Embedded Systems with ARM Cortex-M Microcontrollers in Assembly Language and C

Chapter 10
Preserve Environment via Stack

Z. Gu

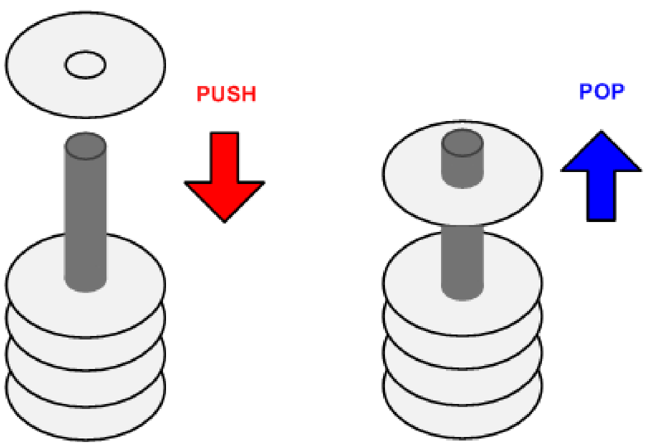
Fall 2025

Overview

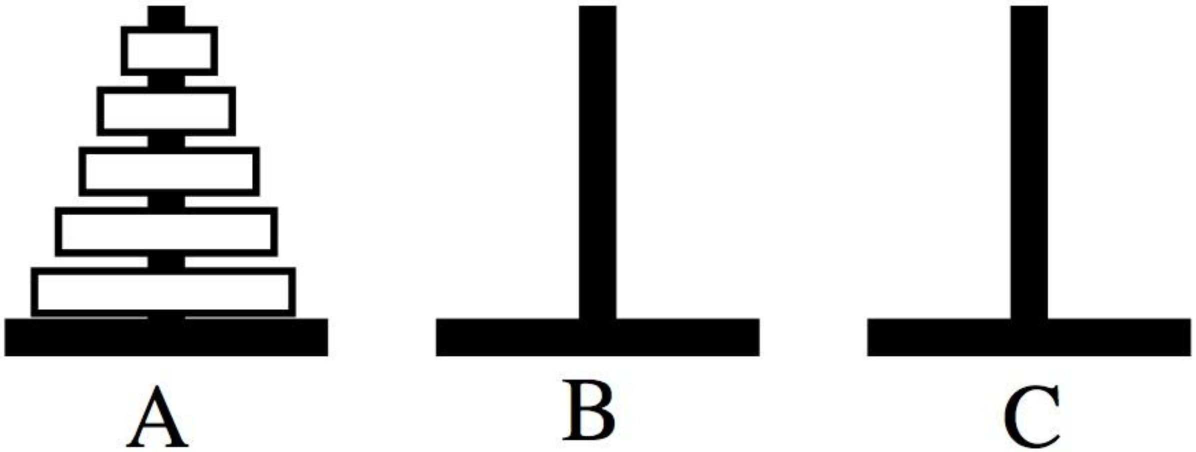
- How to call a subroutine?
- How to return the control back to the caller?
- How to pass arguments into a subroutine?
- How to return a value in a subroutine?
- How to preserve the running environment for the caller?

Stack

- A **Last-In-First-Out** memory model
- 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)



