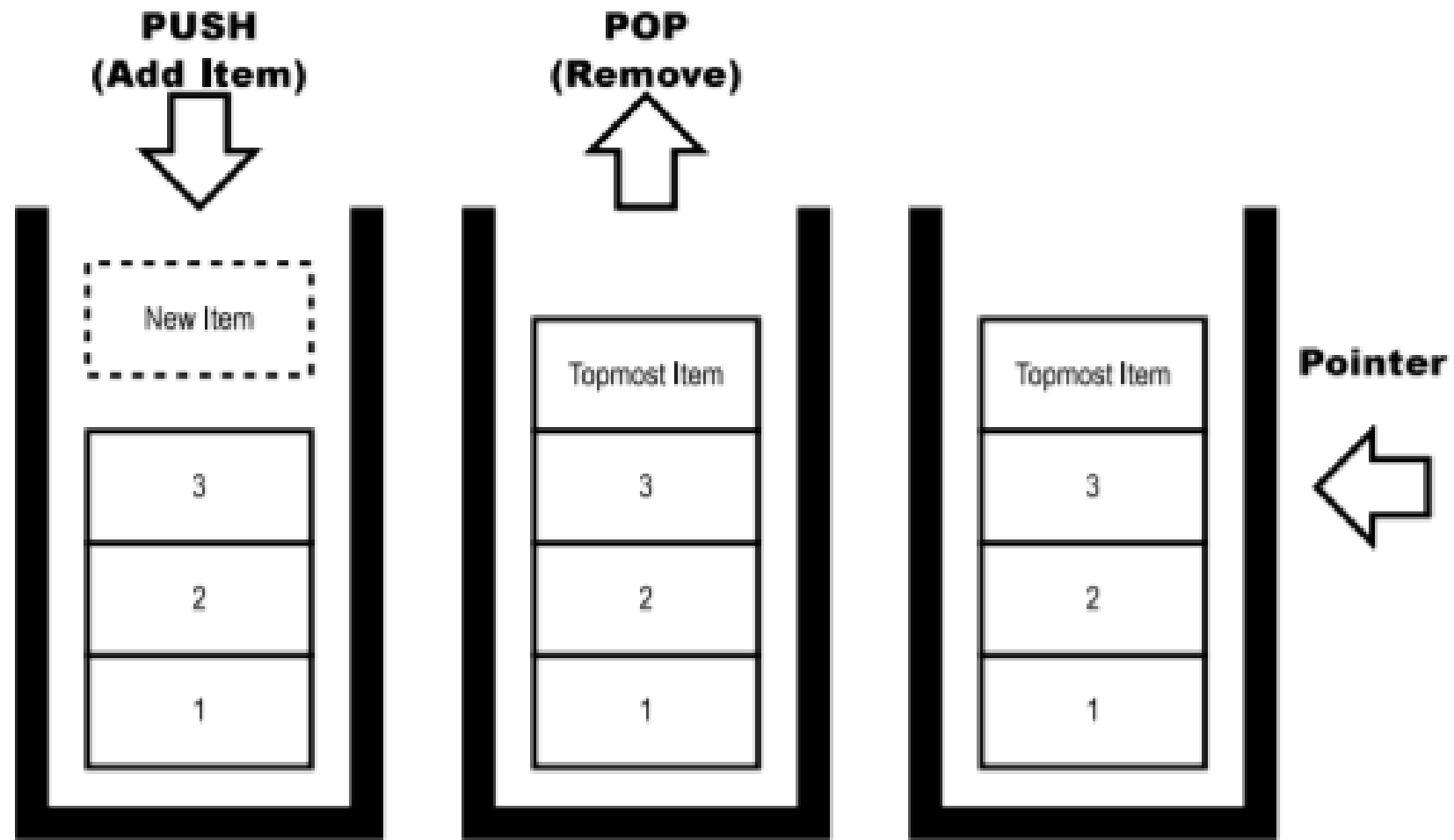
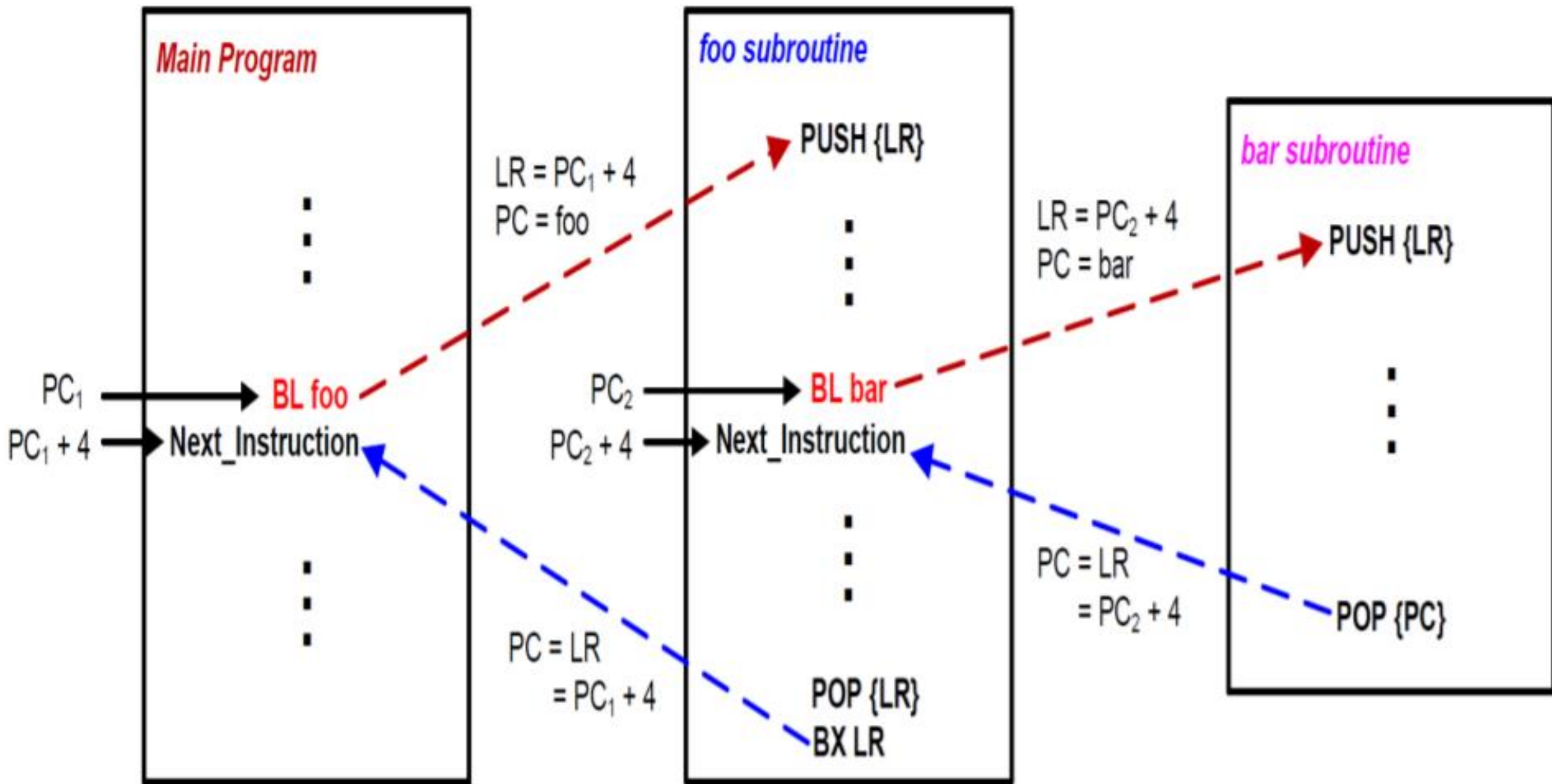


Figure 6.3: Adding, Removing and Accessing Items from Stack

# Subroutine and Stack

- In assembly language programming one subroutine can call another. In this case, the return address of the second call is also stored in the link register destroying the previous contents.
- Hence, it is essential to save the contents of the link register in some other location before calling another subroutine. So, if the link register is pushed onto the stack at entry, additional subroutine calls can safely be made without causing the return address to be lost.
- If you do this, you can also return from a subroutine by popping the PC off the stack at exit, instead of popping LR and then moving that value into the PC.



MAIN PROC

MOV R0, #2

BL QUAD

ENDL ...

ENDP

Function MAIN

QUAD PROC

PUSH {LR}

BL SQ

BL SQ

POP {LR}

BX LR

ENDP

Function QUAD

SQ PROC

MUL R0, R0

BX LR

ENDP

Function SQ

