CET 241: Day 5

Dr. Noori KIM

Agenda

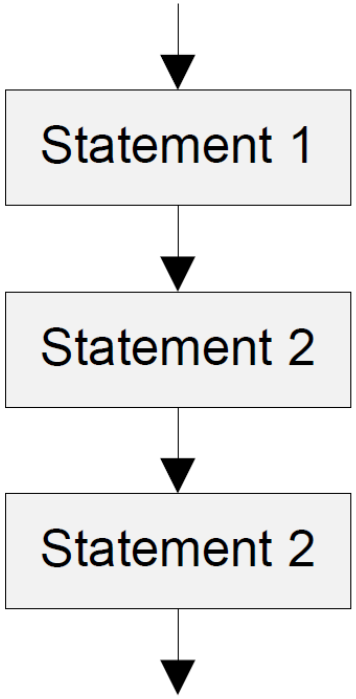
- Top down design
- Reuse of registers
- Passing Parameters to Subroutines via Register
  - “ARM Procedure Call Standard”
  - An example
- Preserve Environment via Stack

"Nothing is particularly hard if you divide it into small jobs."

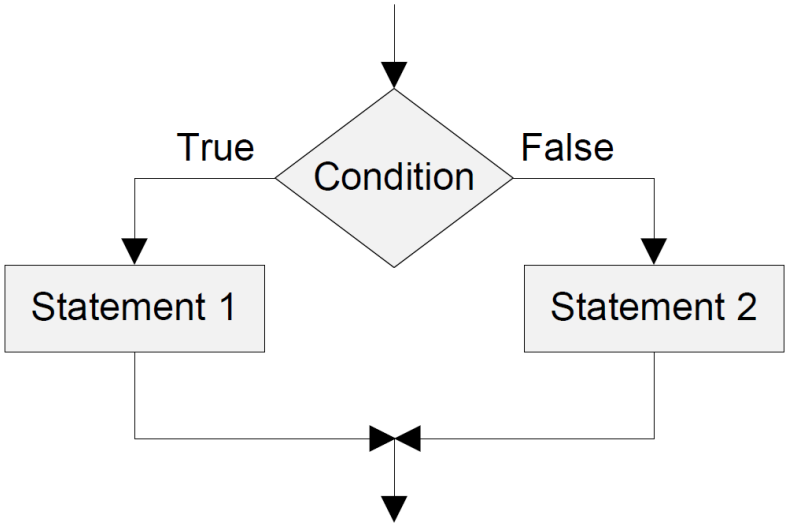
Henry Ford, Founder of Ford Motor

➔The beauty of Divide and Conquer

Three basic control structures

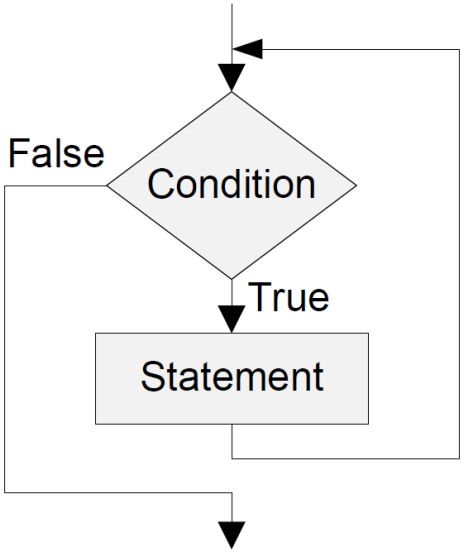


Sequence Structure



Selection Structure

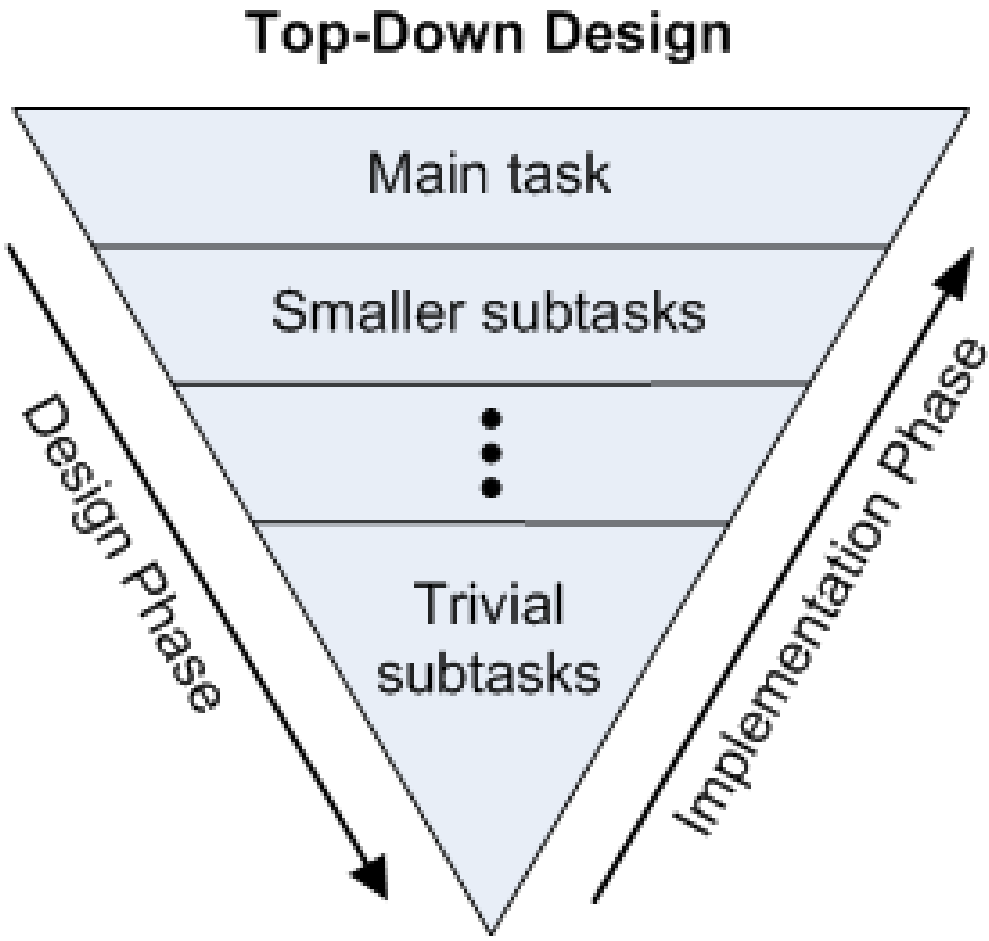
- If, else-if, else
- switch



Loop Structure

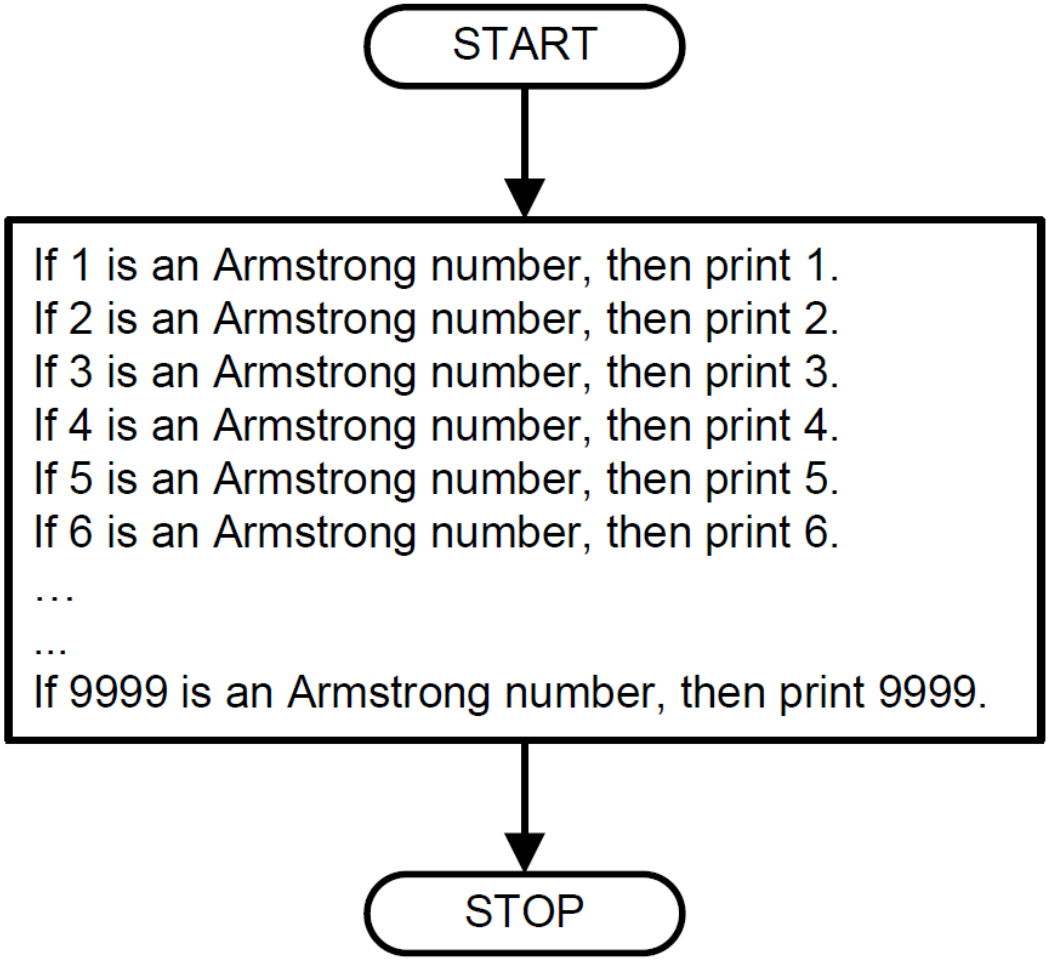
- for
- while

Top-Down Design

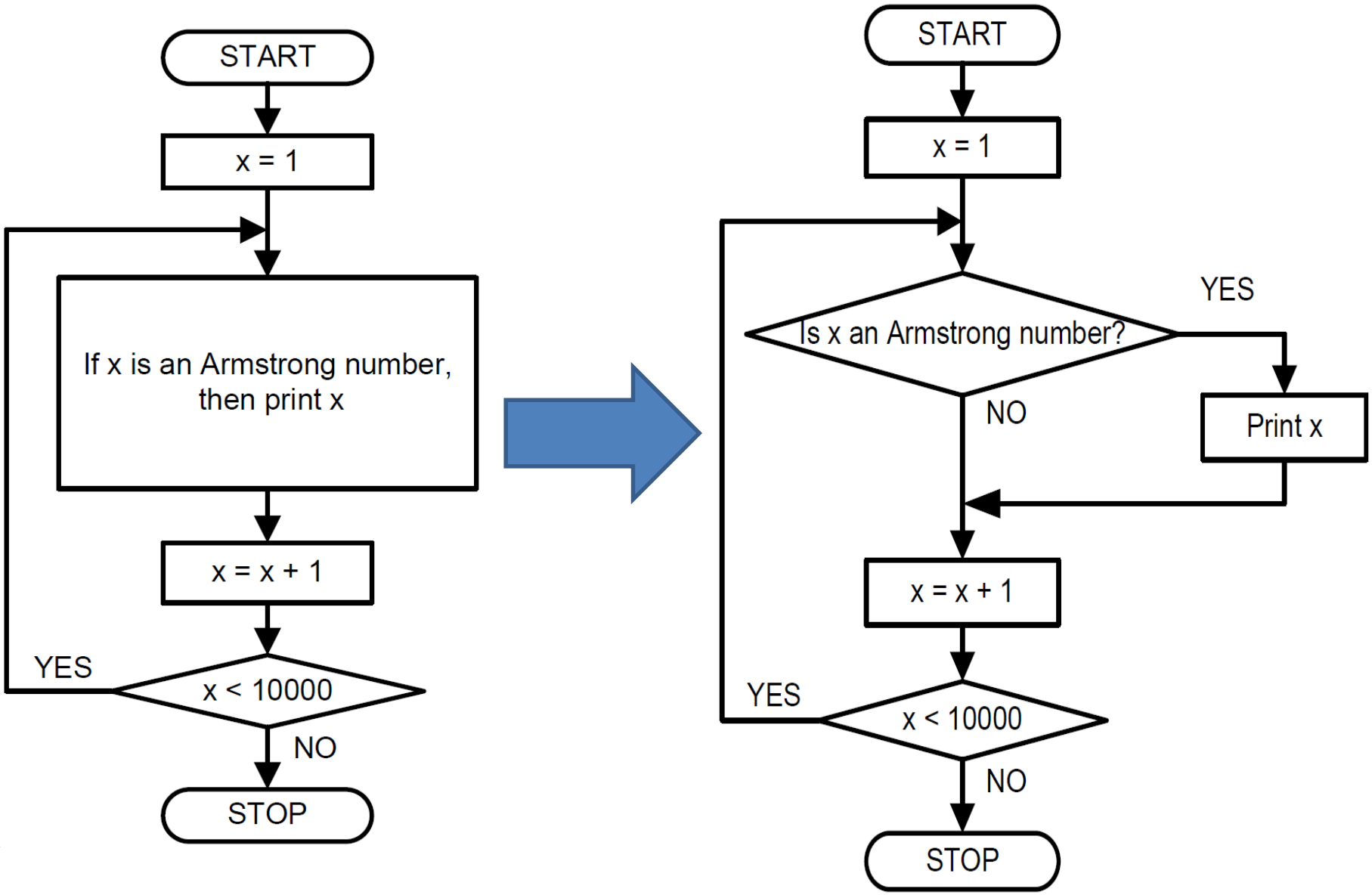


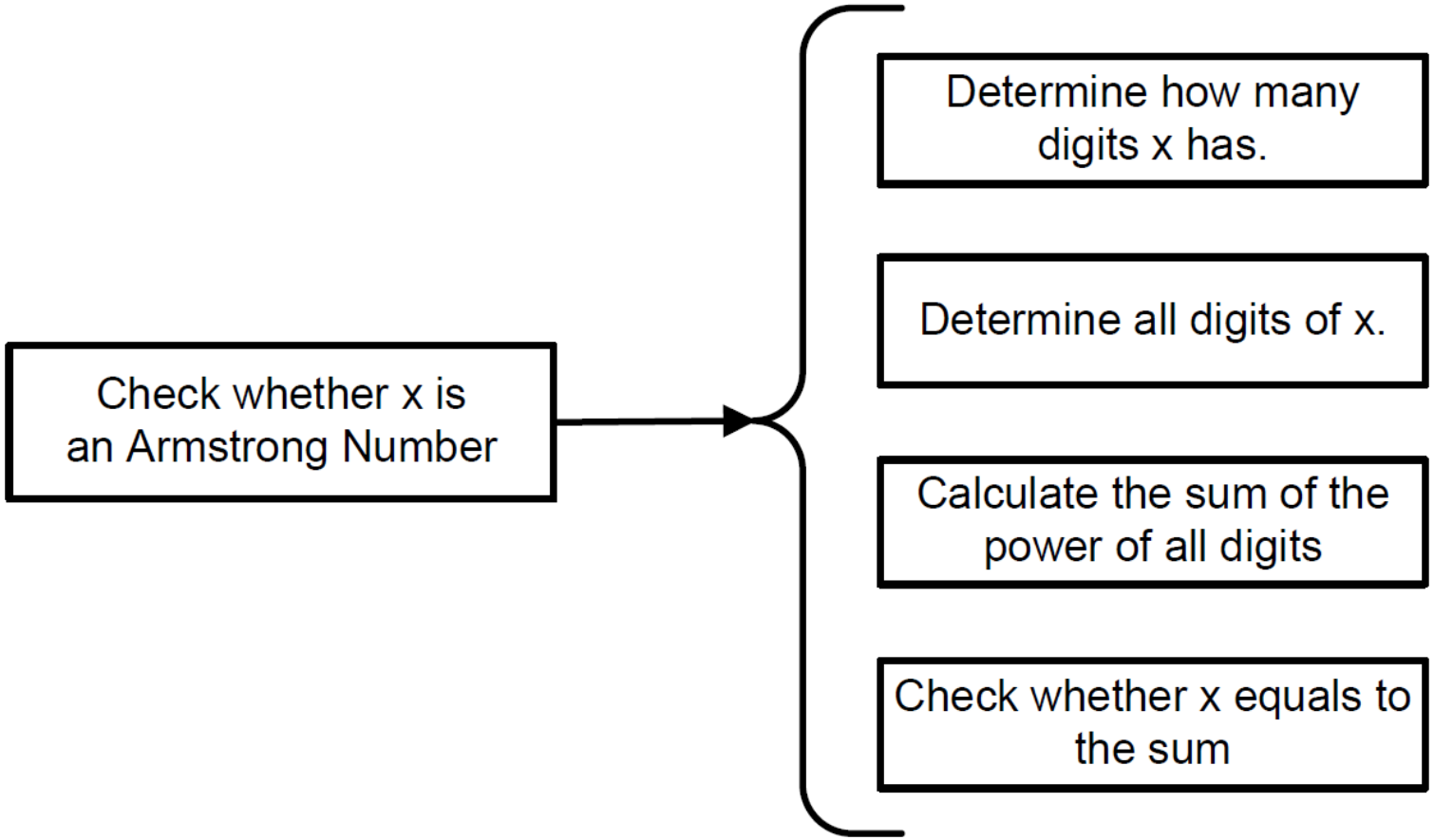
Top-Down Design Example

- Find all Armstrong numbers less than 10,000
- Given a positive integer that has n digits, it is an Armstrong number if the sum of the nth powers of its digits equals the number itself.
- For example, 371 is an Armstrong number since we have $371 = 3^3 + 7^3 + 1^3$.

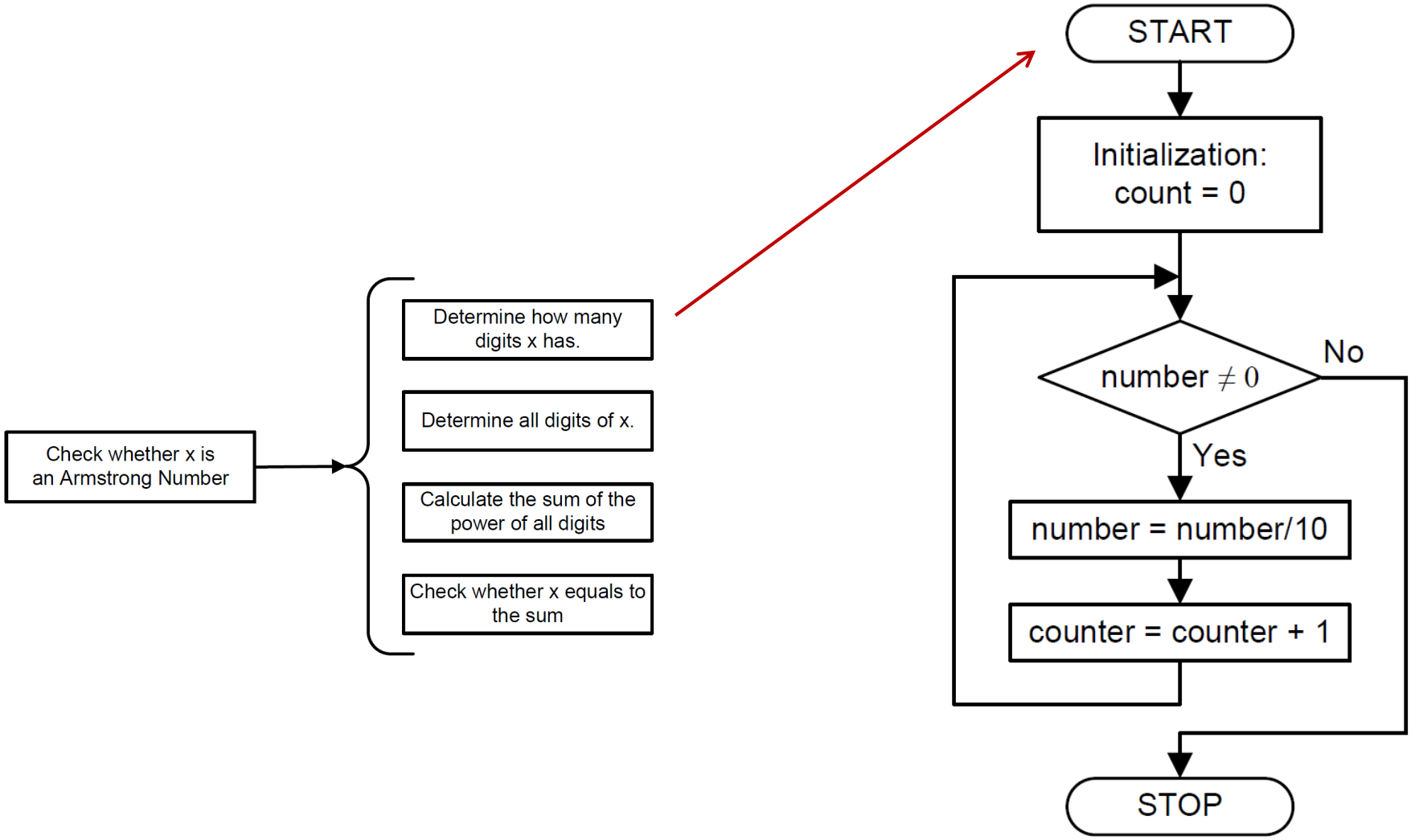

START
If 1 is an Armstrong number, then print 1.
If 2 is an Armstrong number, then print 2.
If 3 is an Armstrong number, then print 3.
If 4 is an Armstrong number, then print 4.
If 5 is an Armstrong number, then print 5.
If 6 is an Armstrong number, then print 6.
…
...
If 9999 is an Armstrong number, then print 9999.
STOP

Restart!


START
x = 1
If x is an Armstrong number, then print x
x = x + 1
YES
x < 10000
NO
STOP
START
x = 1
Is x an Armstrong number?
YES
NO
Print x
x = x + 1
YES
x < 10000
NO
STOP



One of the above will be tested in your quiz


Check whether x is an Armstrong Number
Determine how many digits x has.
Determine all digits of x.
Calculate the sum of the power of all digits
Check whether x equals to the sum
START
Initialization: count = 0
number ≠ 0
No
Yes
number = number/10
counter = counter + 1
STOP

Reuse of Registers

```
int A = 0;  //
int B = -1; //
int C = -2; //
int D = 2;  //

void main(void){
  A = B + C - D;
  return;
}
```

?


```
    AREA myData, DATA
A   DCD  0
B   DCD  -1
C   DCD  -2
D   DCD  2
```

http://www.keil.com/support/man/docs/armasm/armasm_dom1361290005934.htm

```
int A = 0;  // 0x00000000
int B = -1; // 0xFFFFFFFF
int C = -2; // 0xFFFFFFFE
int D = 2;  // 0x00000002

void main(void){
  A = B + C - D;
  return;
}
```

8 Registers are used:

r0,r1,r2,r3,r4,r5,r6,r7

```
AREA myCode, CODE
   EXPORT __main
   ENTRY
__main PROC

   LDR r2, =B      ; r2 = 0x2000,0004
   LDR r3, [r2]    ; r3 = B = -1
   LDR r4, =C      ; r4 = 0x2000,0008
   LDR r5, [r4]    ; r5 = C = -2
   LDR r6, =D      ; r6 = 0x2000,000B
   LDR r7, [r6]    ; r7 = D = 2
   ADD r1, r3, r5 ; r1 = B + C
   SUB r1, r1, r7 ; r1 = B + C - D
   LDR r0, =A      ; r0 = 0x2000,0000
   STR r1, [r0]    ; Save A
   ENDP

   AREA myData, DATA
A  DCD  0
B  DCD  -1
C  DCD  -2
D  DCD  2

   END
```

```
    AREA myCode, CODE
    EXPORT __main
    ENTRY
__main PROC

    LDR r2, =B
    LDR r3, [r2]
    LDR r4, =C
    LDR r5, [r4]
    LDR r6, =D
    LDR r7, [r6]
    ADD r1, r3, r5
    SUB r1, r1, r7
    LDR r0, =A
    STR r1, [r0]
    ENDP

    AREA myData, DATA
A   DCD  0
B   DCD  -1
C   DCD  -2
D   DCD  2

    END
```

Lifetime of r3

Lifetime of r2

Lifetime of r2

8 registers used

```
    AREA myCode, CODE
    EXPORT __main
    ENTRY
__main PROC

    LDR r2, =B
    LDR r3, [r2]
    LDR r2, =C
    LDR r5, [r2]
    LDR r2, =D
    LDR r7, [r2]
    ADD r3, r3, r5
    SUB r3, r3, r7
    LDR r2, =A
    STR r3, [r2]
    ENDP

    AREA myData, DATA
A   DCD  0
B   DCD  -1
C   DCD  -2
D   DCD  2

    END
```

Lifetime of r2

Lifetime of r2

4 registers used

```
    AREA myCode, CODE
    EXPORT __main
    ENTRY
__main PROC

    LDR r2, =B
    LDR r3, [r2]
    LDR r2, =C
    LDR r5, [r2]
    LDR r2, =D
    LDR
    ADD          ?
    SUB
    LDR r2, =A
    STR r3, [r2]
    ENDP

    AREA myData, DATA
A   DCD  0
B   DCD  -1
C   DCD  -2
D   DCD  2

    END
```

3 registers used

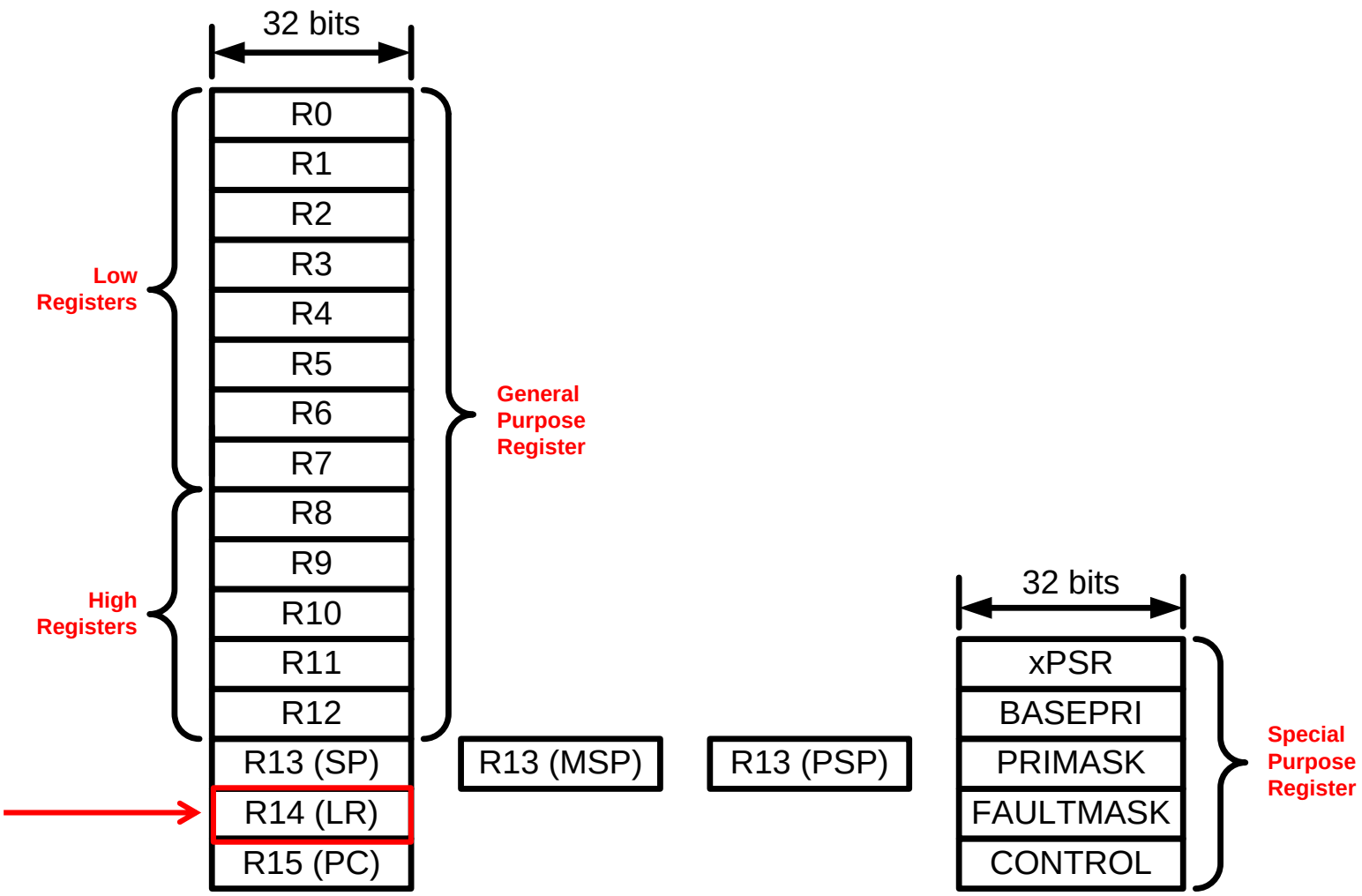
A = B + C – D;

Passing Parameters to Subroutines via Register Preserved Environment via Stack

Subroutine

- A subroutines, also called a function or a procedure,
  - single-entry, single-exit
  - Return to caller after it exits
- When a subroutine is called, the Link Register (LR) holds the memory address of the next instruction to be executed after the subroutine exits.

Link Register



Calling a Subroutine

BL *label*

- Step 1: LR = PC + 4
- Step 2: PC = label

- Notes:
  - *label* is name of subroutine (address)
  - Compiler translates label to memory address
  - After call, LR holds return address (the instruction following the call)

Caller Program

```
MOV r4, #100
...
BL  foo
...
```

Subroutine/Callee

```
foo PROC
    ...
    MOV   r4, #10
    ...
    BX    LR
    ENDP
```

Exiting a Subroutine

BX *LR*

- PC = LR

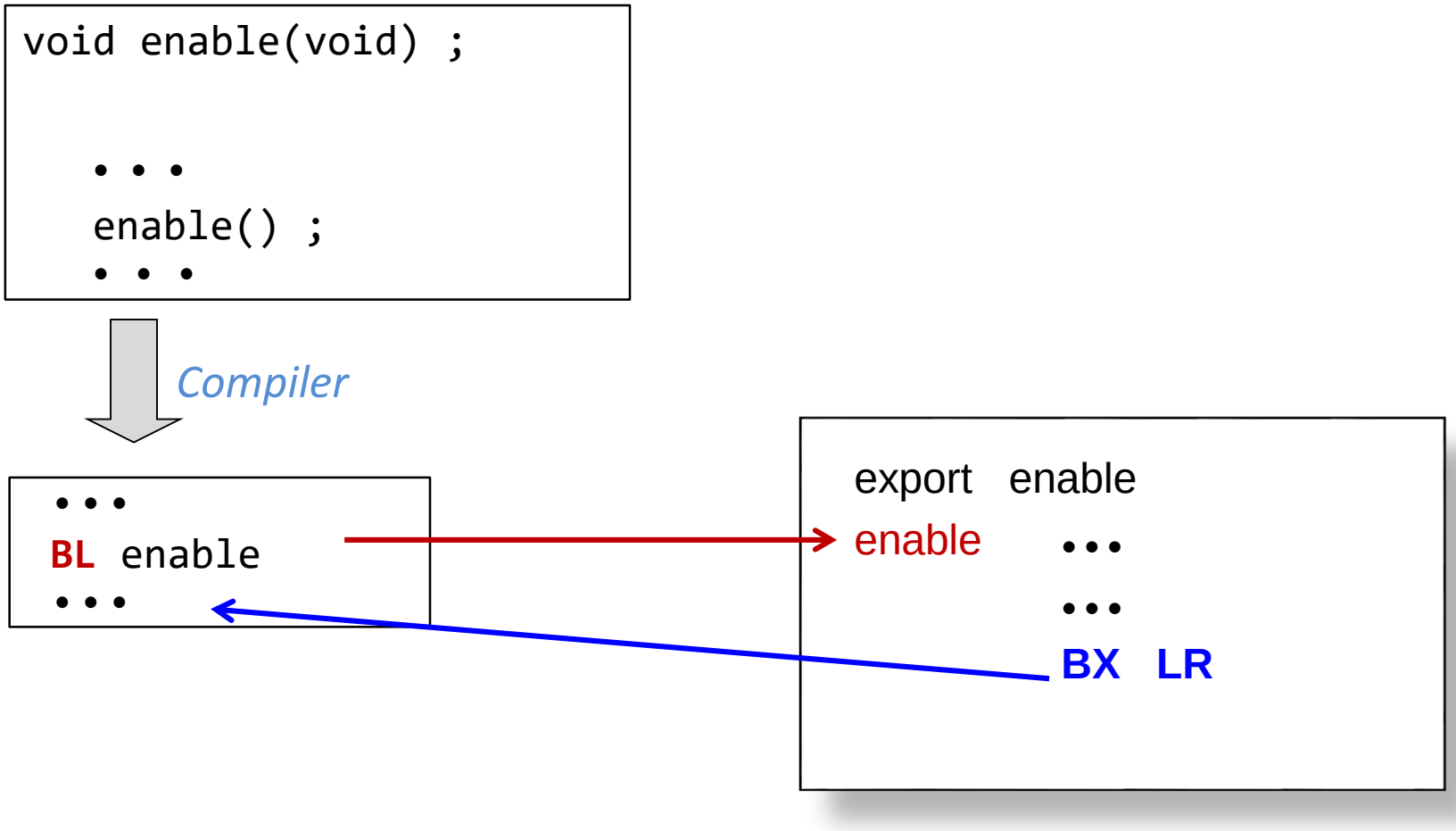
Caller Program

```
MOV r4, #100
...
BL  foo
...
```

Subroutine/Callee

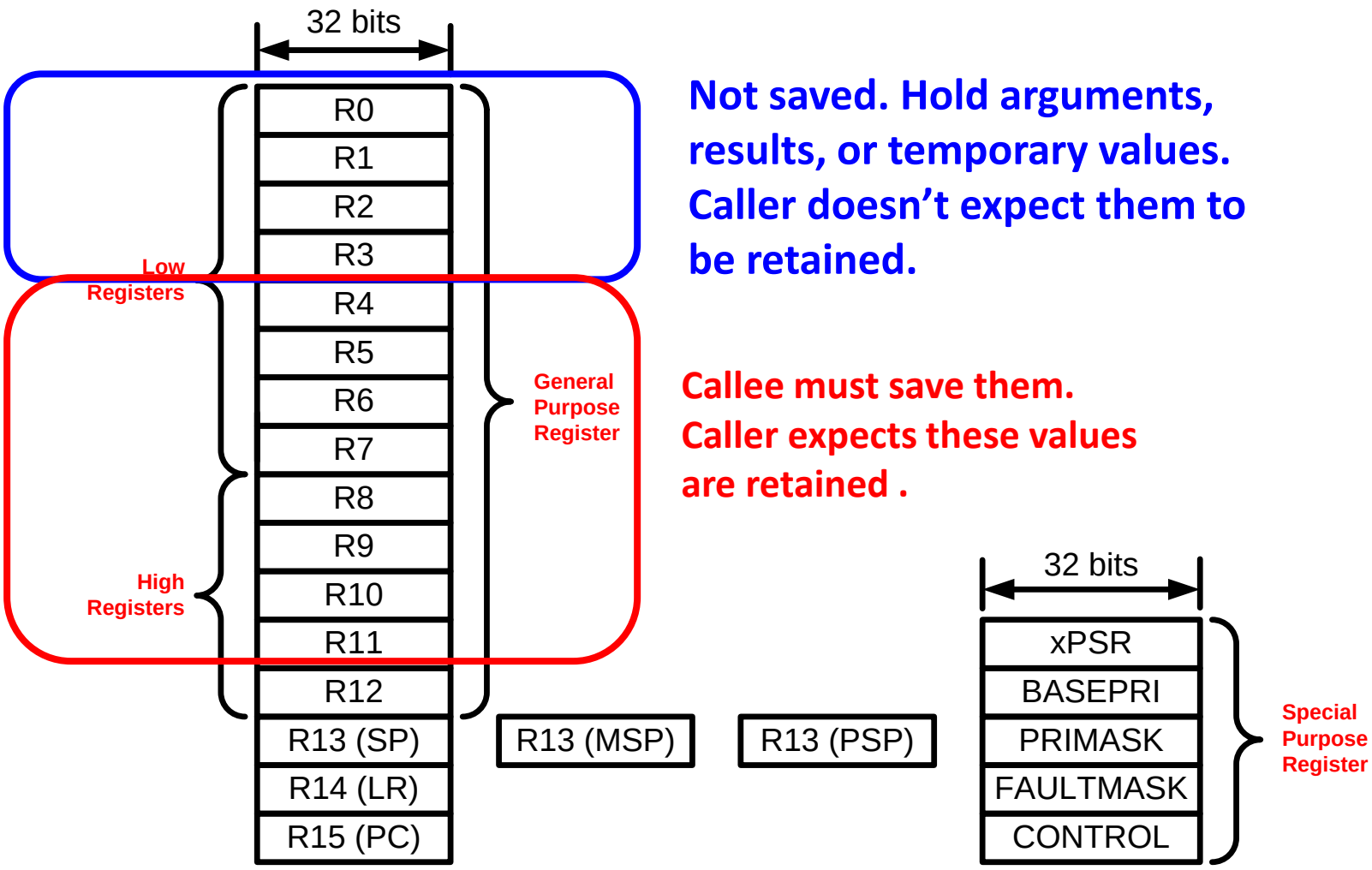
```
foo PROC
    ...
    MOV   r4, #10
    ...
    BX    LR
    ENDP
```

BL and BX


void enable(void) ;
• • •
enable() ;
• • •
Compiler
• • •
BL enable
• • •
export enable
enable
• • •
• • •
BX LR

ARM Procedure Call Standard

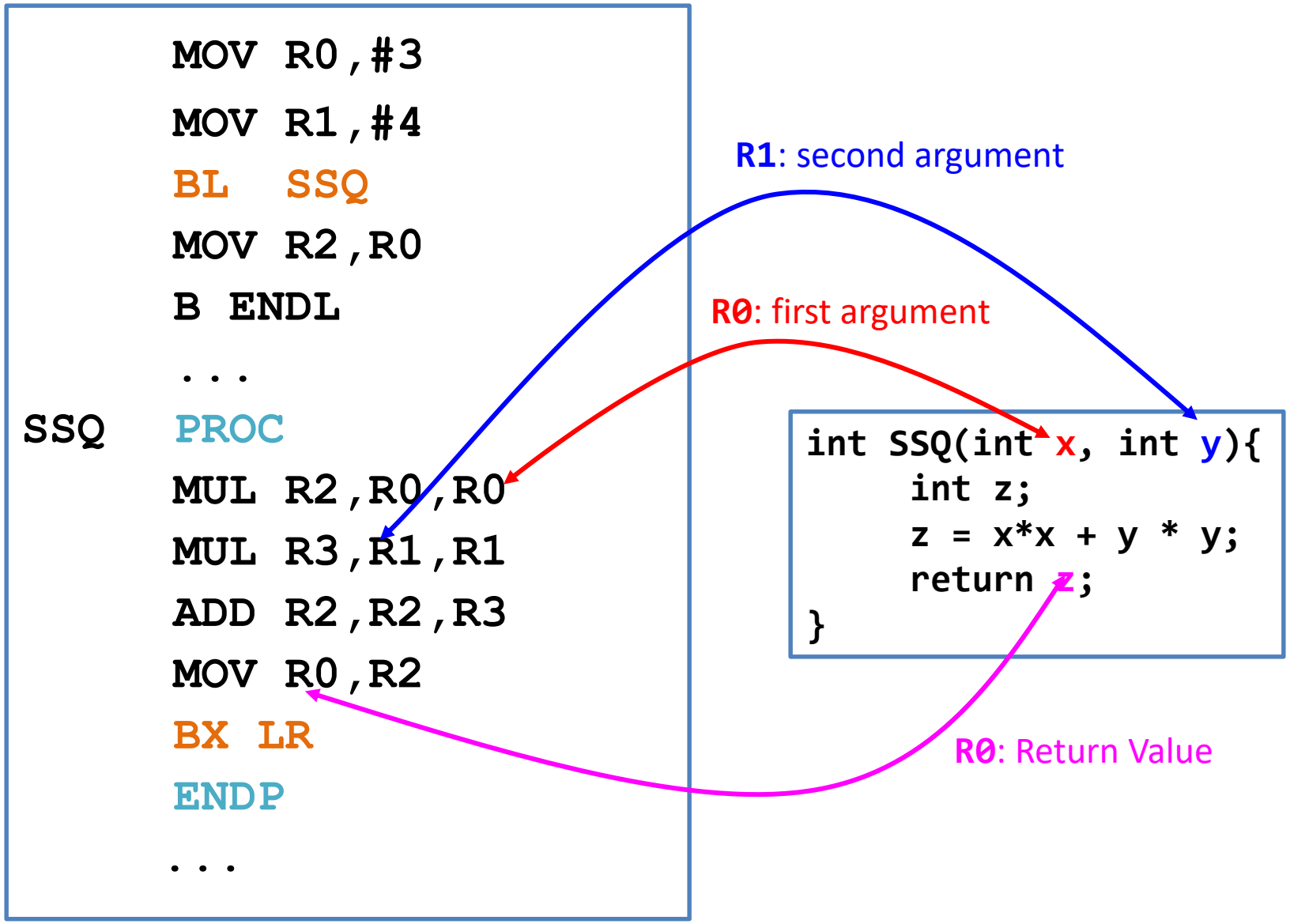
Register	Usage	Subroutine Preserved	Notes
r0	Argument 1 and return value	No	If return has 64 bits, then r0:r1 hold it. If argument 1 has 64 bits, r0:r1 hold it.
r1	Argument 2	No	
r2	Argument 3	No	If the return has 128 bits, r0-r3 hold it.
r3	Argument 4	No	If more than 4 arguments, use the stack
r4	General-purpose V1	Yes	Variable register 1 holds a local variable.
r5	General-purpose V2	Yes	Variable register 2 holds a local variable.
r6	General-purpose V3	Yes	Variable register 3 holds a local variable.
r7	General-purpose V4	Yes	Variable register 4 holds a local variable.
r8	General-purpose V5	YES	Variable register 5 holds a local variable.
r9	Platform specific/V6	No	Usage is platform-dependent.
r10	General-purpose V7	Yes	Variable register 7 holds a local variable.
r11	General-purpose V8	Yes	Variable register 8 holds a local variable.
r12 (IP)	Intra-procedure-call register	No	It holds intermediate values between a procedure and the sub-procedure it calls.
r13 (SP)	Stack pointer	Yes	SP has to be the same after a subroutine has completed.
r14 (LR)	Link register	No	LR does not have to contain the same value after a subroutine has completed.
r15 (PC)	Program counter	N/A	Do not directly change PC



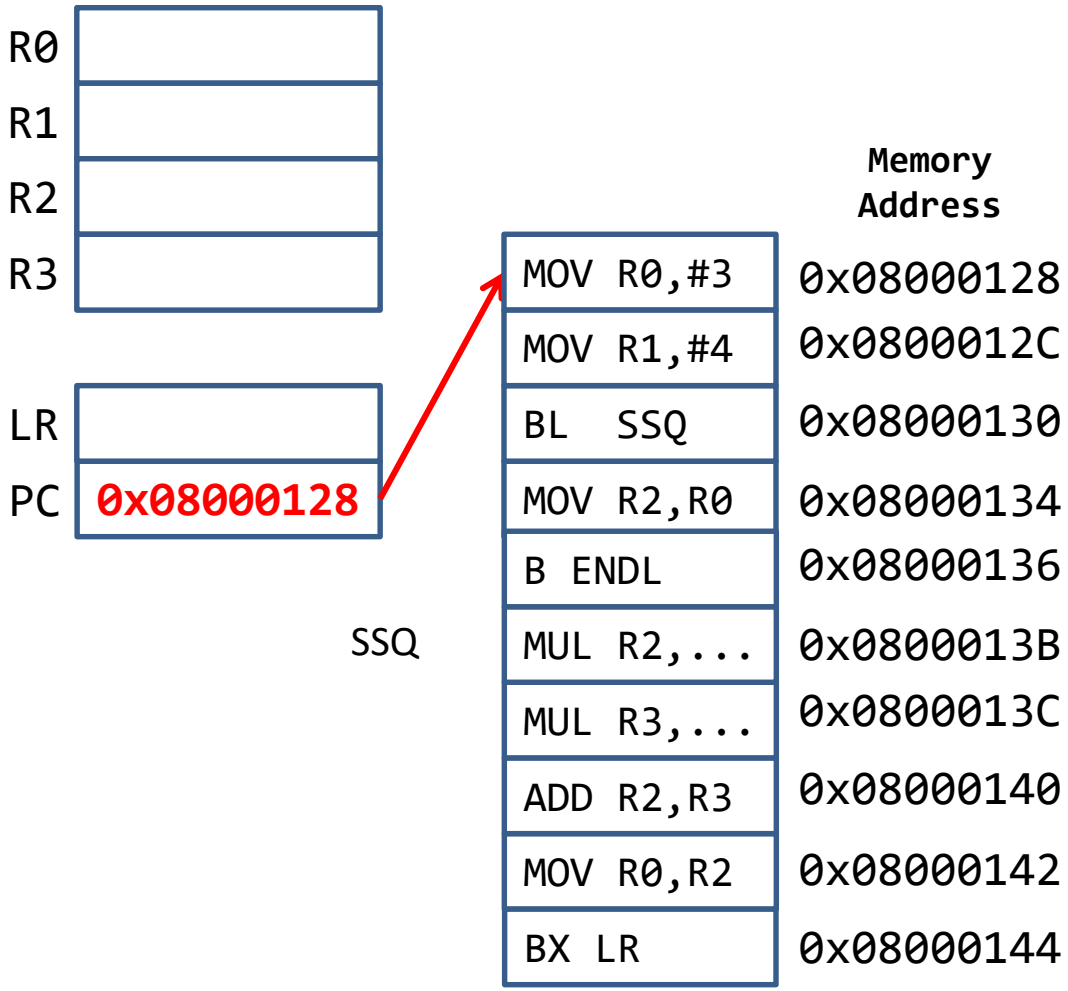
Not saved. Hold arguments, results, or temporary values. Caller doesn't expect them to be retained.

Callee must save them. Caller expects these values are retained .

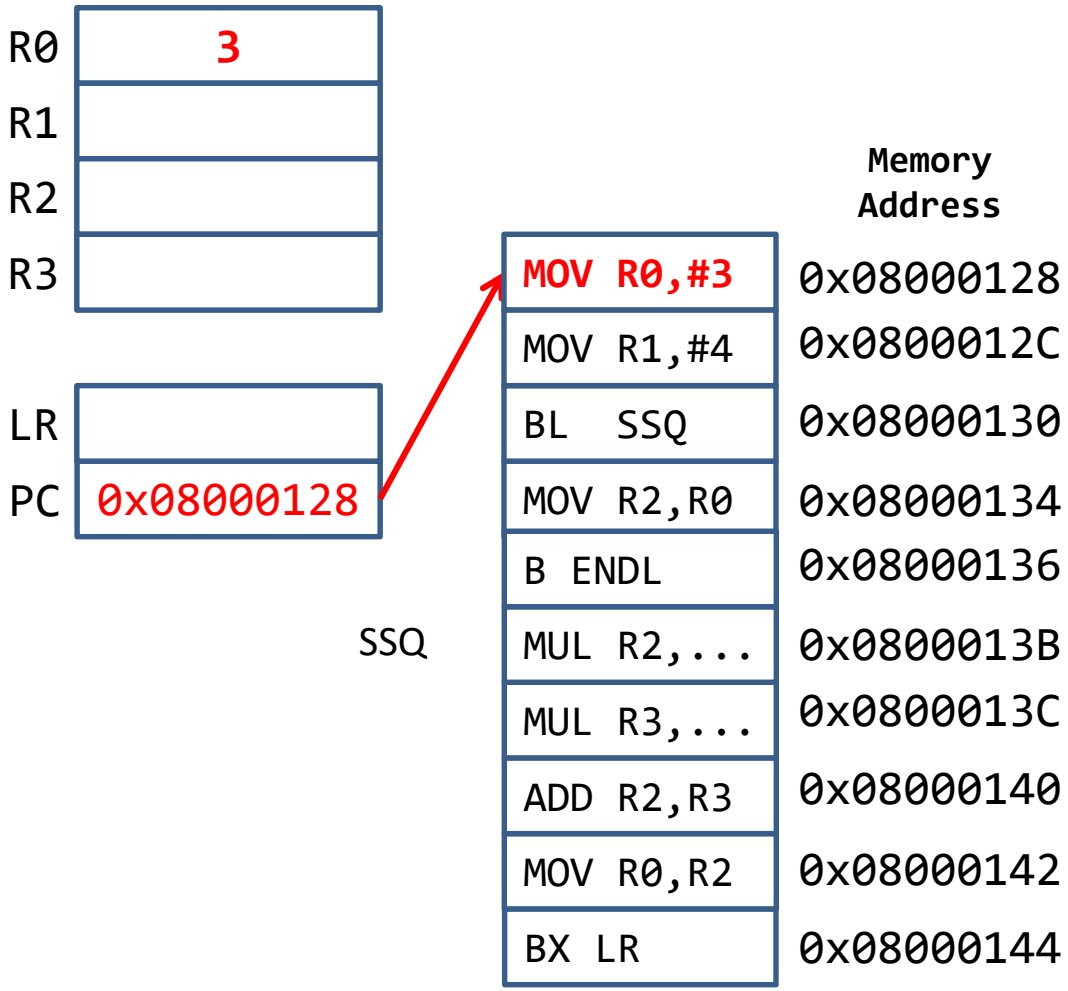
Example: R2 = R0*R0+R1*R1



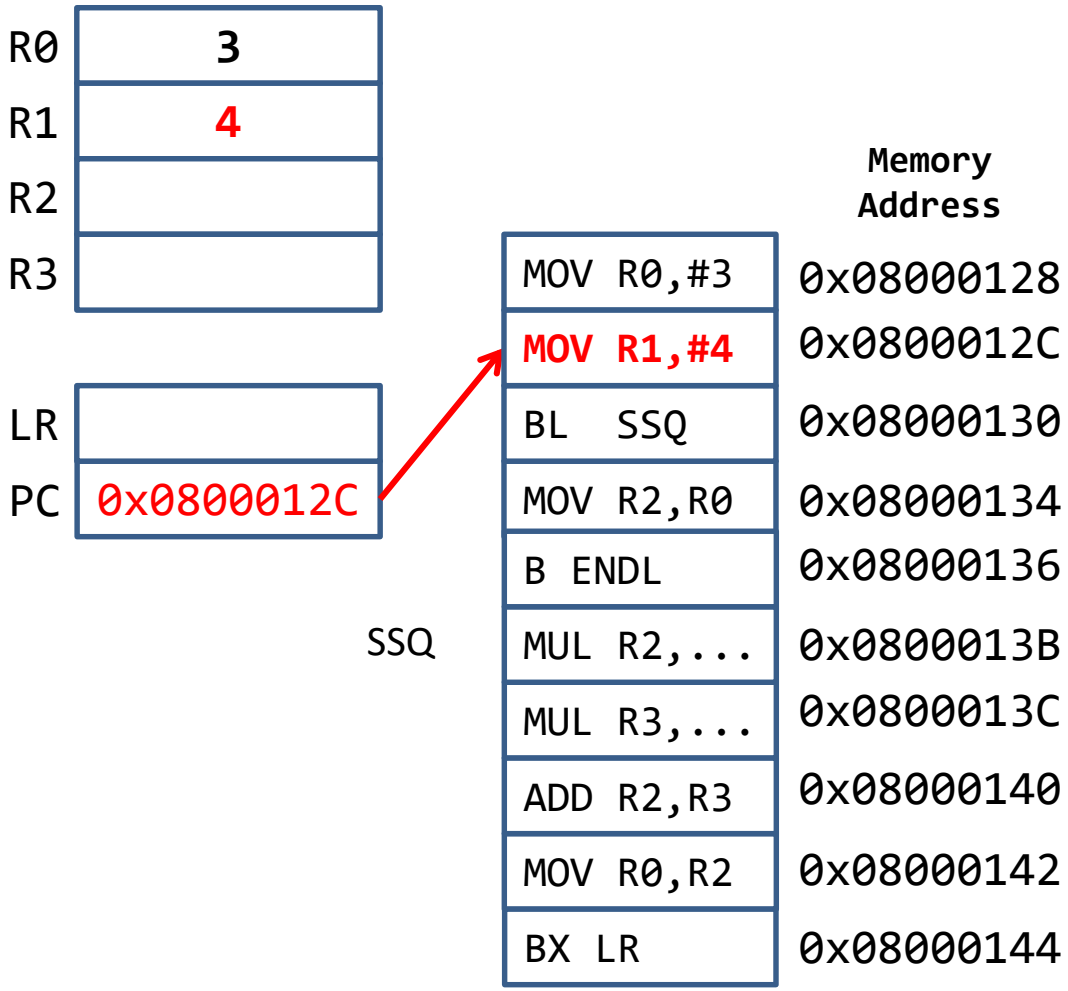
```
        MOV R0,#3
        MOV R1,#4
        BL  SSQ
        MOV R2,R0
        B ENDL
        ...
SSQ     PROC
        MUL R2,R0,R0
        MUL R3,R1,R1
        ADD R2,R2,R3
        MOV R0,R2
        BX  LR
        ENDP
ENDL    ...
```



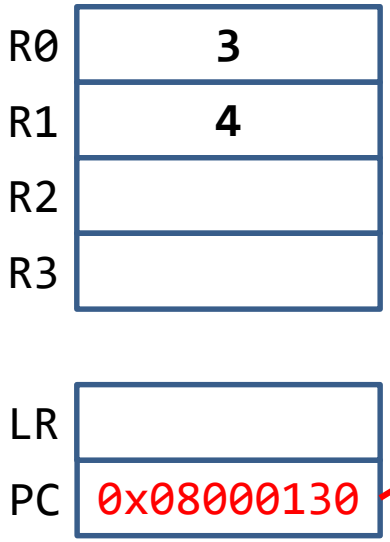
```
        MOV R0,#3
        MOV R1,#4
        BL  SSQ
        MOV R2,R0
        B ENDL
        ...
SSQ     PROC
        MUL R2,R0,R0
        MUL R3,R1,R1
        ADD R2,R2,R3
        MOV R0,R2
        BX  LR
        ENDP
ENDL    ...
```

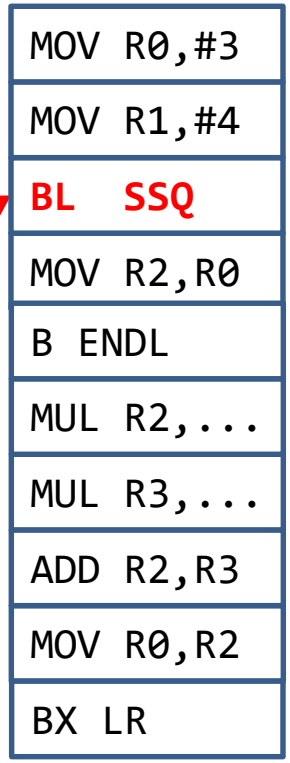


```
        MOV R0,#3
        MOV R1,#4
        BL  SSQ
        MOV R2,R0
        B ENDL
        ...
SSQ     PROC
        MUL R2,R0,R0
        MUL R3,R1,R1
        ADD R2,R2,R3
        MOV R0,R2
        BX  LR
        ENDP
ENDL    ...
```



```
        MOV R0,#3
        MOV R1,#4
        BL  SSQ
        MOV R2,R0
        B ENDL
        ...
SSQ     PROC
        MUL R2,R0,R0
        MUL R3,R1,R1
        ADD R2,R2,R3
        MOV R0,R2
        BX  LR
        ENDP
ENDL    ...
```





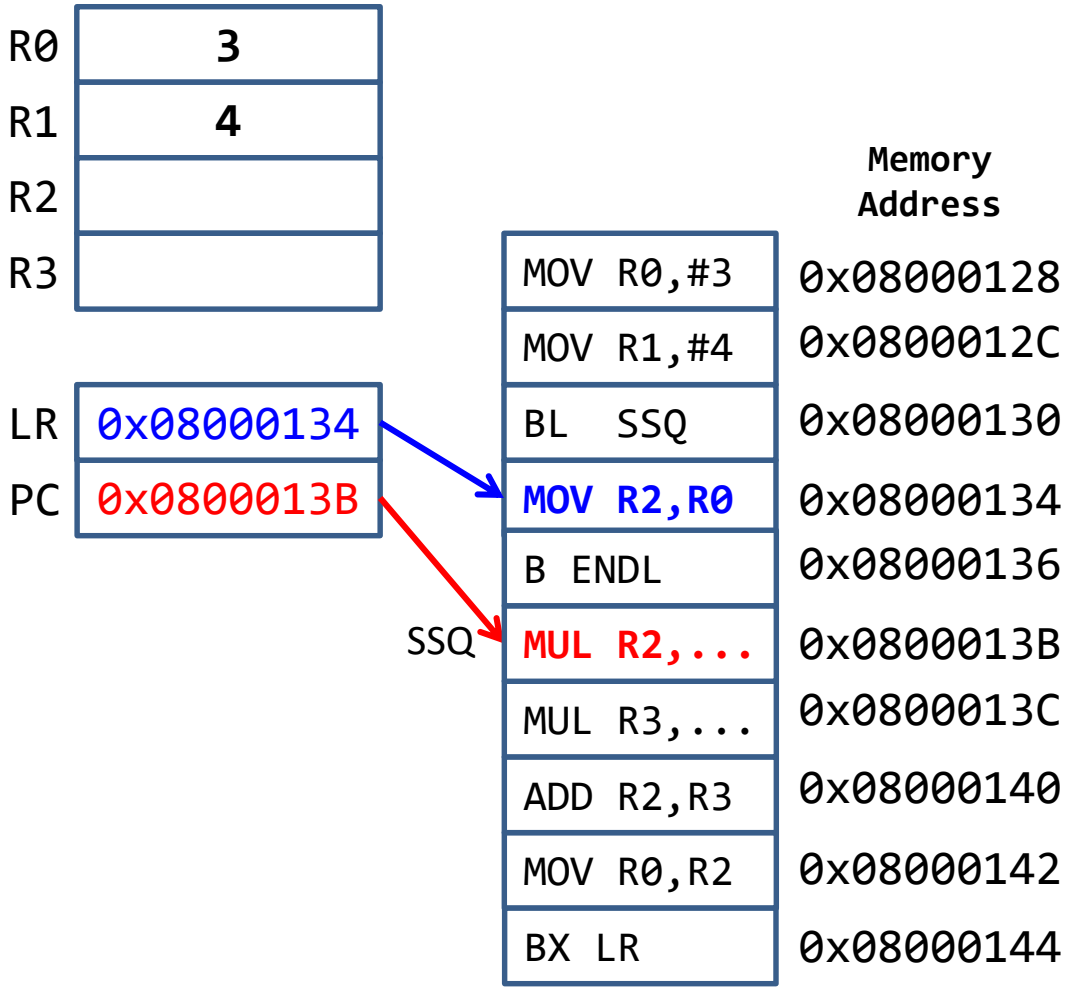
SSQ

Memory Address

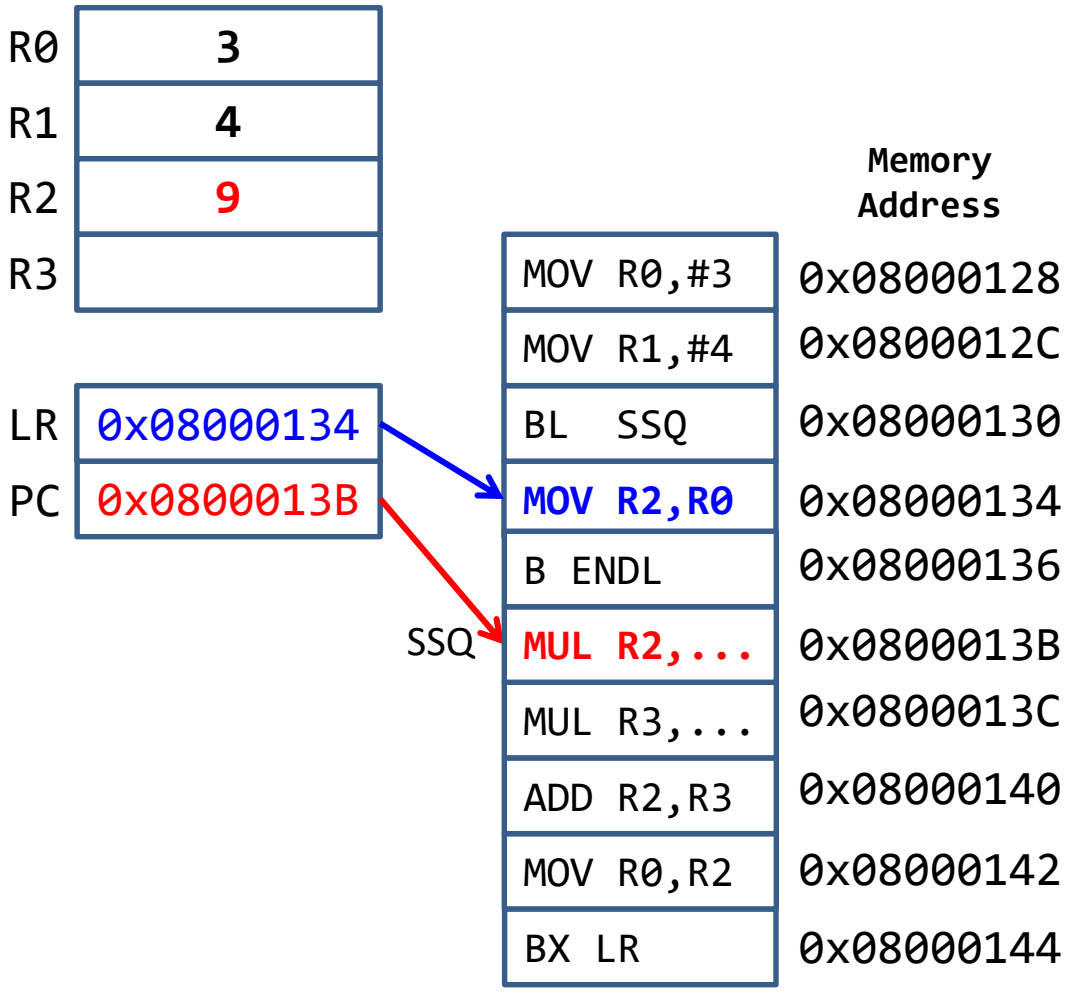
0x08000128
0x0800012C
0x08000130
0x08000134
0x08000136
0x0800013B
0x0800013C
0x08000140
0x08000142
0x08000144

```
        MOV R0,#3
        MOV R1,#4
        BL  SSQ
        MOV R2,R0
        B ENDL
        ...
SSQ     PROC
        MUL R2,R0,R0
        MUL R3,R1,R1
        ADD R2,R2,R3
        MOV R0,R2
        BX  LR
        ENDP
ENDL    ...
```

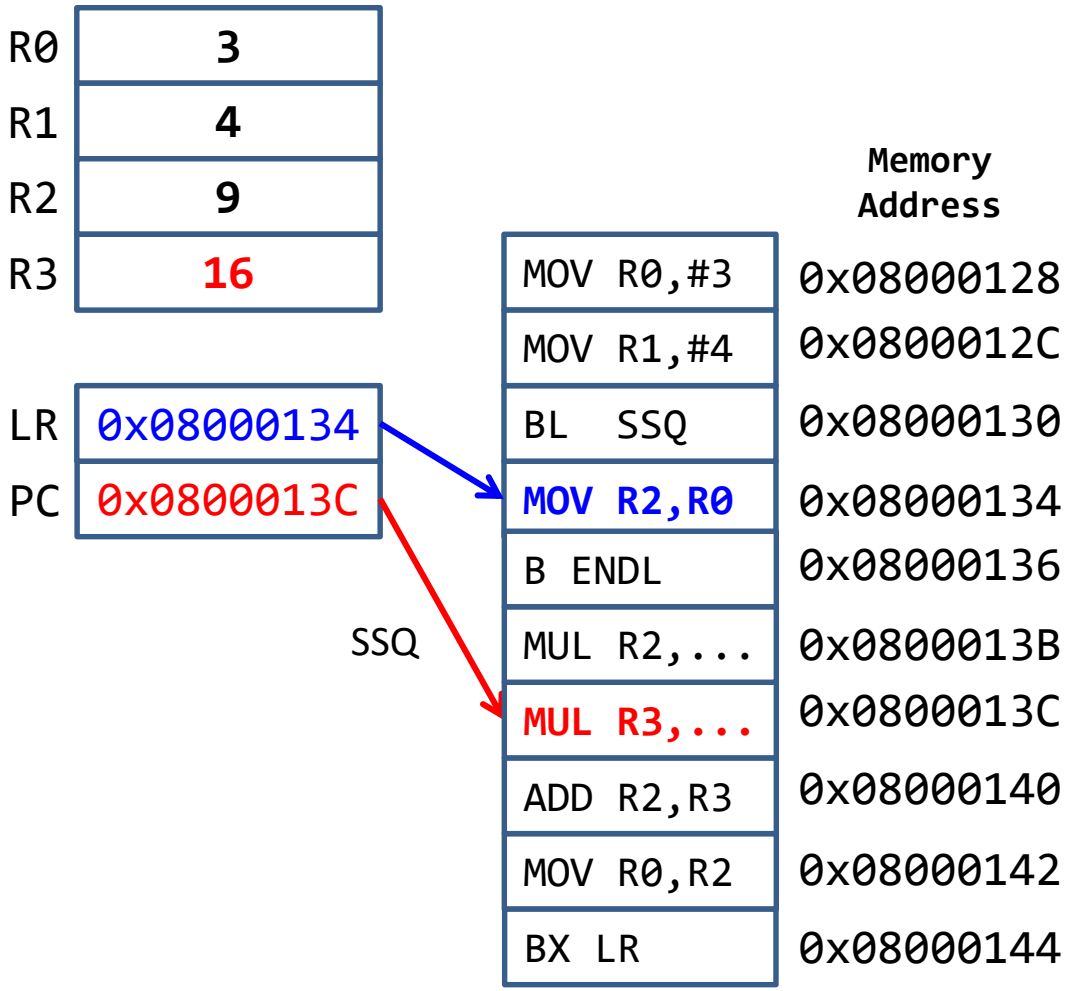
Address of the next instruction after the branch is saved into LR.



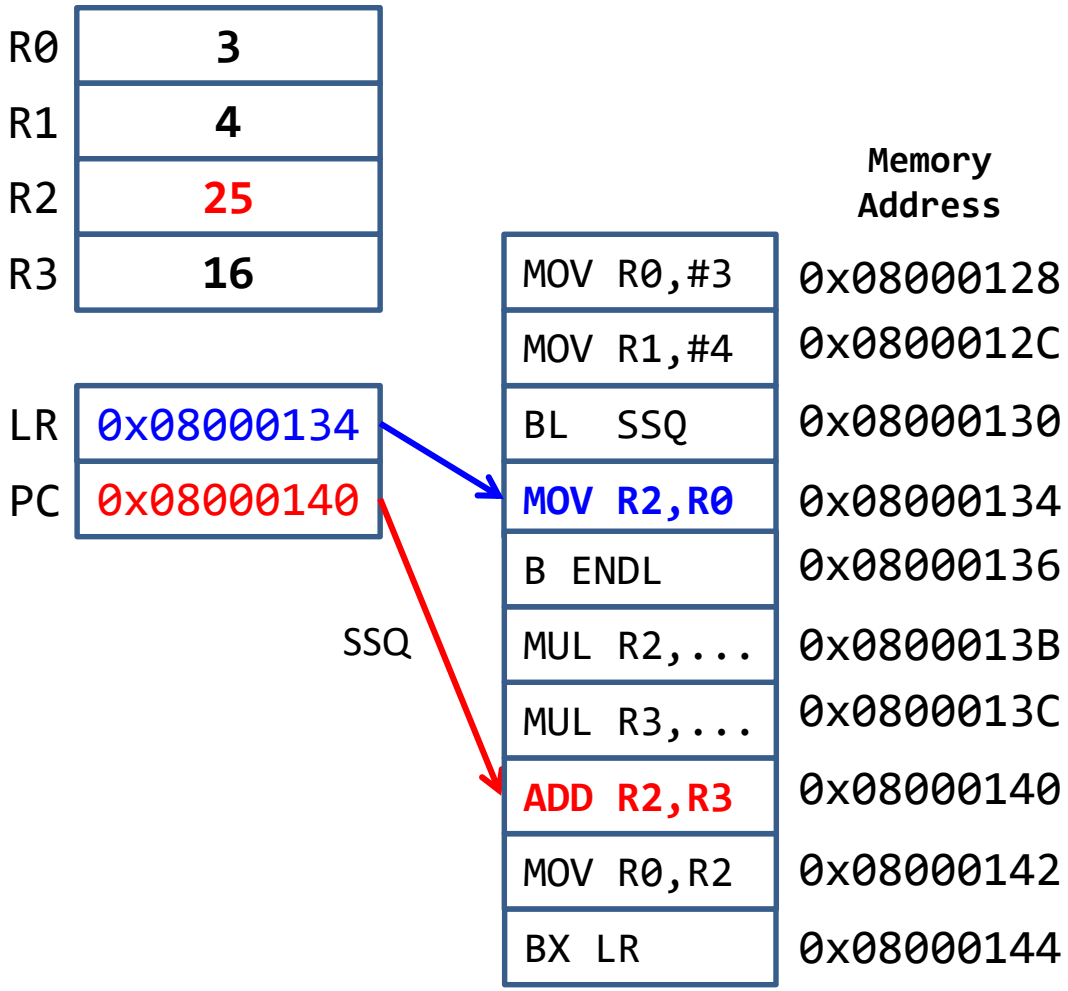
```
        MOV R0,#3
        MOV R1,#4
        BL  SSQ
        MOV R2,R0
        B ENDL
        ...
SSQ     PROC
        MUL R2,R0,R0
        MUL R3,R1,R1
        ADD R2,R2,R3
        MOV R0,R2
        BX  LR
        ENDP
ENDL    ...
```

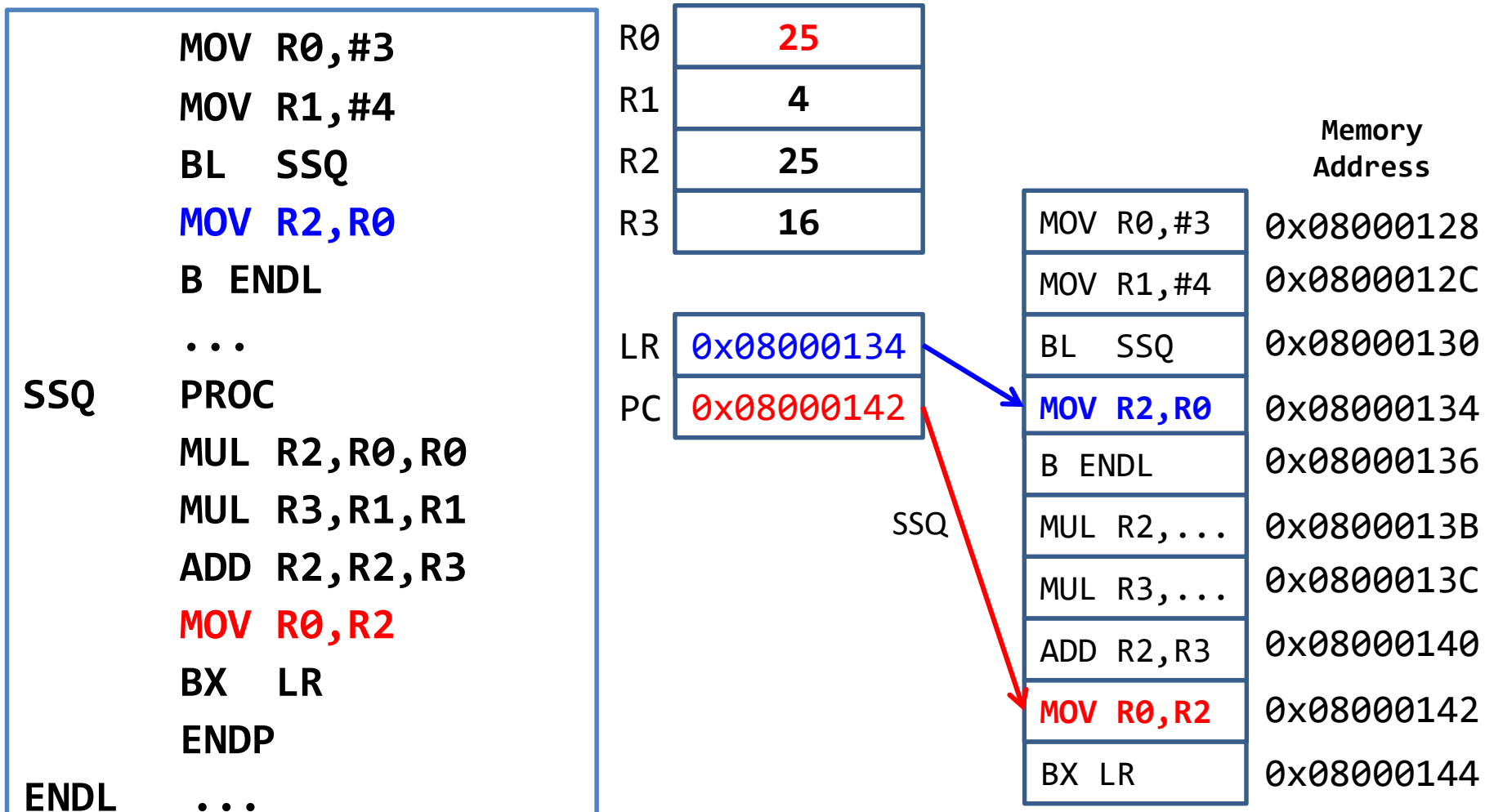


```
        MOV R0,#3
        MOV R1,#4
        BL  SSQ
        MOV R2,R0
        B ENDL
        ...
SSQ     PROC
        MUL R2,R0,R0
        MUL R3,R1,R1
        ADD R2,R2,R3
        MOV R0,R2
        BX  LR
        ENDP
ENDL    ...
```

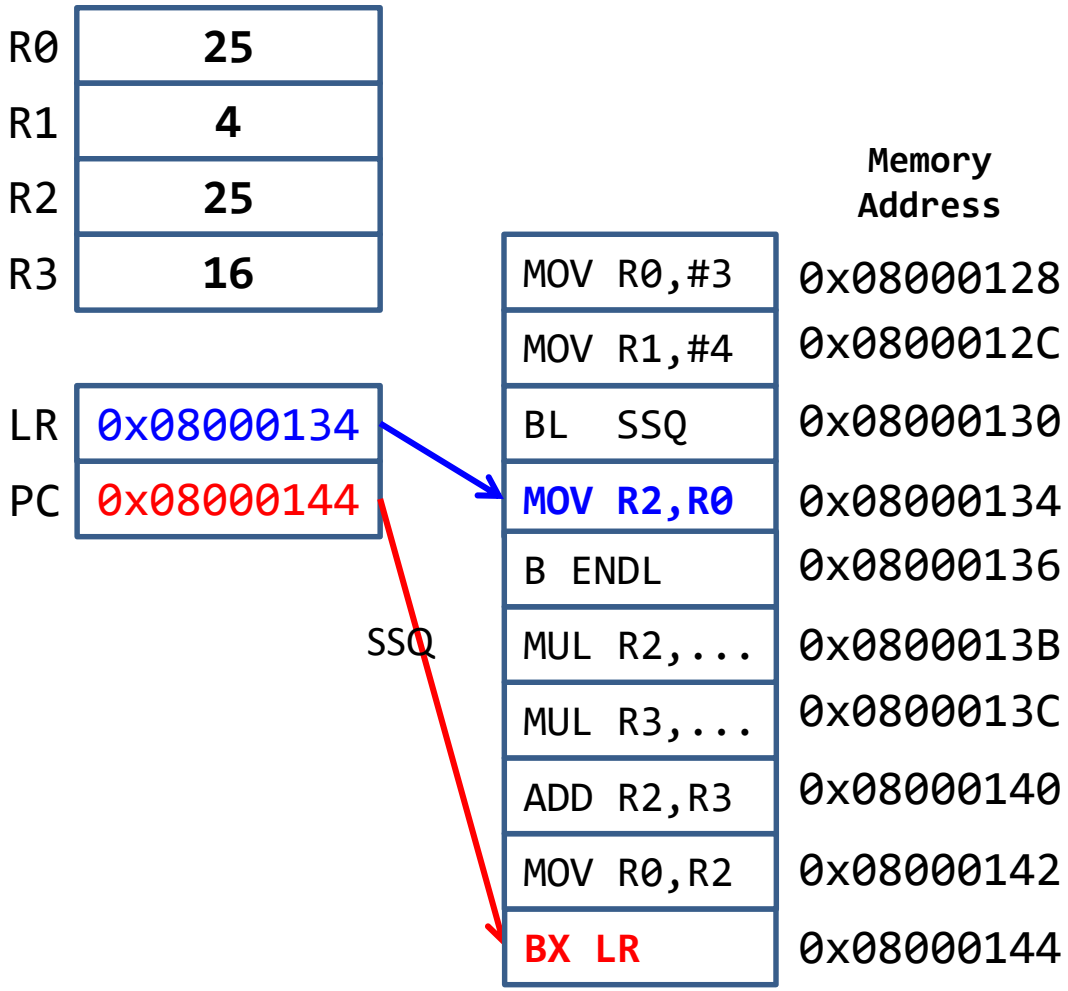


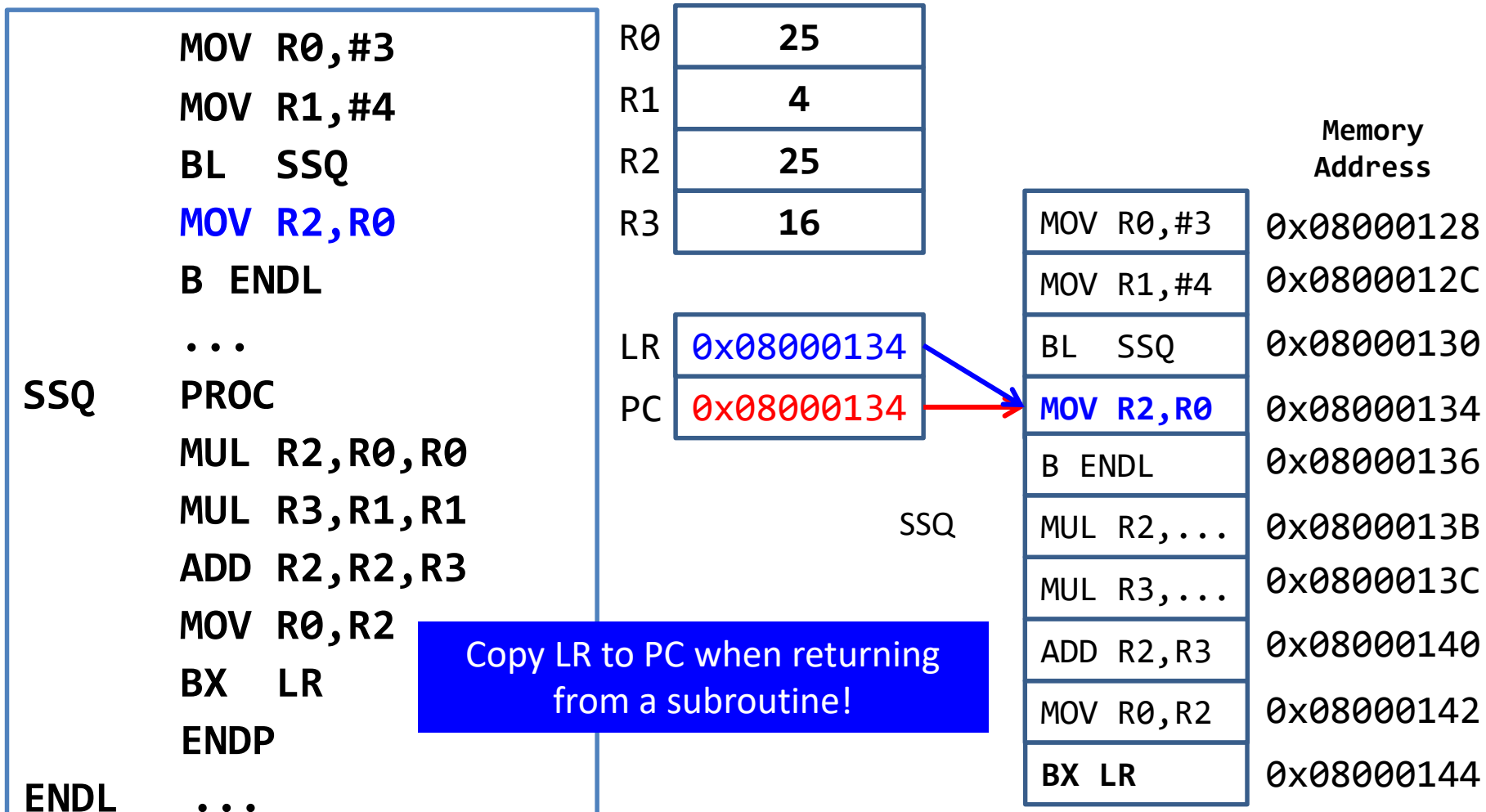
```
        MOV R0,#3
        MOV R1,#4
        BL  SSQ
        MOV R2,R0
        B ENDL
        ...
SSQ     PROC
        MUL R2,R0,R0
        MUL R3,R1,R1
        ADD R2,R2,R3
        MOV R0,R2
        BX  LR
        ENDP
ENDL    ...
```



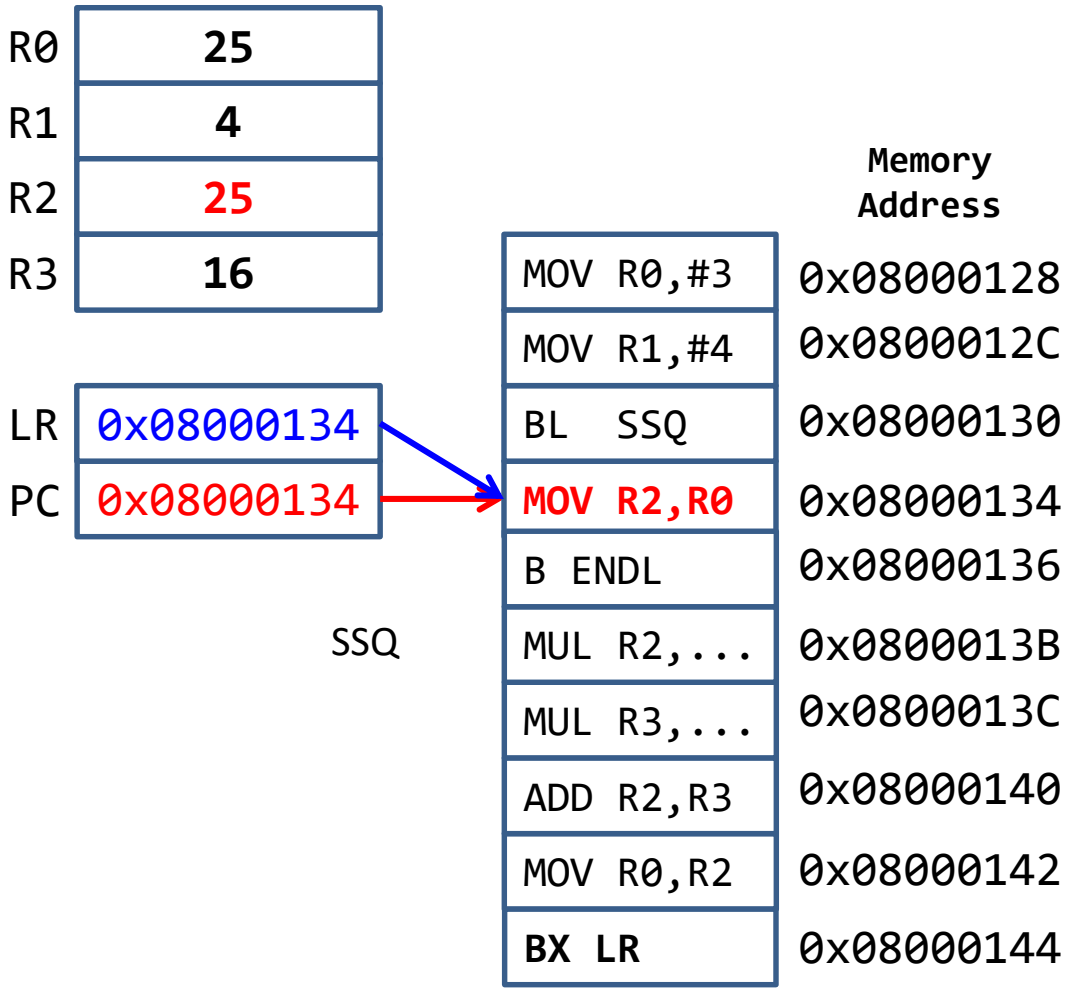

MOV R0,#3
MOV R1,#4
BL SSQ
MOV R2,R0
B ENDL
...
SSQ PROC
MUL R2,R0,R0
MUL R3,R1,R1
ADD R2,R2,R3
MOV R0,R2
BX LR
ENDP
ENDL ...
R0
25
R1
4
R2
25
R3
16
LR
0x08000134
PC
0x08000142
SSQ
Memory Address
MOV R0,#3
0x08000128
MOV R1,#4
0x0800012C
BL SSQ
0x08000130
MOV R2,R0
0x08000134
B ENDL
0x08000136
MUL R2,...
0x0800013B
MUL R3,...
0x0800013C
ADD R2,R3
0x08000140
MOV R0,R2
0x08000142
BX LR
0x08000144

```
        MOV R0,#3
        MOV R1,#4
        BL  SSQ
        MOV R2,R0
        B ENDL
        ...
SSQ     PROC
        MUL R2,R0,R0
        MUL R3,R1,R1
        ADD R2,R2,R3
        MOV R0,R2
        BX  LR
        ENDP
ENDL    ...
```

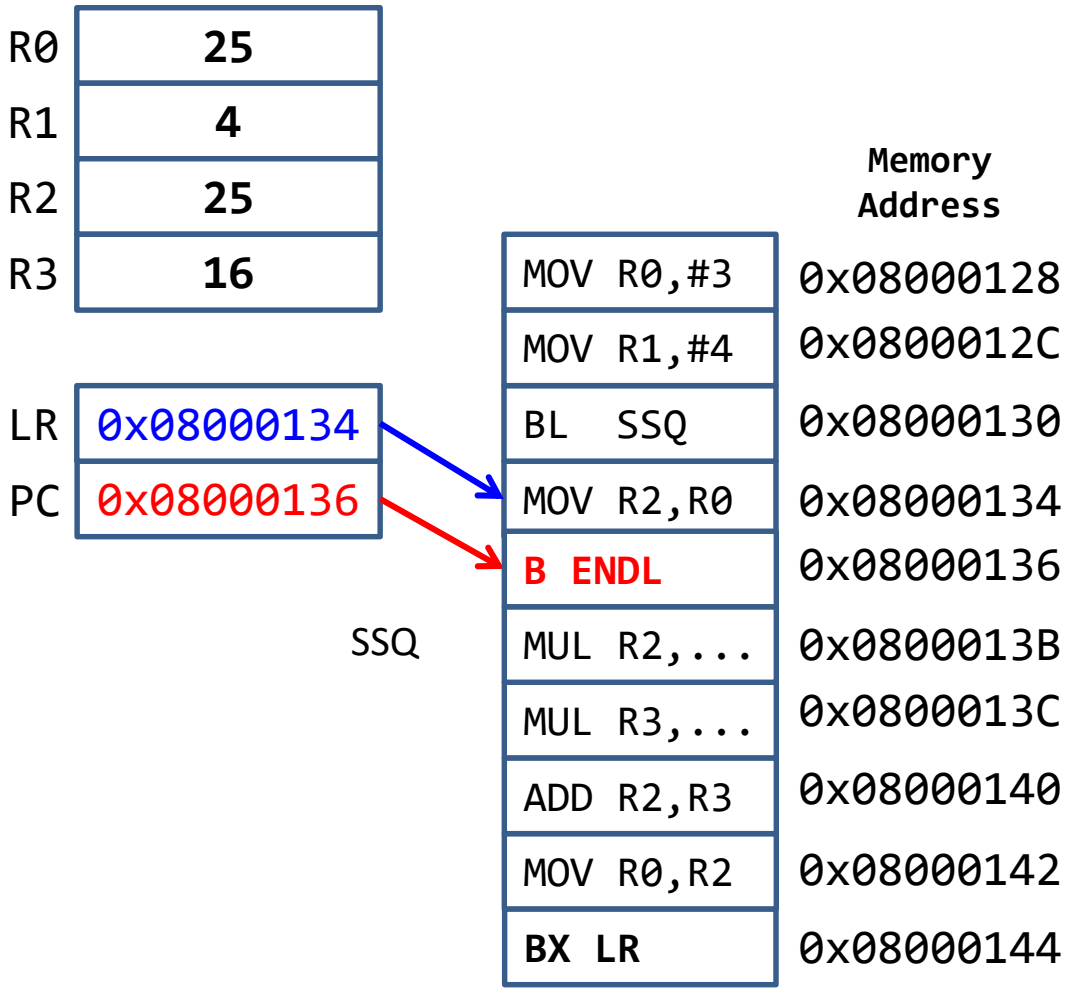



MOV R0,#3
MOV R1,#4
BL SSQ
MOV R2,R0
B ENDL
...
SSQ PROC
MUL R2,R0,R0
MUL R3,R1,R1
ADD R2,R2,R3
MOV R0,R2
BX LR
ENDP
ENDL ...
R0 25
R1 4
R2 25
R3 16
LR 0x08000134
PC 0x08000134
SSQ
Copy LR to PC when returning from a subroutine!
Memory Address
MOV R0,#3 0x08000128
MOV R1,#4 0x0800012C
BL SSQ 0x08000130
MOV R2,R0 0x08000134
B ENDL 0x08000136
MUL R2,... 0x0800013B
MUL R3,... 0x0800013C
ADD R2,R3 0x08000140
MOV R0,R2 0x08000142
BX LR 0x08000144

```
        MOV R0,#3
        MOV R1,#4
        BL  SSQ
        MOV R2,R0
        B ENDL
        ...
SSQ     PROC
        MUL R2,R0,R0
        MUL R3,R1,R1
        ADD R2,R2,R3
        MOV R0,R2
        BX  LR
        ENDP
ENDL    ...
```

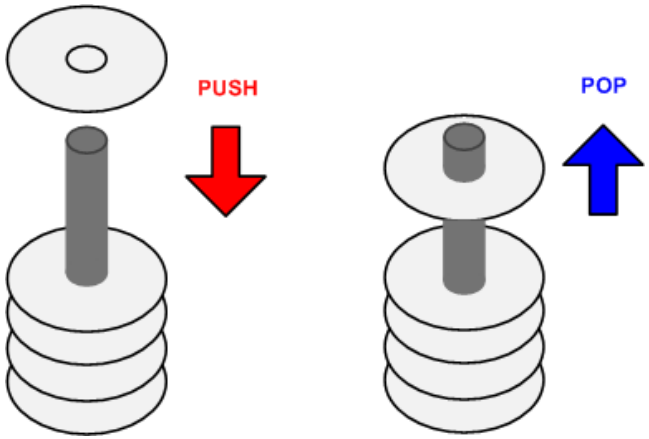


```
        MOV R0,#3
        MOV R1,#4
        BL  SSQ
        MOV R2,R0
        B ENDL
        ...
SSQ     PROC
        MUL R2,R0,R0
        MUL R3,R1,R1
        ADD R2,R2,R3
        MOV R0,R2
        BX  LR
        ENDP
ENDL    ...
```

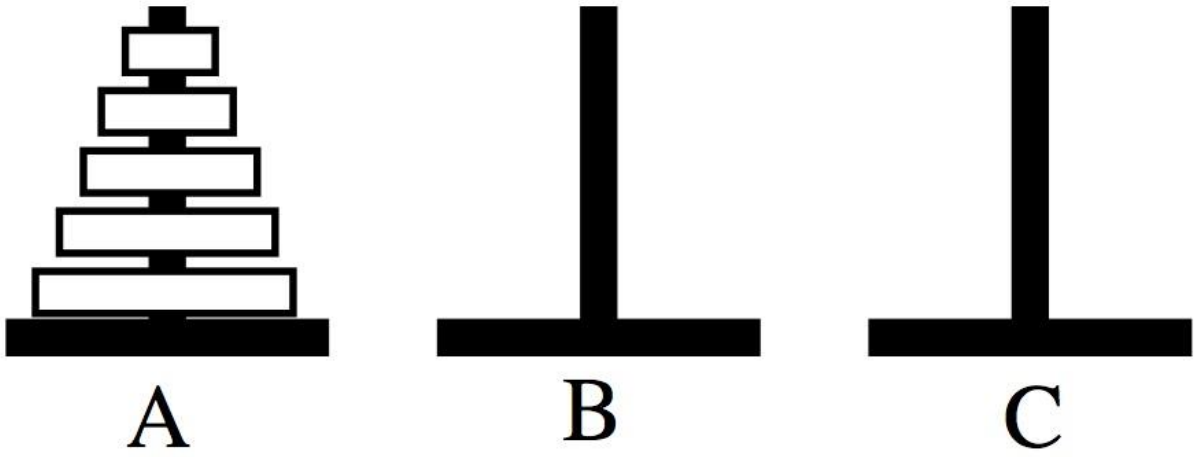


Stack

- A **Last-In-First-Out** data structure
- Only allow to access the most recently added item
  - Also called the top of the stack
- Key operations:
  - push (add item to stack)
  - pop (remove top item from stack)



Stack ex: Tower of Hanoi



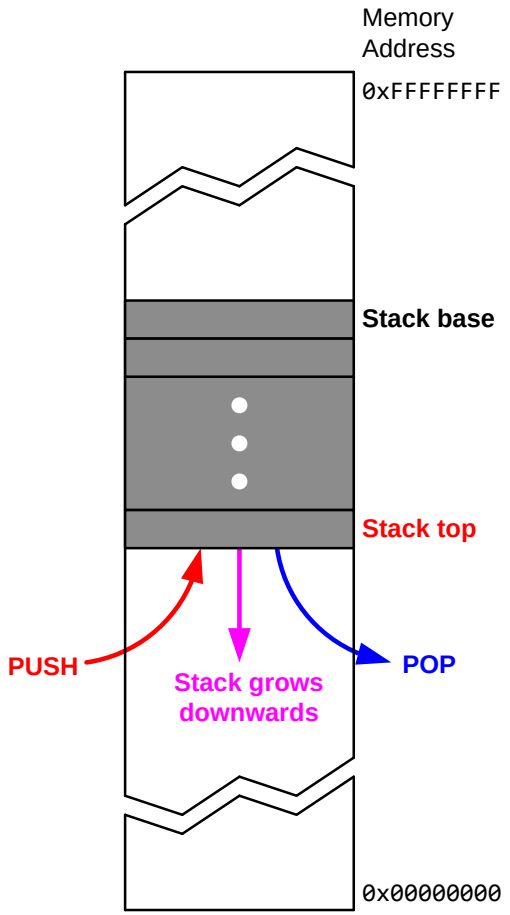
- Only one disk may be moved at a time.
- Each move consists of taking the upper disk from one of the rods and sliding it onto another rod, on top of the other disks that may already be present on that rod.
- **No disk may be placed on top of a smaller disk.**

STACK: Last In First Out

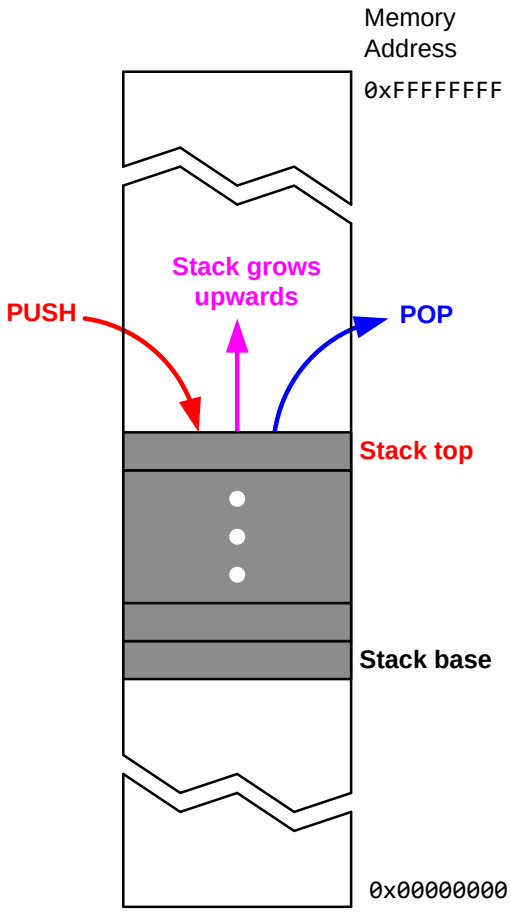
Stack 1 **Stack 2** **Stack 3**

http://en.wikipedia.org/wiki/File:Tower_of_Hanoi_4.gif

Stack Growth Convention:
Ascending *vs* Descending



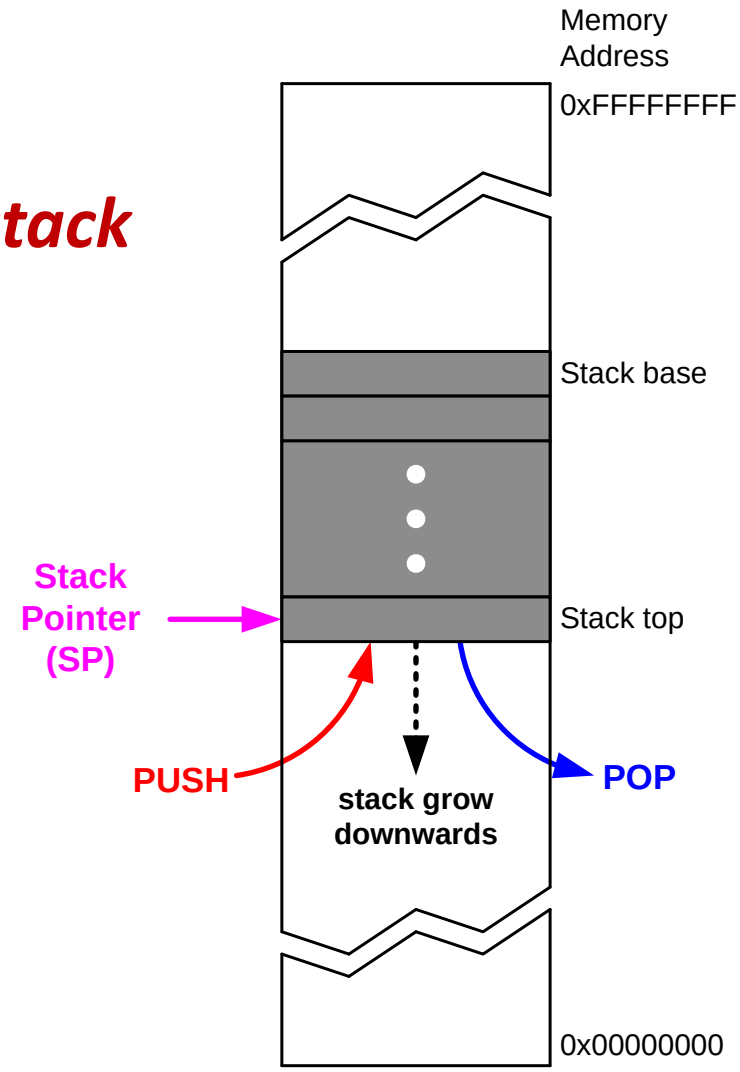
Descending stack: Stack grows towards low memory address



Ascending stack: Stack grows towards high memory address

Cortex-M Stack

- stack pointer (SP) = R13
- Cortex-M uses ***full descending stack***
- stack pointer
  - decremented on **PUSH**
  - incremented on **POP**



PUSH {*Rd*}

- SP = SP-4 ⟶ descending stack
- (*SP) = *Rd* ⟶ full stack

(SP points to the last item pushed onto the stack)

Push multiple registers

They are equivalent.
(push large first)

```
PUSH {r6, r7, r8}   ⟺   PUSH {r8, r7, r6}   ⟺   PUSH {r8}
                                                  PUSH {r7}
                         PUSH {r6-r8}             PUSH {r6}
```

- The order in which registers listed in the register list does not matter.
- When pushing multiple registers, these registers are automatically sorted by name and the lowest-numbered register is stored to the lowest memory address, *i.e.* is stored last.

POP {*Rd*}

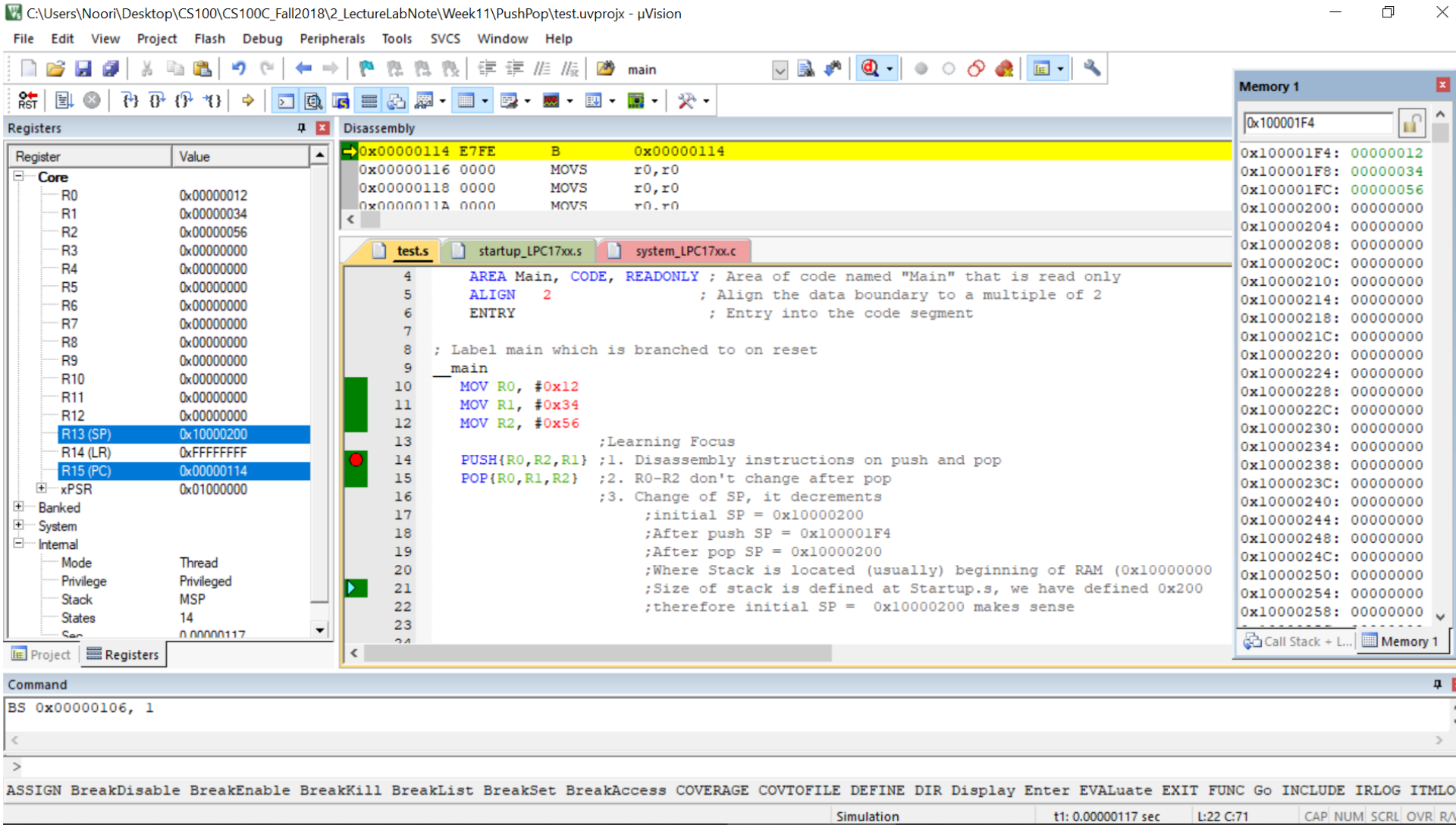
- *Rd* = (*SP) ⟶ full stack
- SP = SP + 4 ⟶ Stack shrinks

Pop multiple registers

They are equivalent.
Pop small first

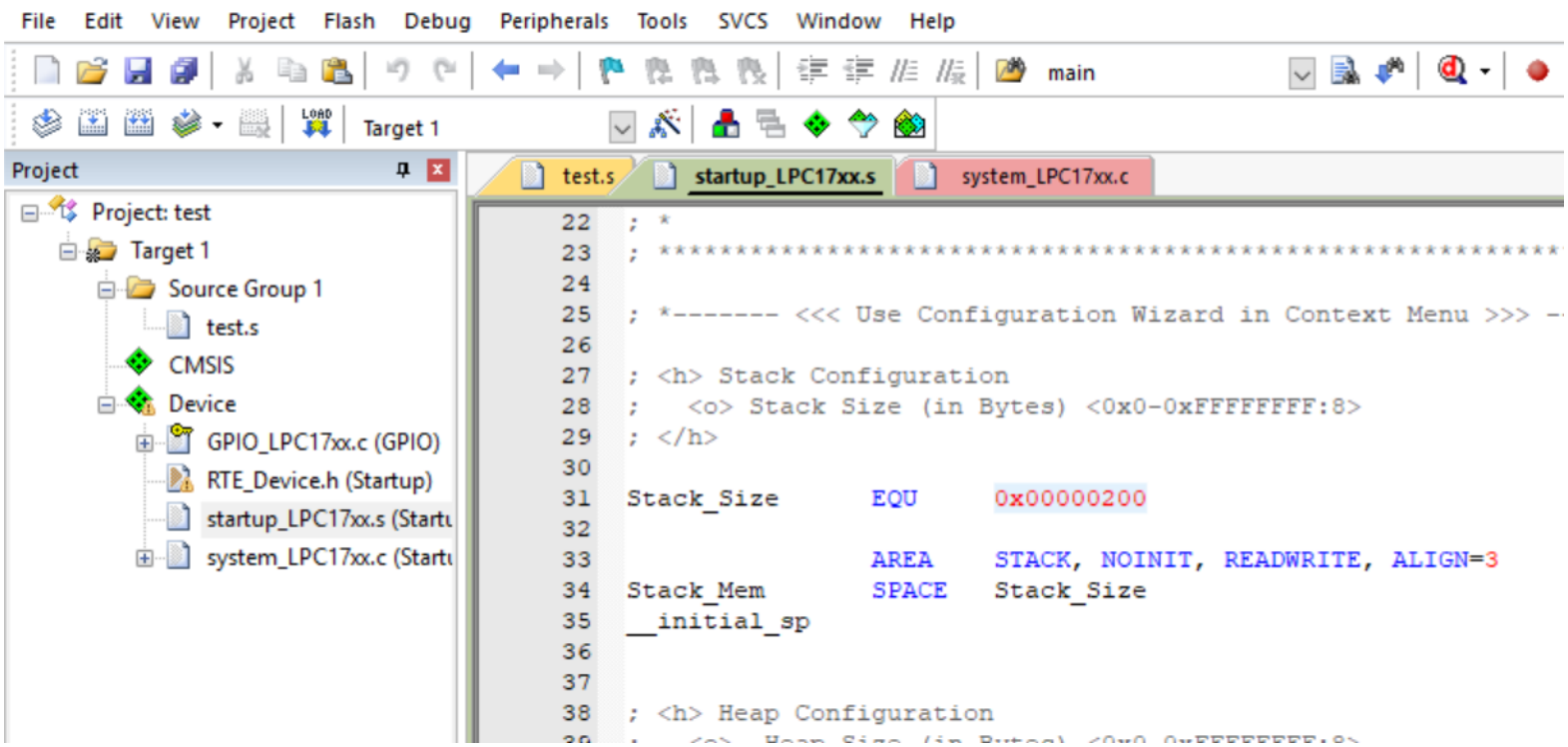
```
POP {r6, r7, r8}   ⟺   POP {r8, r7, r6}   ⟺   POP {r6}
                                               POP {r7}
                                               POP {r8}
```

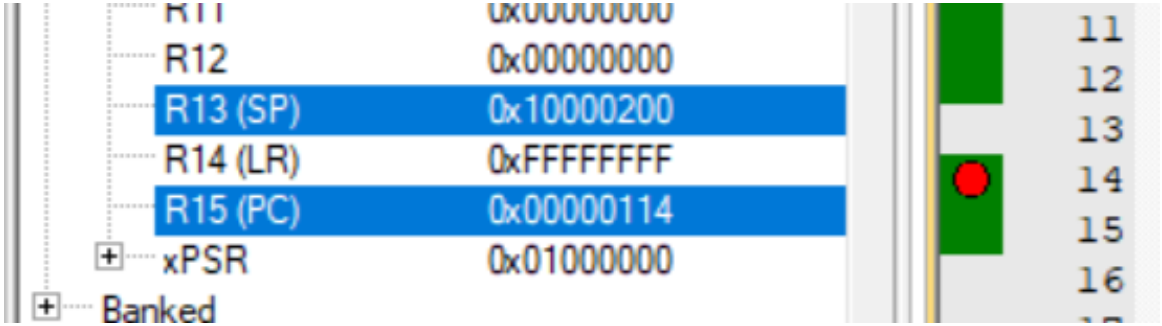
- The order in which registers listed in the register list does not matter.
- When popping multiple registers, these registers are automatically sorted by name and the lowest-numbered register is loaded from the lowest memory address, *i.e.* is loaded first.


C:\Users\Noori\Desktop\CS100\CS100C_Fall2018\2_LectureLabNote\Week11\PushPop\test.uvprojx - µVision
File Edit View Project Flash Debug Peripherals Tools SVCS Window Help
main
Registers
Register Value
Core
R0 0x00000012
R1 0x00000034
R2 0x00000056
R3 0x00000000
R4 0x00000000
R5 0x00000000
R6 0x00000000
R7 0x00000000
R8 0x00000000
R9 0x00000000
R10 0x00000000
R11 0x00000000
R12 0x00000000
R13 (SP) 0x10000200
R14 (LR) 0xFFFFFFFF
R15 (PC) 0x00000114
xPSR 0x01000000
Banked
System
Internal
Mode Thread
Privilege Privileged
Stack MSP
States 14
Project Registers
Disassembly
0x00000114 E7FE B 0x00000114
0x00000116 0000 MOVS r0,r0
0x00000118 0000 MOVS r0,r0
test.s startup_LPC17xx.s system_LPC17xx.c
4 AREA Main, CODE, READONLY ; Area of code named "Main" that is read only
5 ALIGN 2 ; Align the data boundary to a multiple of 2
6 ENTRY ; Entry into the code segment
7
8 ; Label main which is branched to on reset
9 __main
10 MOV R0, #0x12
11 MOV R1, #0x34
12 MOV R2, #0x56
13 ;Learning Focus
14 PUSH{R0,R2,R1} ;1. Disassembly instructions on push and pop
15 POP{R0,R1,R2} ;2. R0-R2 don't change after pop
16 ;3. Change of SP, it decrements
17 ;initial SP = 0x10000200
18 ;After push SP = 0x100001F4
19 ;After pop SP = 0x10000200
20 ;Where Stack is located (usually) beginning of RAM (0x10000000
21 ;Size of stack is defined at Startup.s, we have defined 0x200
22 ;therefore initial SP = 0x10000200 makes sense
23
Memory 1
0x100001F4
0x100001F4: 00000012
0x100001F8: 00000034
0x100001FC: 00000056
0x10000200: 00000000
0x10000204: 00000000
0x10000208: 00000000
0x1000020C: 00000000
0x10000210: 00000000
0x10000214: 00000000
0x10000218: 00000000
0x1000021C: 00000000
0x10000220: 00000000
0x10000224: 00000000
0x10000228: 00000000
0x1000022C: 00000000
0x10000230: 00000000
0x10000234: 00000000
0x10000238: 00000000
0x1000023C: 00000000
0x10000240: 00000000
0x10000244: 00000000
0x10000248: 00000000
0x1000024C: 00000000
0x10000250: 00000000
0x10000254: 00000000
0x10000258: 00000000
Call Stack + L... Memory 1
Command
BS 0x00000106, 1
ASSIGN BreakDisable BreakEnable BreakKill BreakList BreakSet BreakAccess COVERAGE COVTOFILE DEFINE DIR Display Enter EVALuate EXIT FUNC Go INCLUDE IRLOG ITMLO
Simulation t1: 0.00000117 sec L:22 C:71 CAP NUM SCRL OVR R/

Initializing the stack pointer (SP)

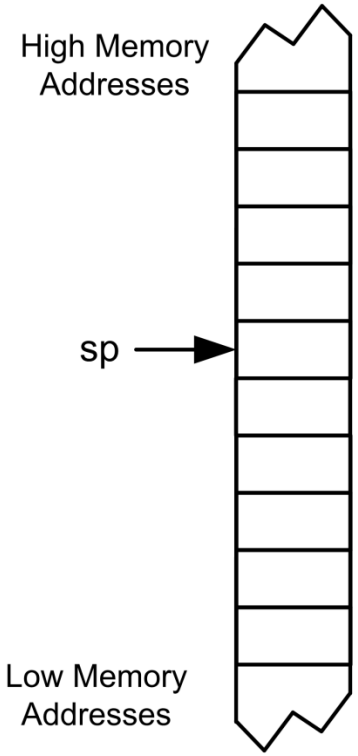
- Before using the stack, software has to define stack space and initialize the stack pointer (SP).
- The assembly file startup.s defines stack space and initialize SP.


File Edit View Project Flash Debug Peripherals Tools SVCS Window Help
main
Target 1
Project
Project: test
Target 1
Source Group 1
test.s
CMSIS
Device
GPIO_LPC17xx.c (GPIO)
RTE_Device.h (Startup)
startup_LPC17xx.s (Startu
system_LPC17xx.c (Startu
test.s
startup_LPC17xx.s
system_LPC17xx.c
22 ; *
23 ; ***
24
25 ; *------- <<< Use Configuration Wizard in Context Menu >>> --
26
27 ; <h> Stack Configuration
28 ; <o> Stack Size (in Bytes) <0x0-0xFFFFFFFF:8>
29 ; </h>
30
31 Stack_Size EQU 0x00000200
32
33 AREA STACK, NOINIT, READWRITE, ALIGN=3
34 Stack_Mem SPACE Stack_Size
35 __initial_sp
36
37
38 ; <h> Heap Configuration

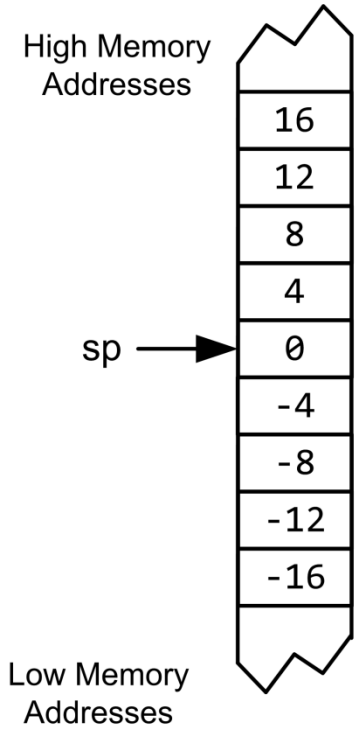

R12 0x00000000
R13 (SP) 0x10000200
R14 (LR) 0xFFFFFFFF
R15 (PC) 0x00000114
xPSR 0x01000000
Banked
11
12
13
14
15
16

Full Descending Stack

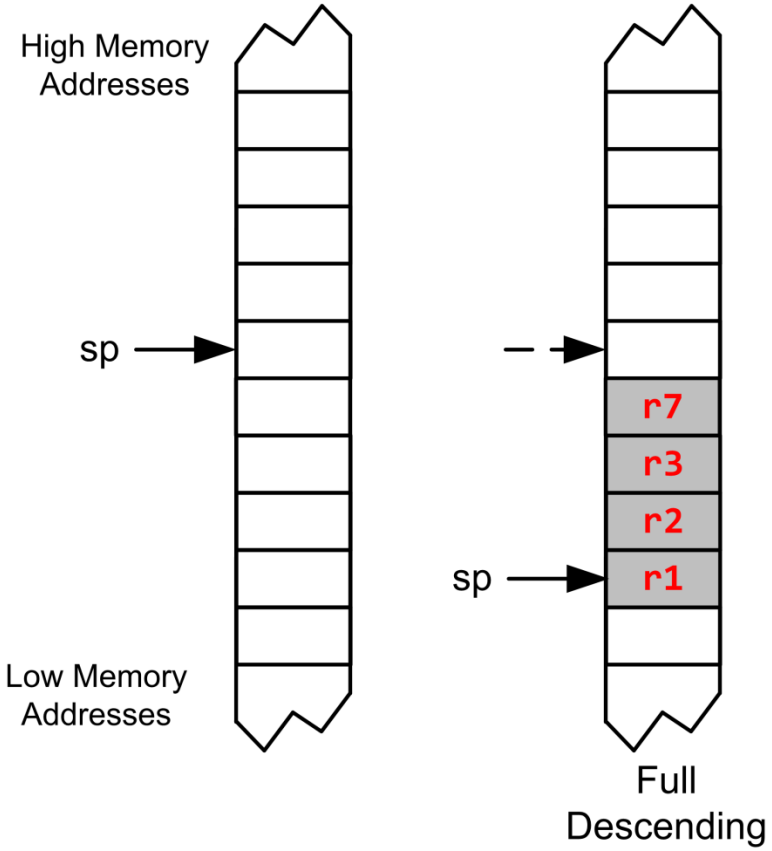
PUSH {r3, r1, r7, r2}



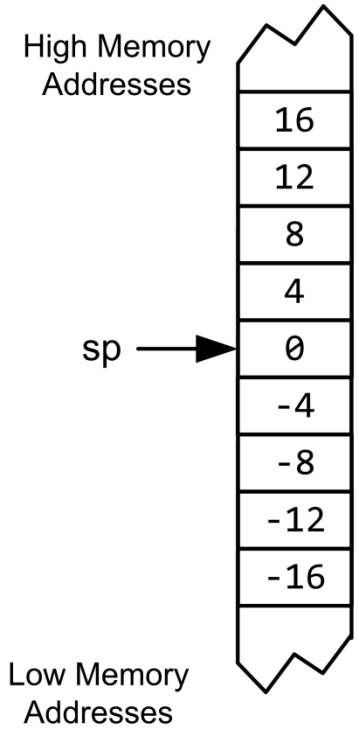
POP {r3, r1, r7, r2}



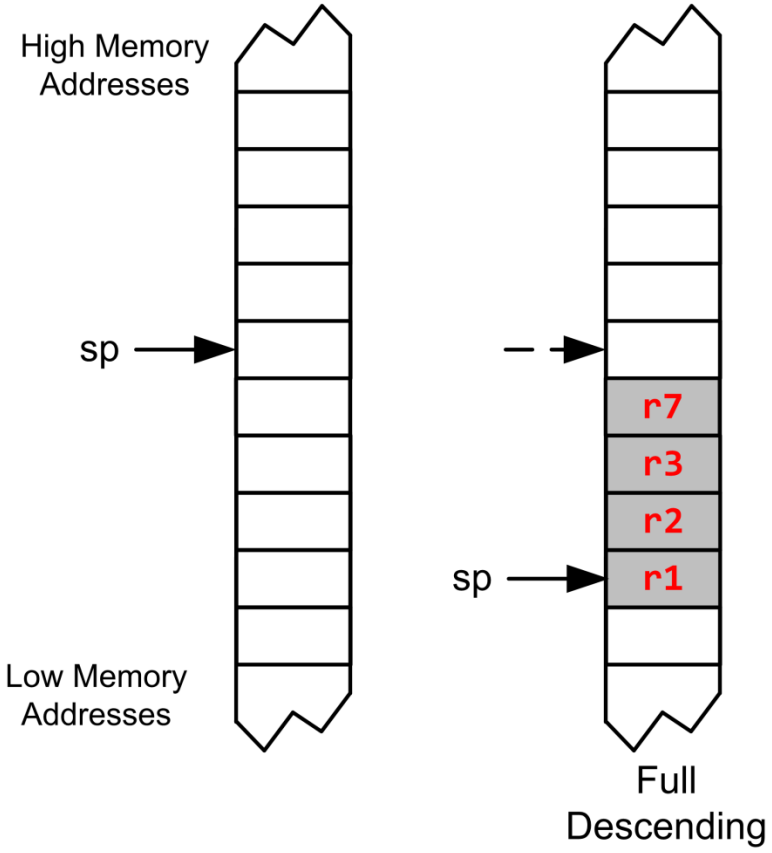
PUSH {r3, r1, r7, r2}



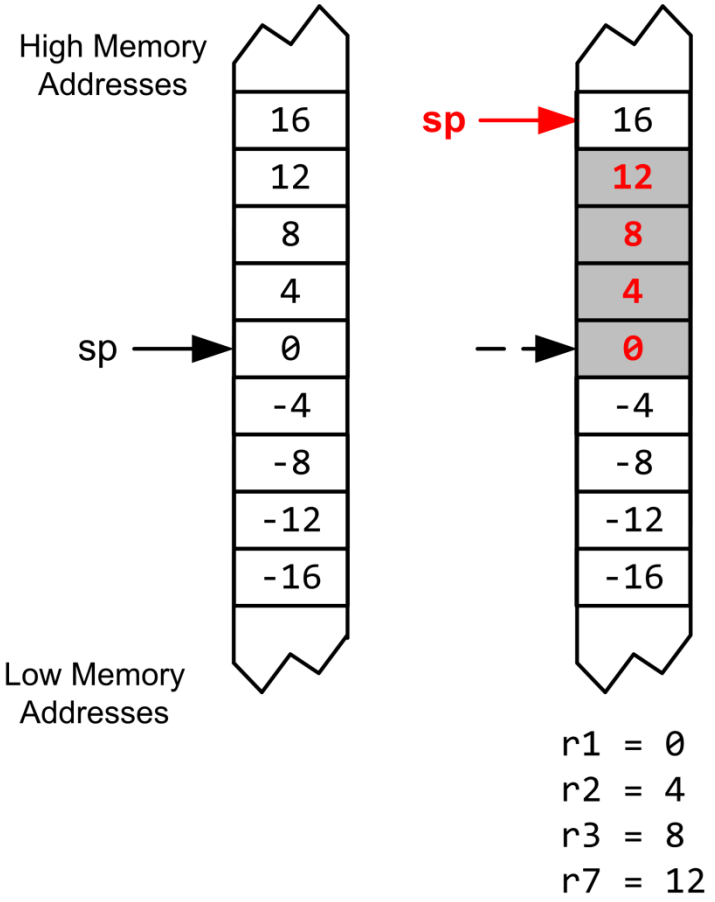
POP {r3, r1, r7, r2}



PUSH {r3, r1, r7, r2}

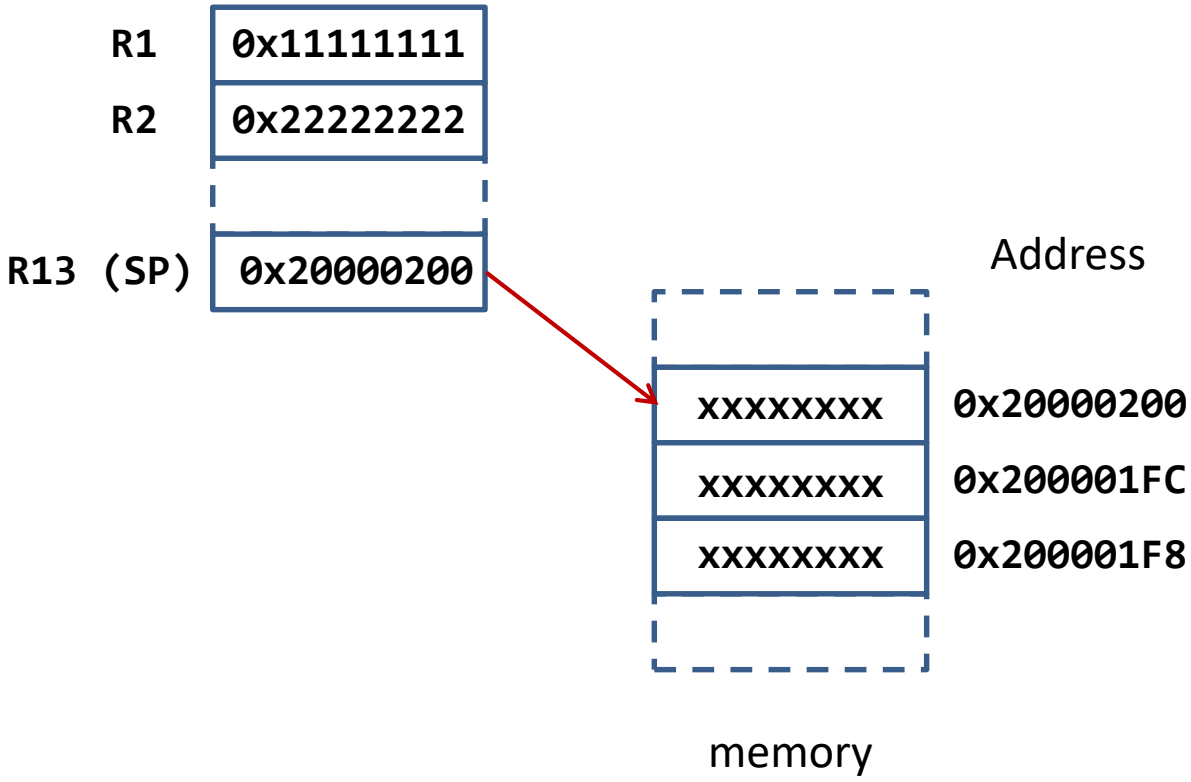


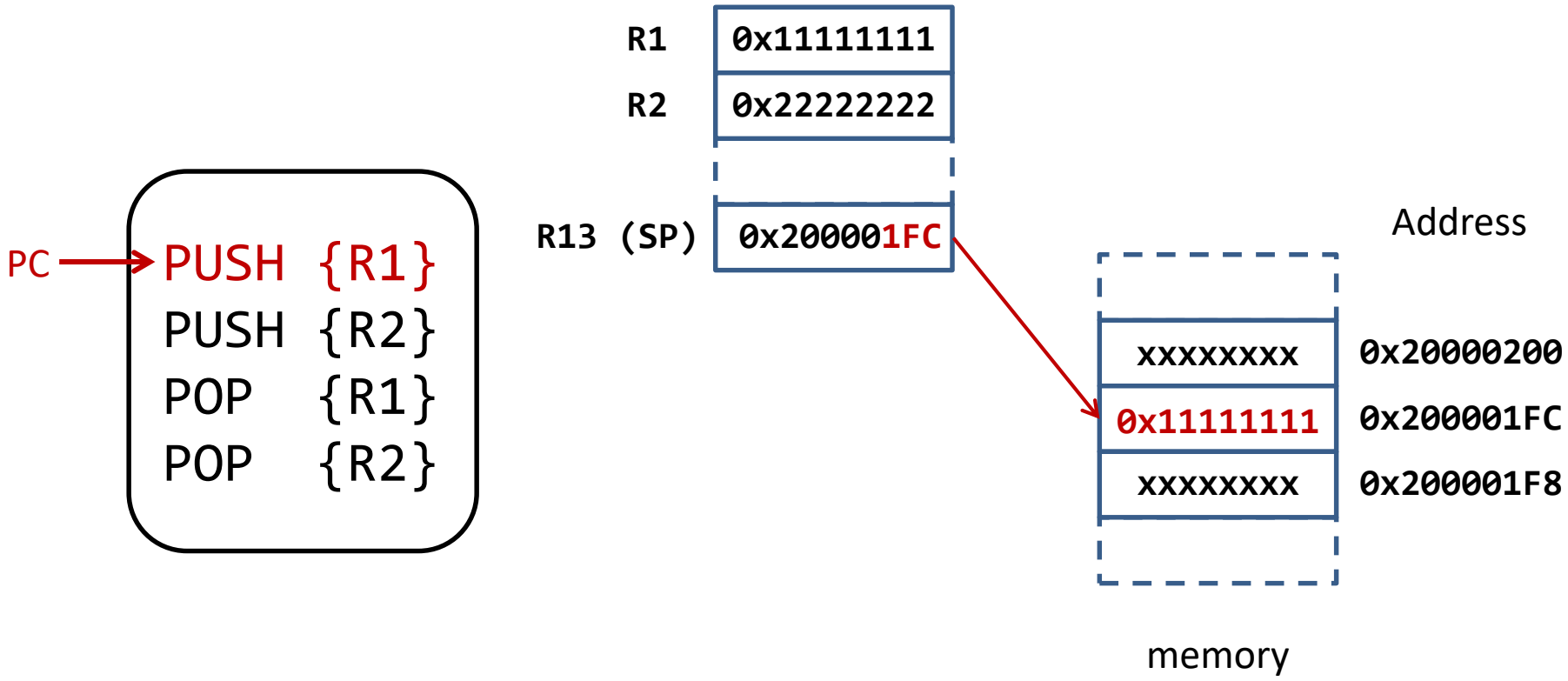
POP {r3, r1, r7, r2}

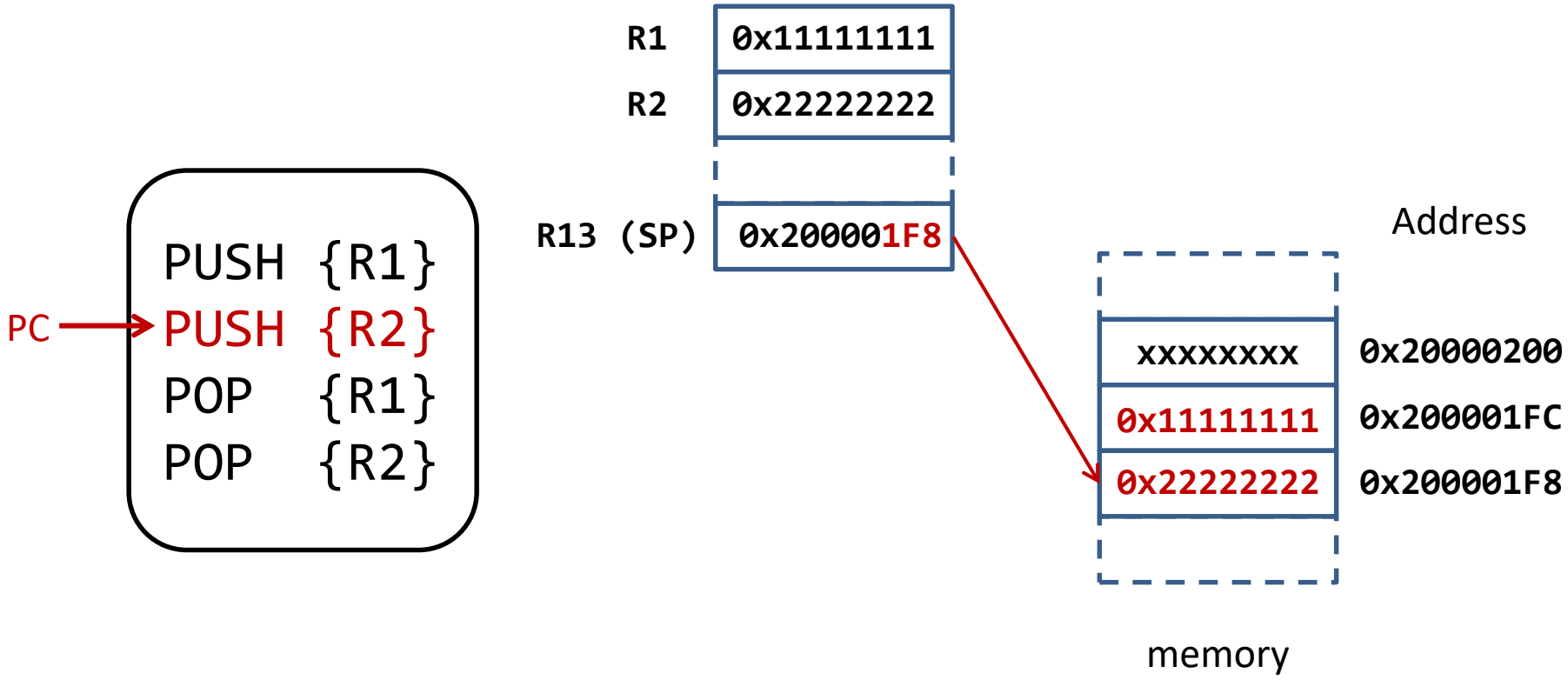


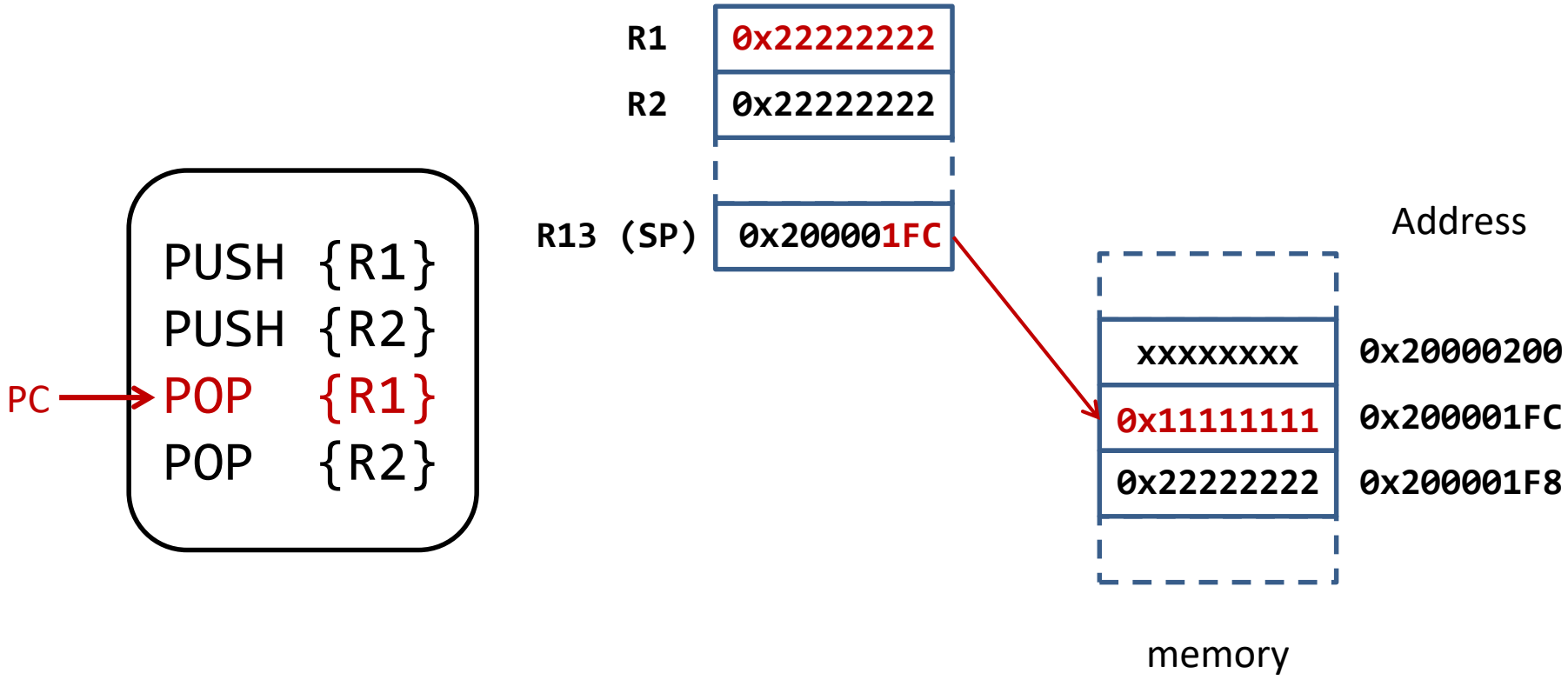
Example: swap R1 & R2

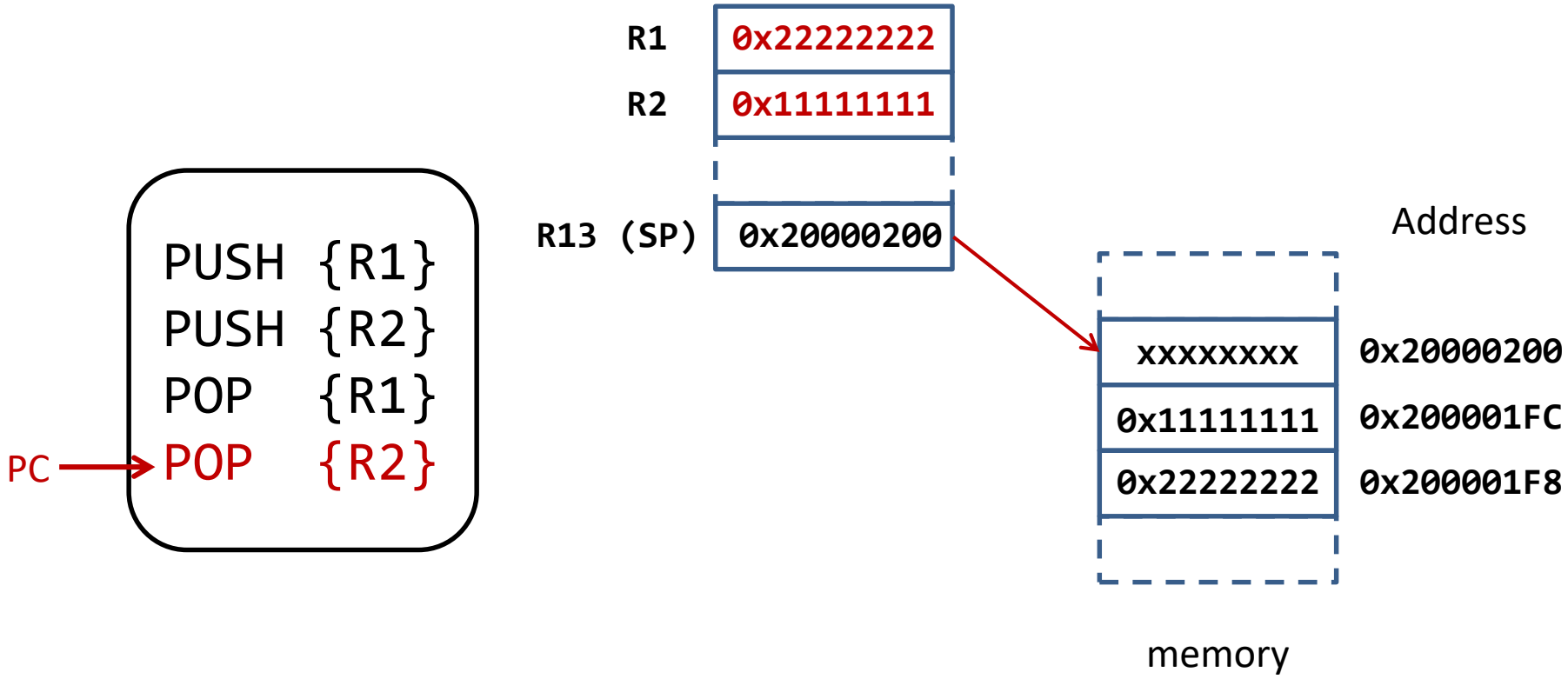
```
PUSH {R1}
PUSH {R2}
POP  {R1}
POP  {R2}
```




PC
PUSH {R1}
PUSH {R2}
POP {R1}
POP {R2}
R1
0x11111111
R2
0x22222222
R13 (SP)
0x20001FC
Address
xxxxxxxx
0x11111111
xxxxxxxx
0x20000200
0x200001FC
0x200001F8
memory


PUSH {R1}
PC
PUSH {R2}
POP {R1}
POP {R2}
R1
0x11111111
R2
0x22222222
R13 (SP)
0x20001F8
Address
xxxxxxxx
0x20000200
0x11111111
0x200001FC
0x22222222
0x200001F8
memory


PUSH {R1}
PUSH {R2}
PC
POP {R1}
POP {R2}
R1
0x22222222
R2
0x22222222
R13 (SP)
0x200001FC
Address
xxxxxxxx
0x11111111
0x22222222
0x20000200
0x200001FC
0x200001F8
memory


PUSH {R1}
PUSH {R2}
POP {R1}
POP {R2}
PC
R1
0x22222222
R2
0x11111111
R13 (SP)
0x20000200
Address
xxxxxxxx
0x11111111
0x22222222
0x20000200
0x200001FC
0x200001F8
memory

Are the values of R1 and R2 swapped?

```
PUSH {R1, R2}
POP  {R2, R1}
```

Answer: No.

But you can:

Recap: Call a Subroutine I (has a problem)

Caller Program	Subroutine/Callee

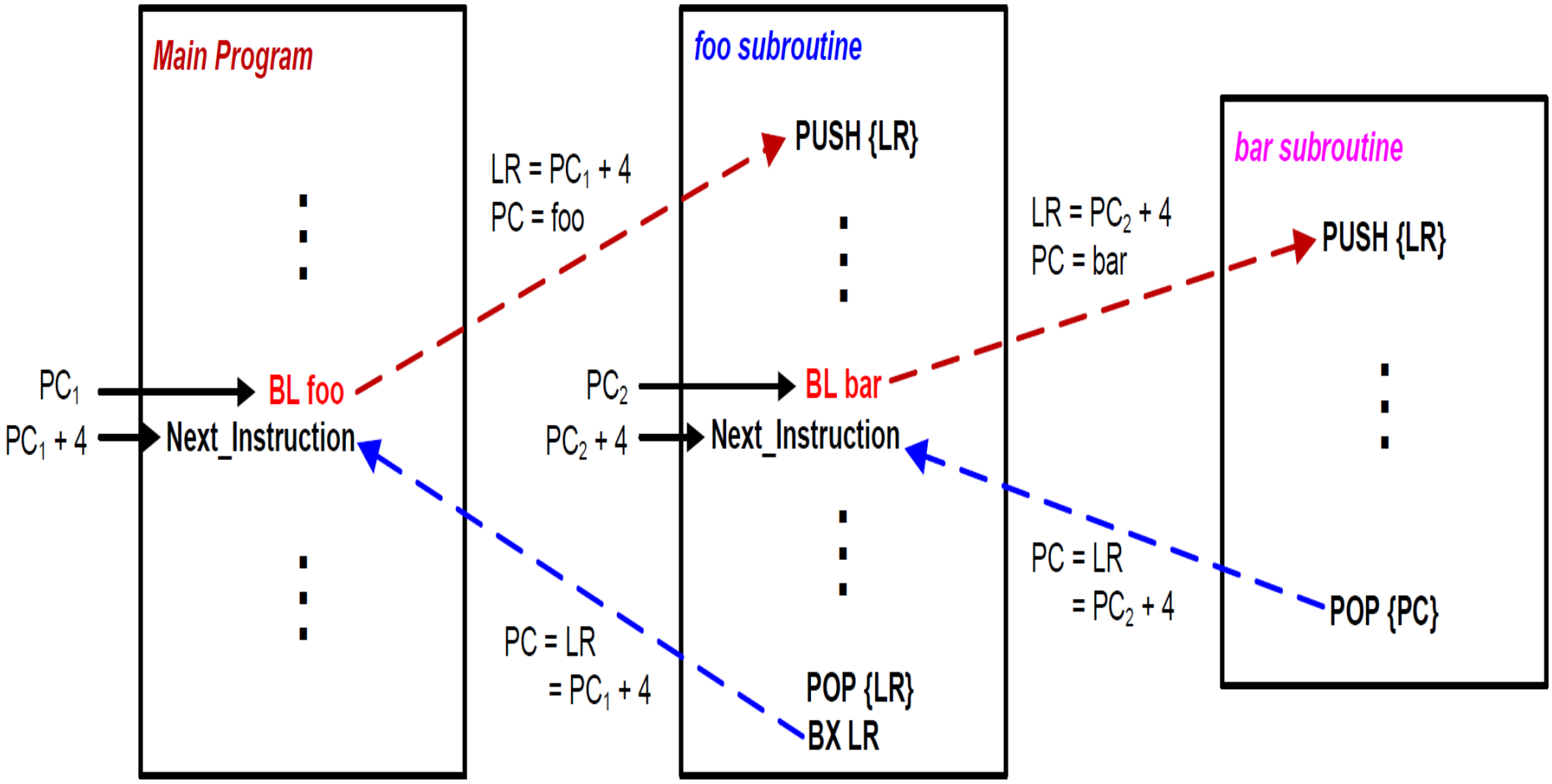
`MOV r4, #100`
`...`
`BL foo`
`...`
`ADD r4, r4, #1 ; r4 = 101, not 11`
	`foo PROC`
	`...`
	`MOV r4, #10 ; foo changes r4`
	`...`
	`BX LR`
	`ENDP`

Call a Subroutine II: Preserve Runtime Environment via Stack

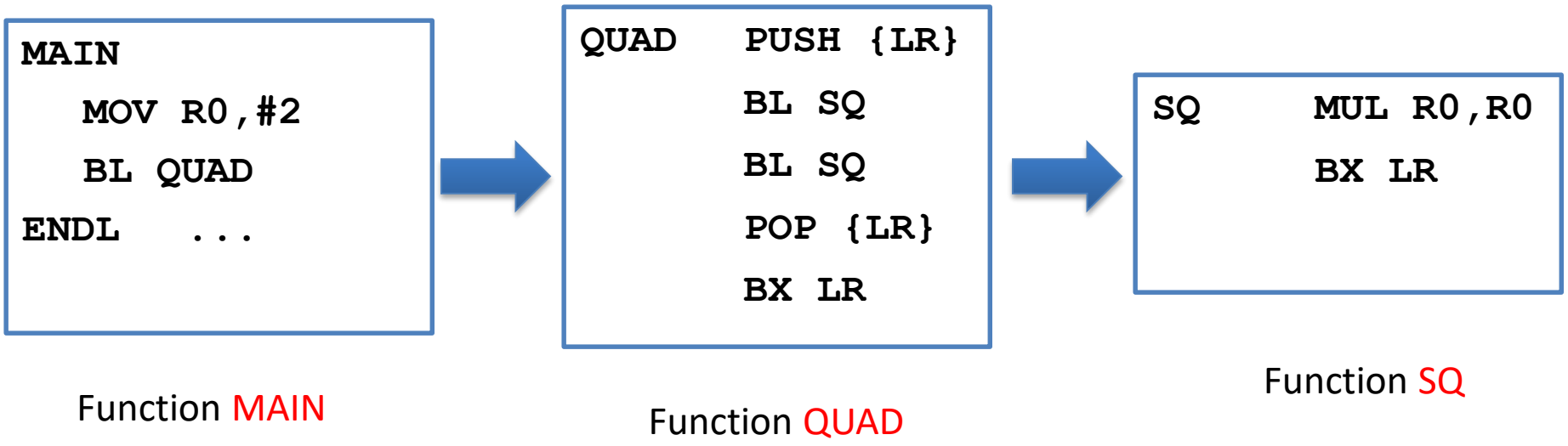
Caller Program	Subroutine/Callee

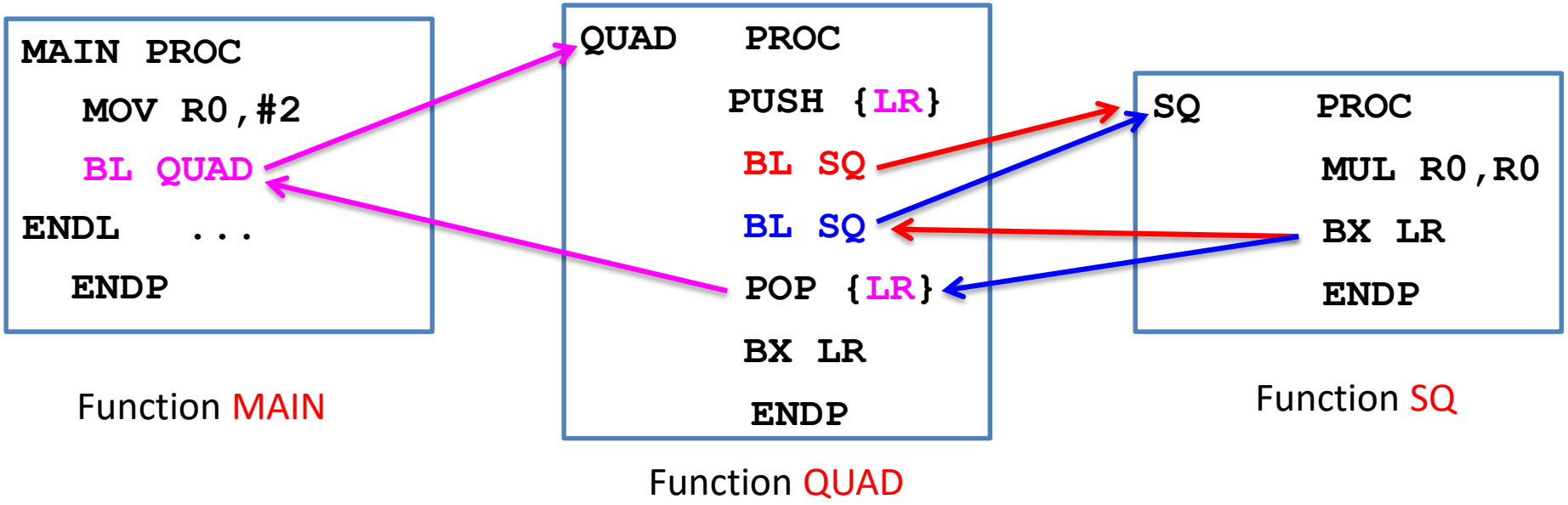
`MOV r4, #100`
`...`
`BL  foo`
`...`
`ADD r4, r4, #1  ; r4 = 101, not 11`
	`foo PROC`
	`  PUSH  {r4}  ; preserve r4`
	`  ...`
	`  MOV  r4, #10  ; foo changes r4`
	`  ...`
	`  POP  {r4}  ; Recover r4`
	`  BX  LR`
	`ENDP`

Stacks and Subroutines



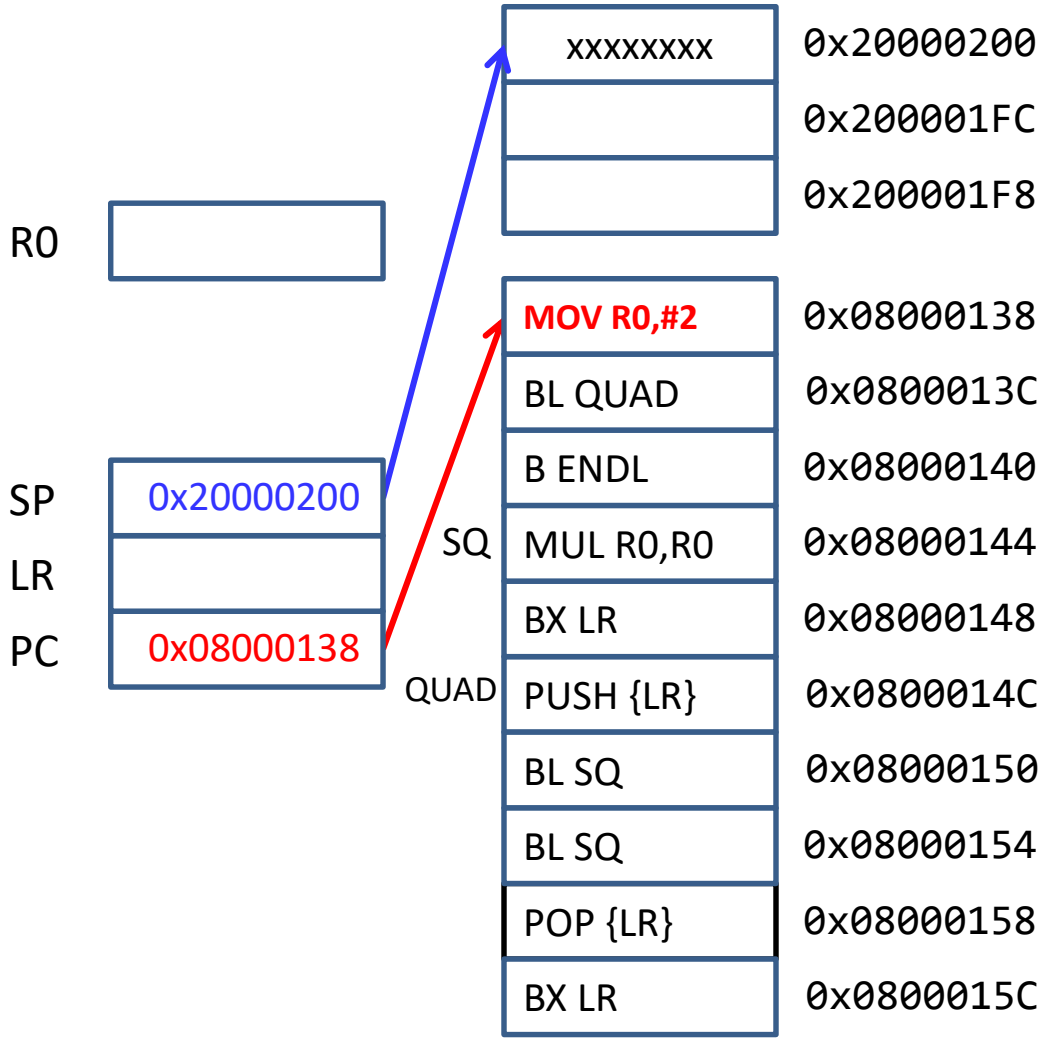
Subroutine Calling Another Subroutine



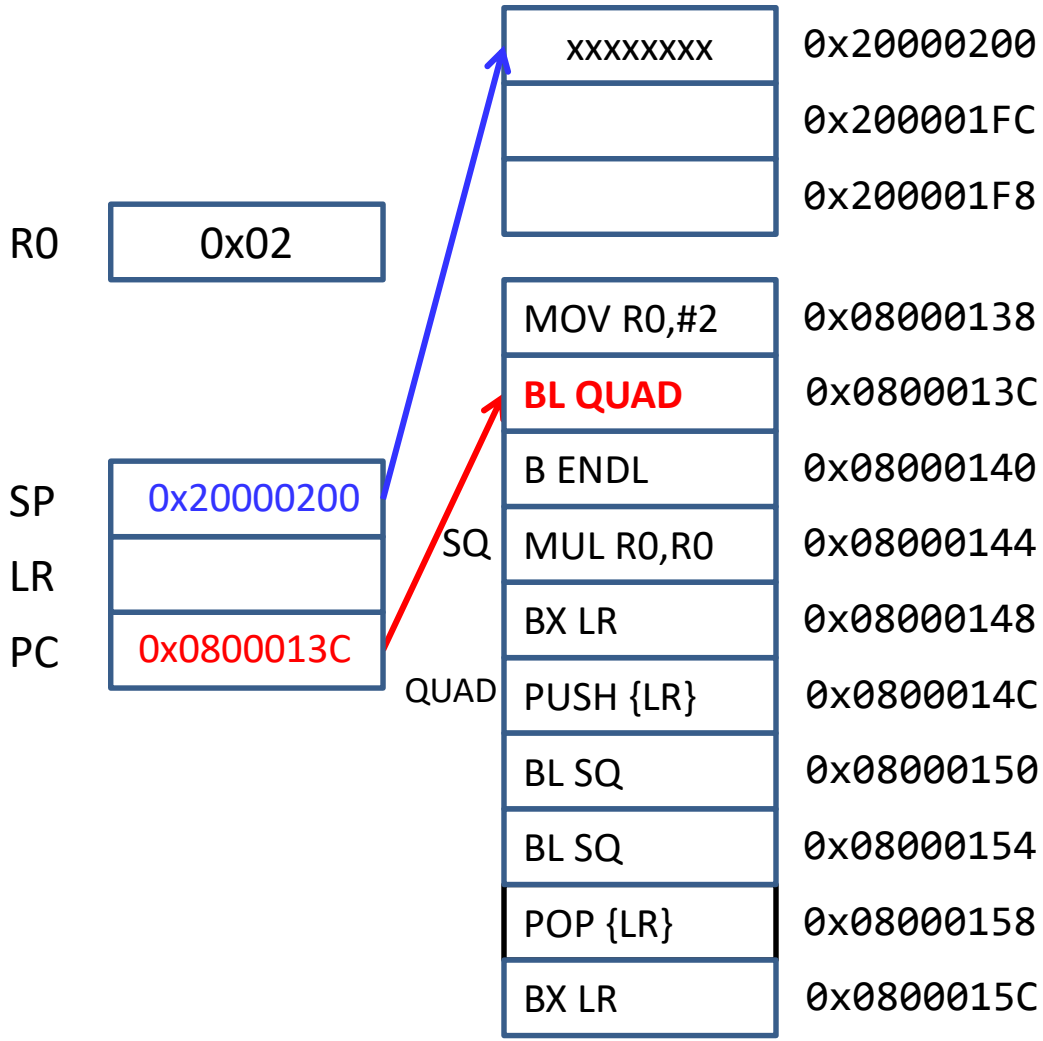

MAIN PROC
MOV R0,#2
BL QUAD
ENDL ...
ENDP
Function MAIN
QUAD PROC
PUSH {LR}
BL SQ
BL SQ
POP {LR}
BX LR
ENDP
Function QUAD
SQ PROC
MUL R0,R0
BX LR
ENDP
Function SQ

Example: R0 = R0^4

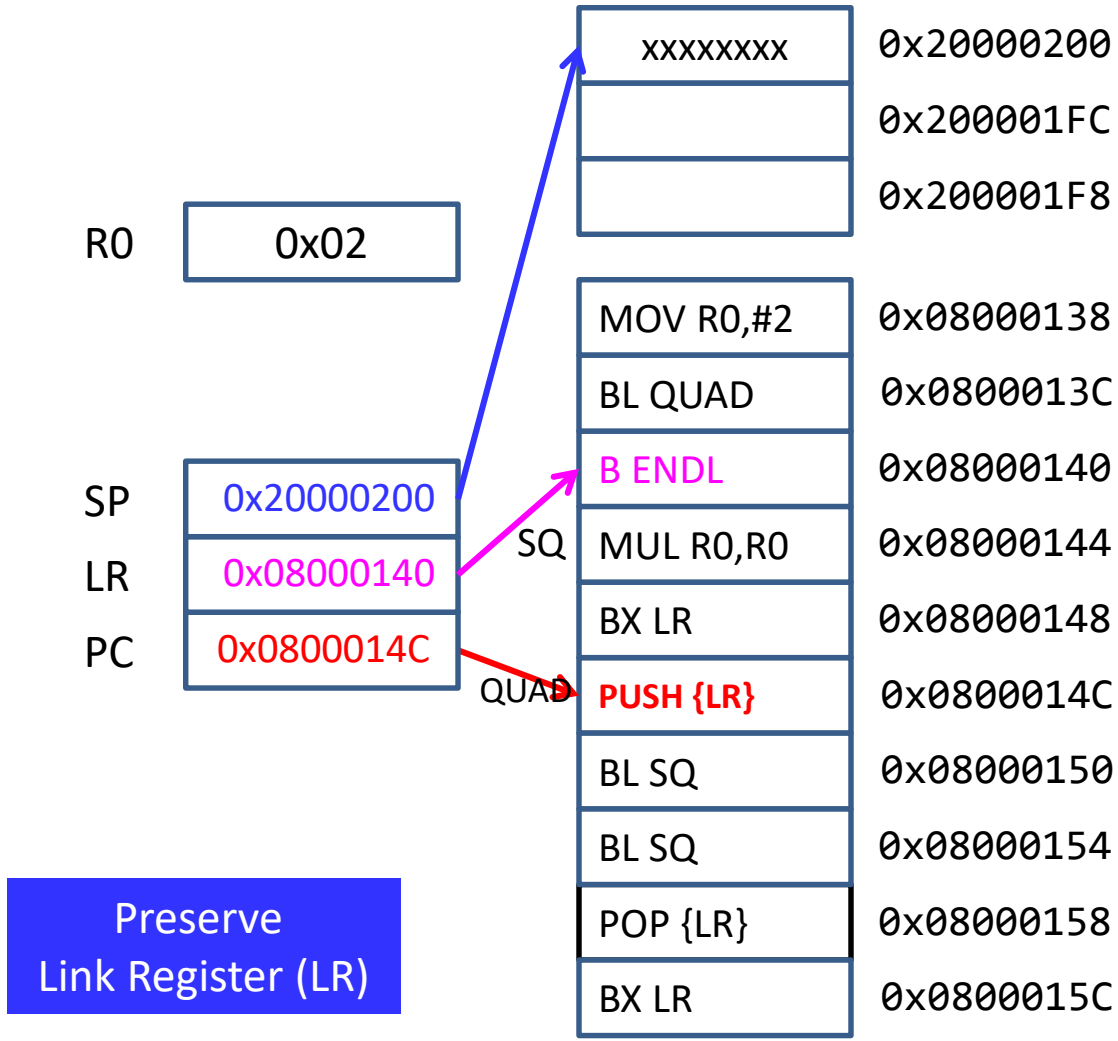
```
        MOV R0,#2
        BL QUAD
        B ENDL

SQ      MUL R0,R0
        BX LR

QUAD    PUSH {LR}
        BL SQ
        BL SQ
        POP {LR}
        BX LR
ENDL    ...
```



```
        MOV R0,#2
        BL QUAD
        B ENDL

SQ      MUL R0,R0
        BX LR

QUAD    PUSH {LR}
        BL SQ
        BL SQ
        POP {LR}
        BX LR
ENDL    ...
```

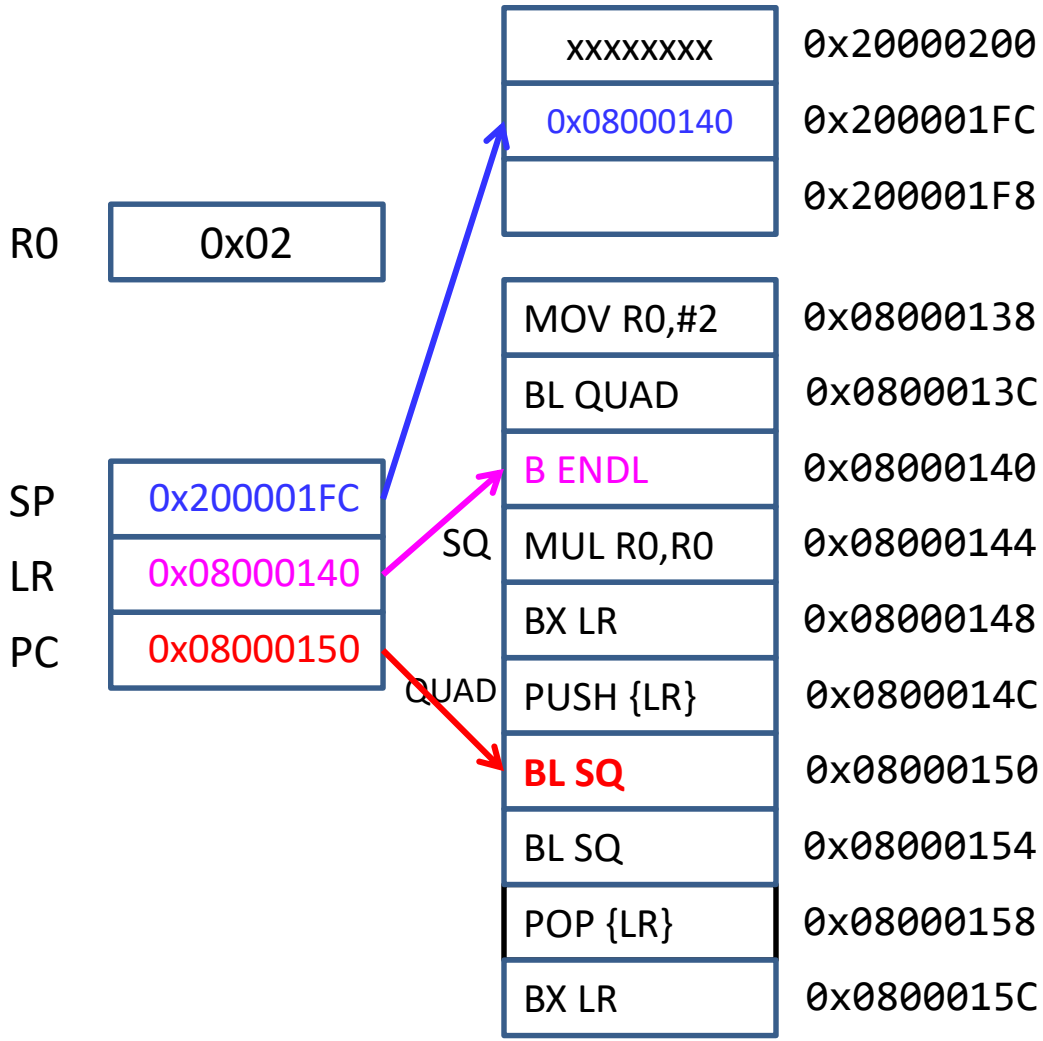


```
        MOV R0,#2
        BL QUAD
        B ENDL

SQ      MUL R0,R0
        BX LR

QUAD    PUSH {LR}
        BL SQ
        BL SQ
        POP {LR}
        BX LR
ENDL    ...
```

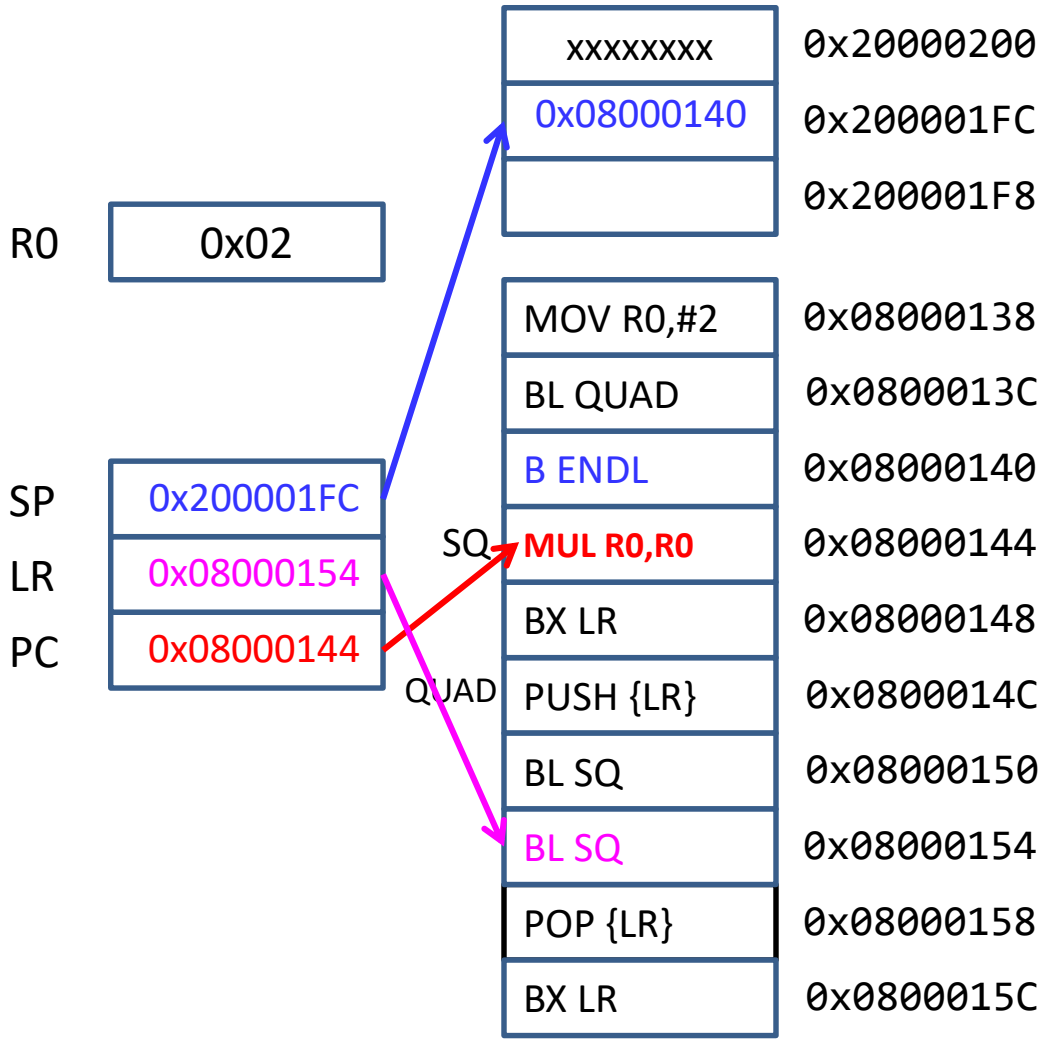


Preserve
Link Register (LR)

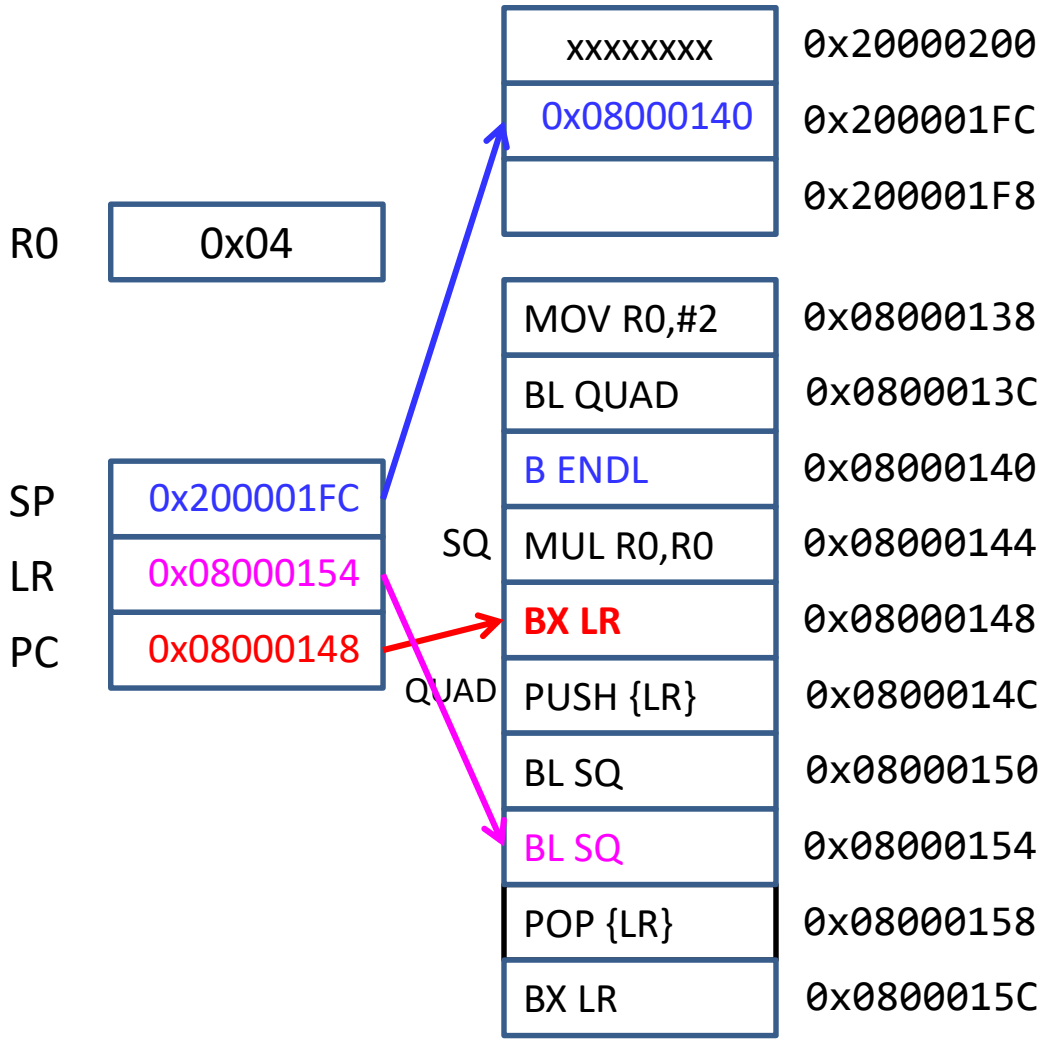
```
        MOV R0,#2
        BL QUAD
        B ENDL

SQ      MUL R0,R0
        BX LR

QUAD    PUSH {LR}
        BL SQ
        BL SQ
        POP {LR}
        BX LR
ENDL    ...
```



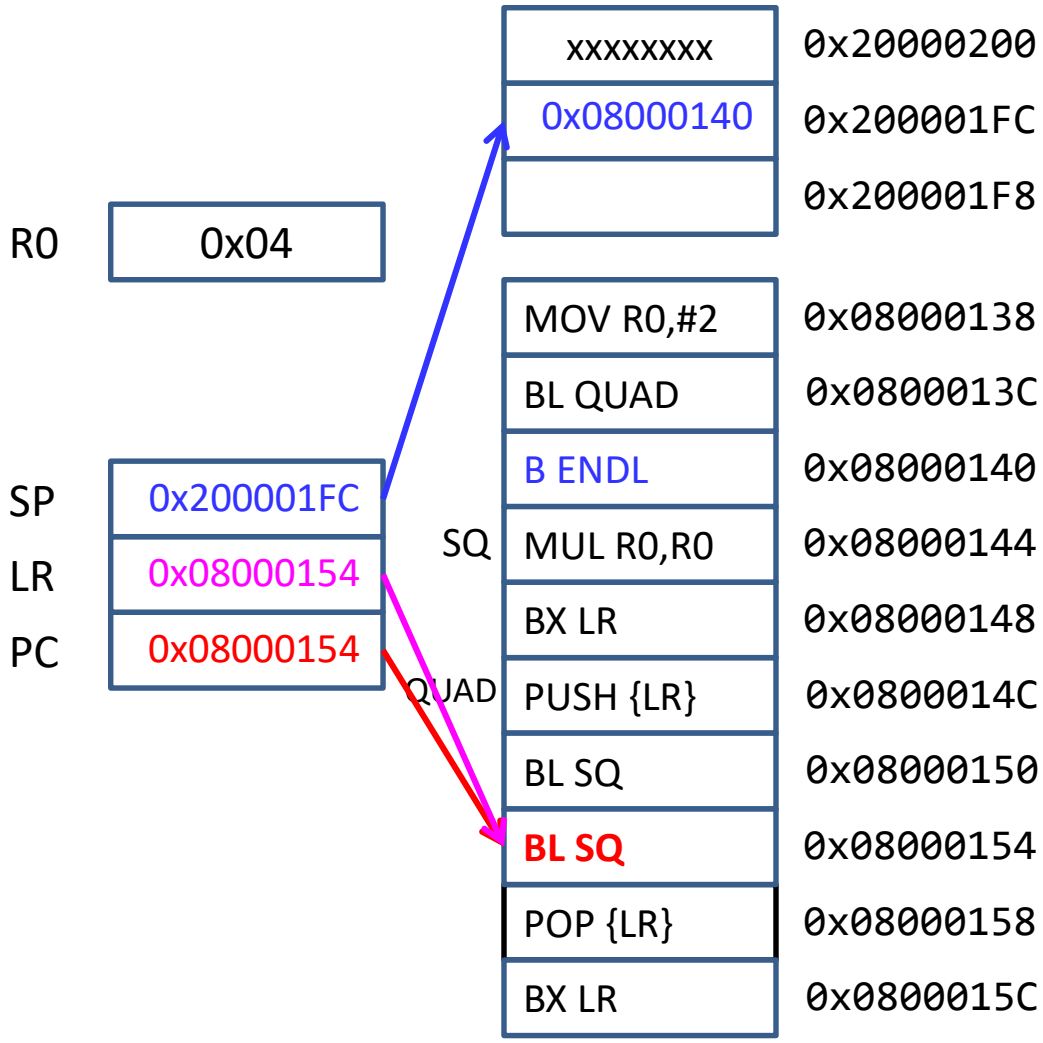
```
        MOV R0,#2
        BL QUAD
        B ENDL

SQ      MUL R0,R0
        BX LR

QUAD    PUSH {LR}
        BL SQ
        BL SQ
        POP {LR}
        BX LR
ENDL    ...
```



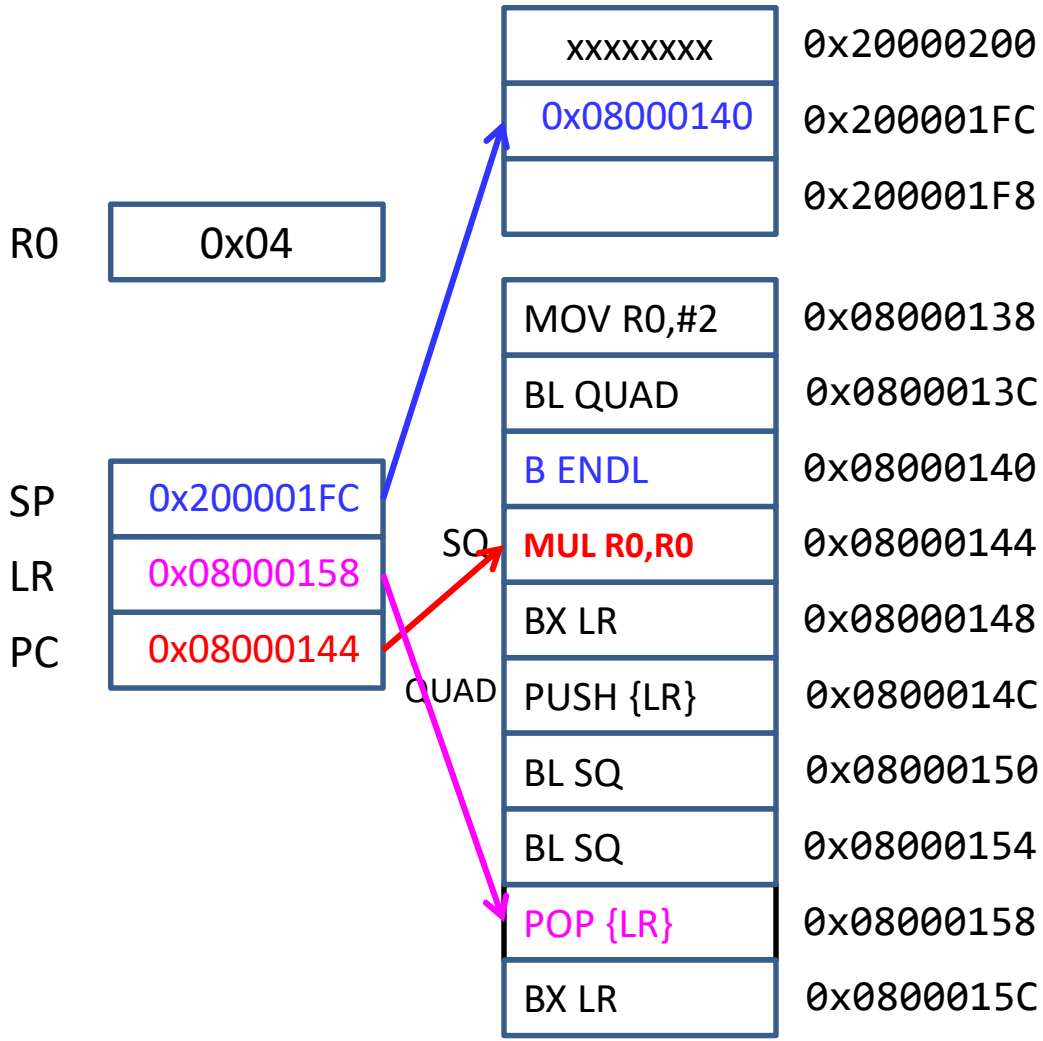
```
        MOV R0,#2
        BL QUAD
        B ENDL

SQ      MUL R0,R0
        BX LR

QUAD    PUSH {LR}
        BL SQ
        BL SQ
        POP {LR}
        BX LR
ENDL    ...
```



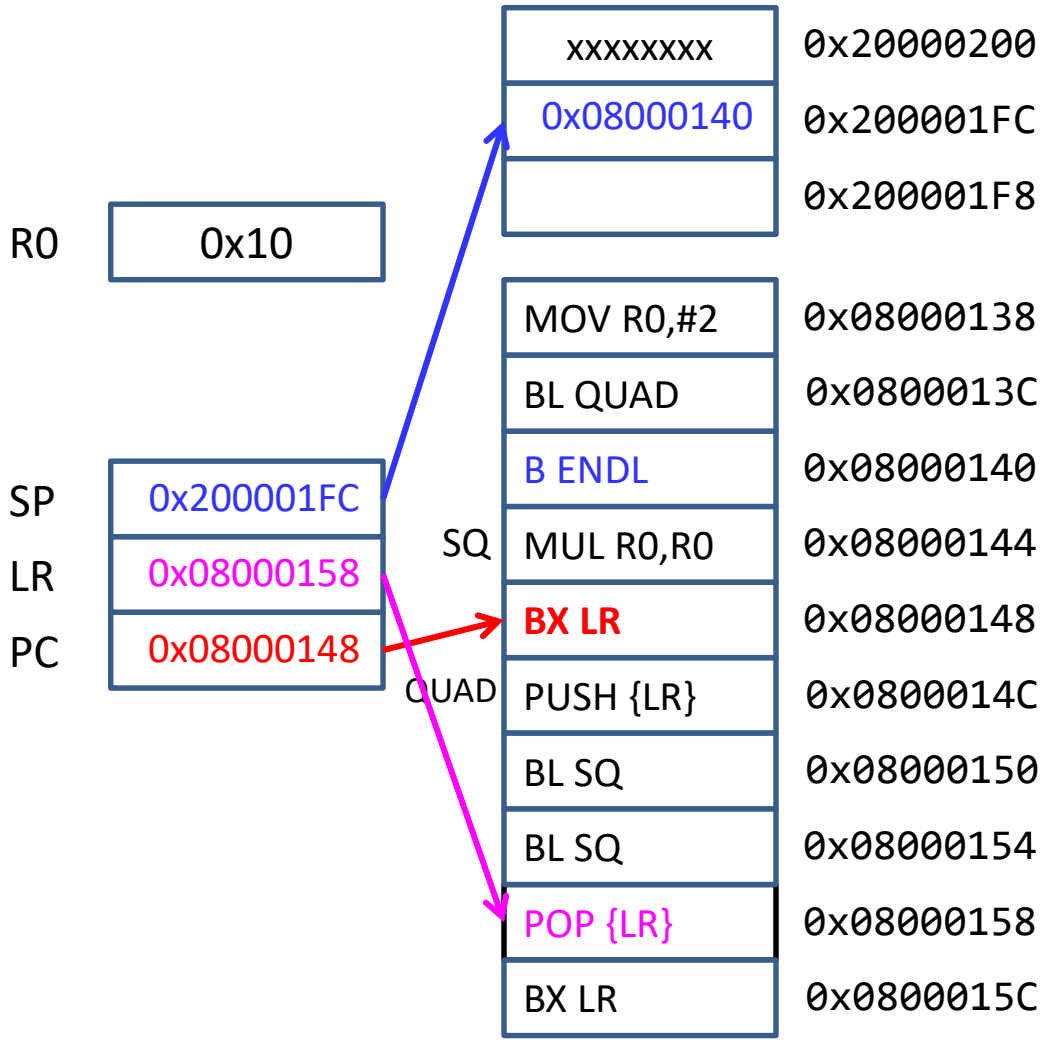
```
        MOV R0,#2
        BL QUAD
        B ENDL

SQ      MUL R0,R0
        BX LR

QUAD    PUSH {LR}
        BL SQ
        BL SQ
        POP {LR}
        BX LR
ENDL    ...
```



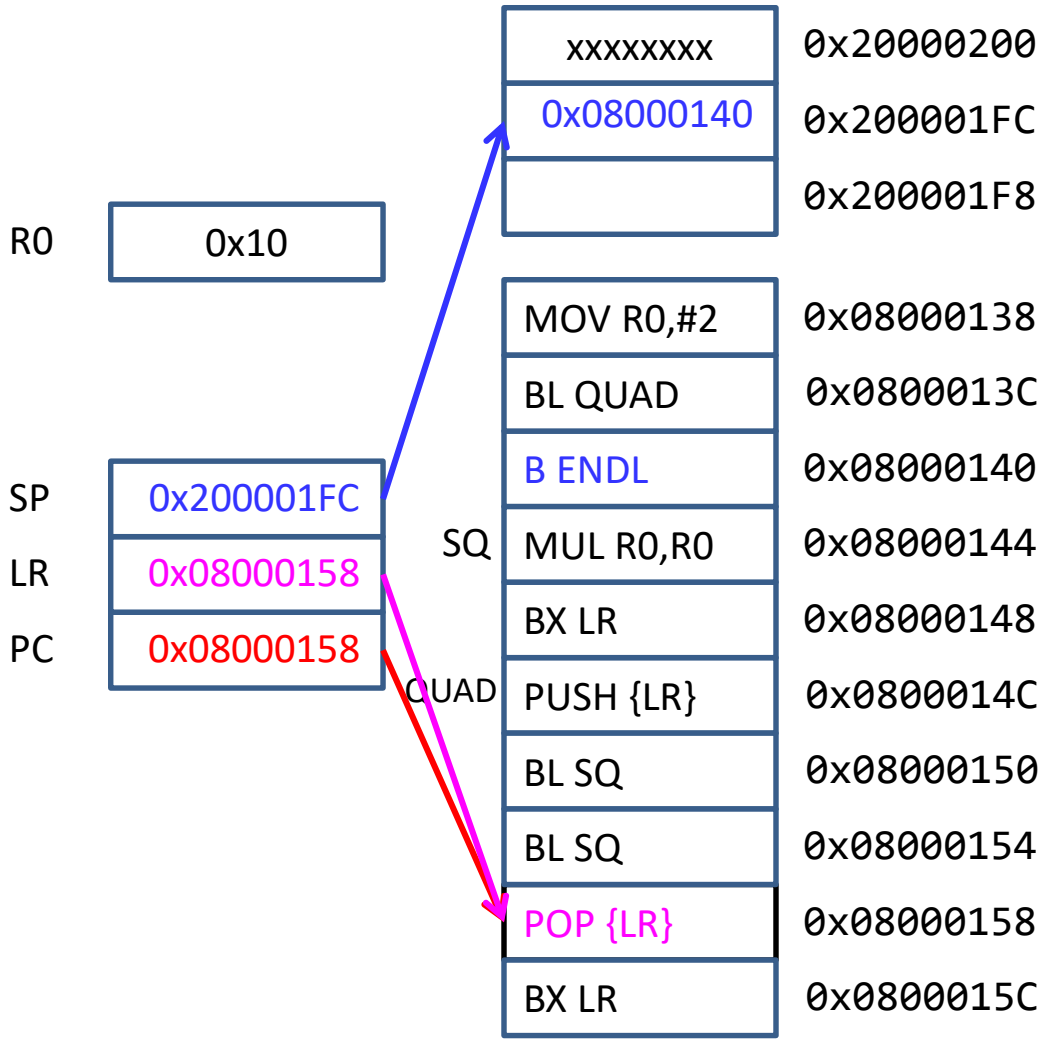
```
        MOV R0,#2
        BL QUAD
        B ENDL

SQ      MUL R0,R0
        BX LR

QUAD    PUSH {LR}
        BL SQ
        BL SQ
        POP {LR}
        BX LR
ENDL    ...
```



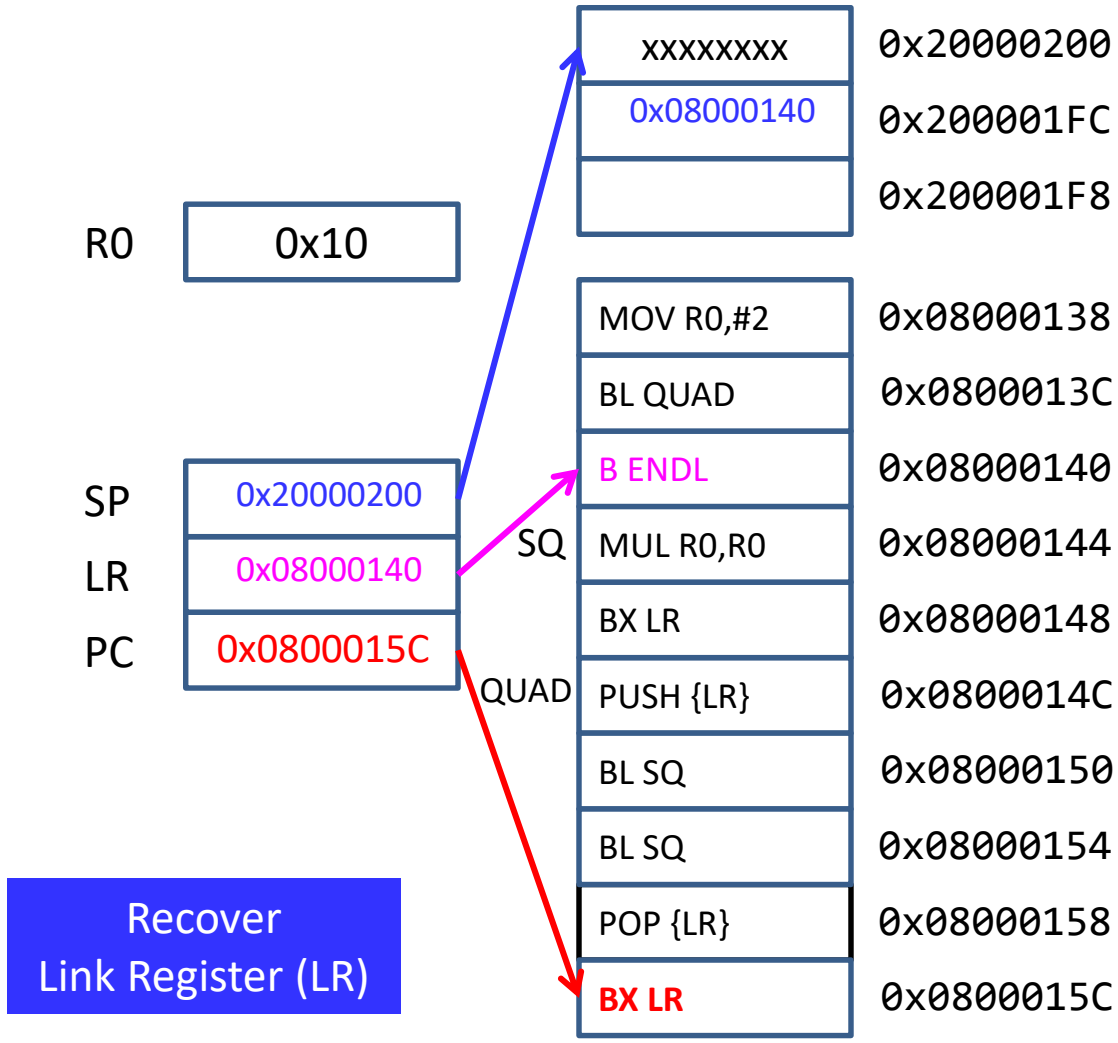
```
        MOV R0,#2
        BL QUAD
        B ENDL

SQ      MUL R0,R0
        BX LR

QUAD    PUSH {LR}
        BL SQ
        BL SQ
        POP {LR}
        BX LR
ENDL    ...
```



```
        MOV R0,#2
        BL QUAD
        B ENDL

SQ      MUL R0,R0
        BX LR

QUAD    PUSH {LR}
        BL SQ
        BL SQ
        POP {LR}
        BX LR
ENDL    ...
```



```
        MOV R0,#2
        BL QUAD
        B ENDL

SQ      MUL R0,R0
        BX LR

QUAD    PUSH {LR}
        BL SQ
        BL SQ
        POP {LR}
        BX LR
ENDL    ...
```



Recover
Link Register (LR)

```
        MOV R0,#2
        BL QUAD
        B ENDL

SQ      MUL R0,R0
        BX LR

QUAD    PUSH {LR}
        BL SQ
        BL SQ
        POP {LR}
        BX LR
ENDL    ...
```