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.**

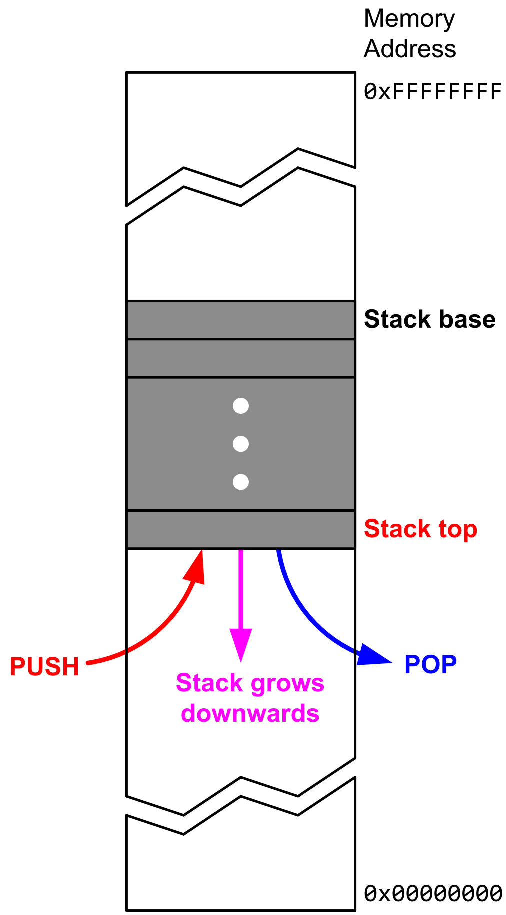
Tower of Hanoi

STACK: Last In First Out

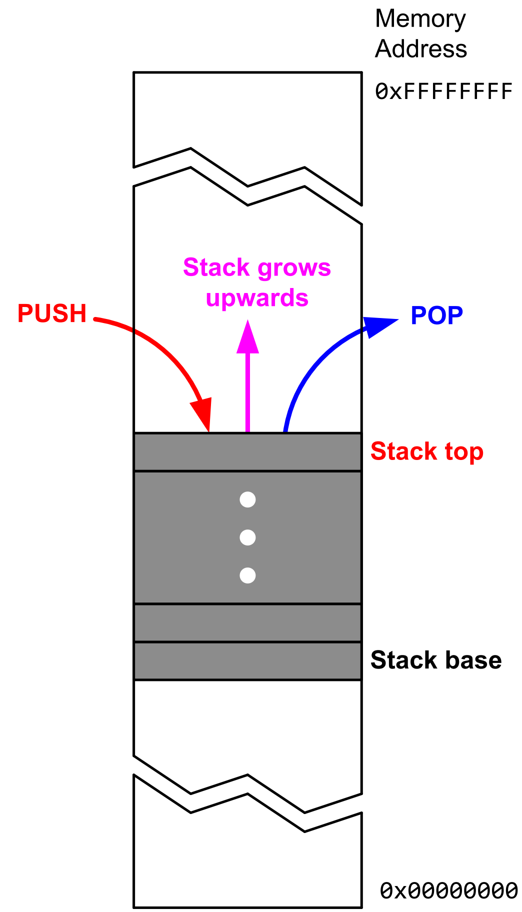
Stack 1 **Stack 2** **Stack 3**

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

Stack Growth Convention: Ascending *vs* Descending

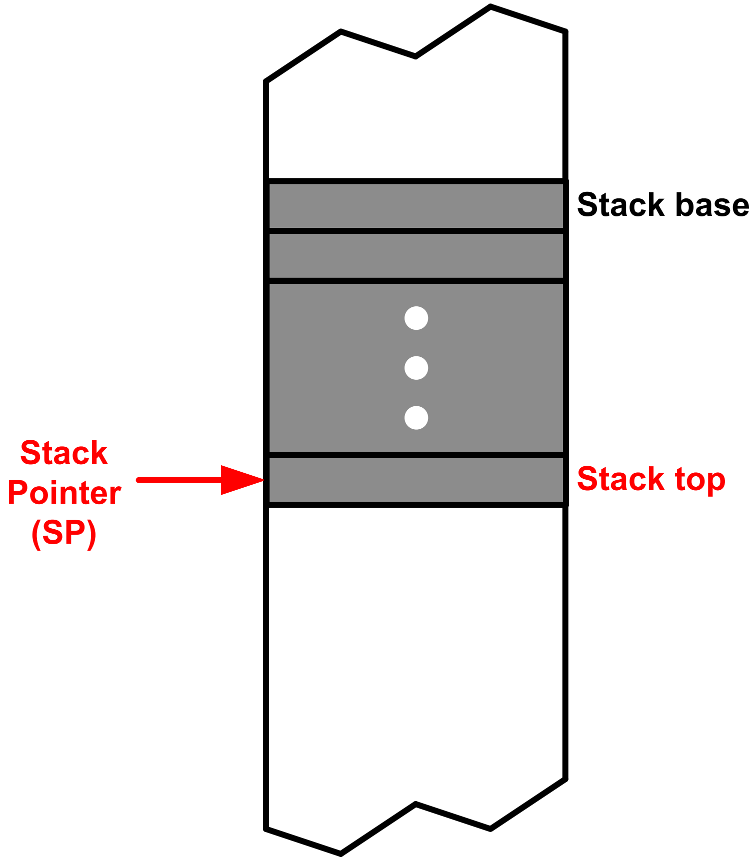


Descending stack: Stack grows towards low memory address

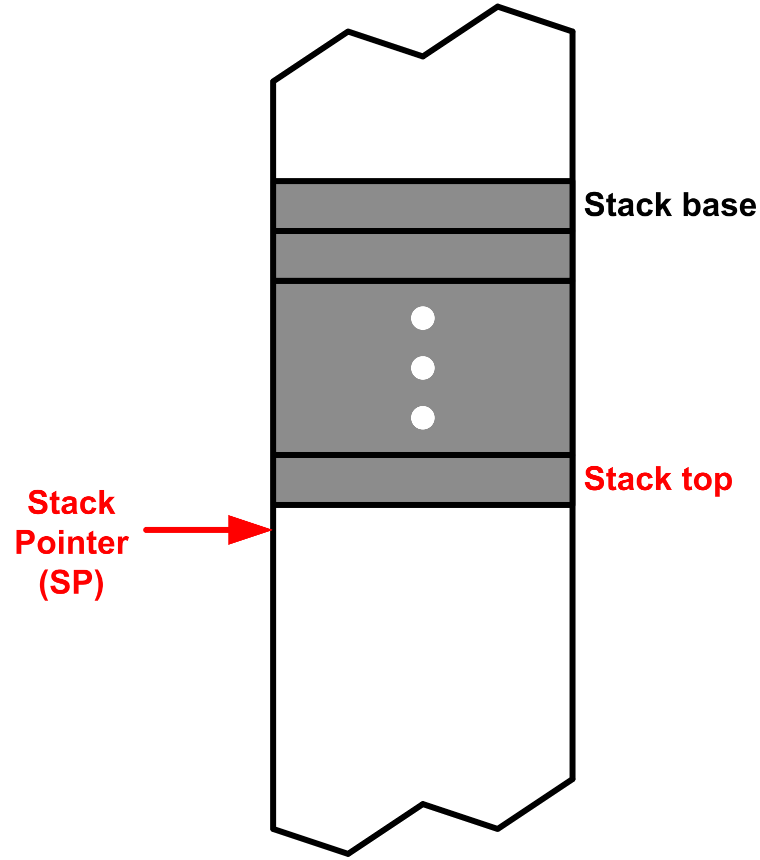


Ascending stack: Stack grows towards high memory address

Stack Growth Convention: Full *vs* Empty



Full stack: SP points to the last item pushed onto the stack



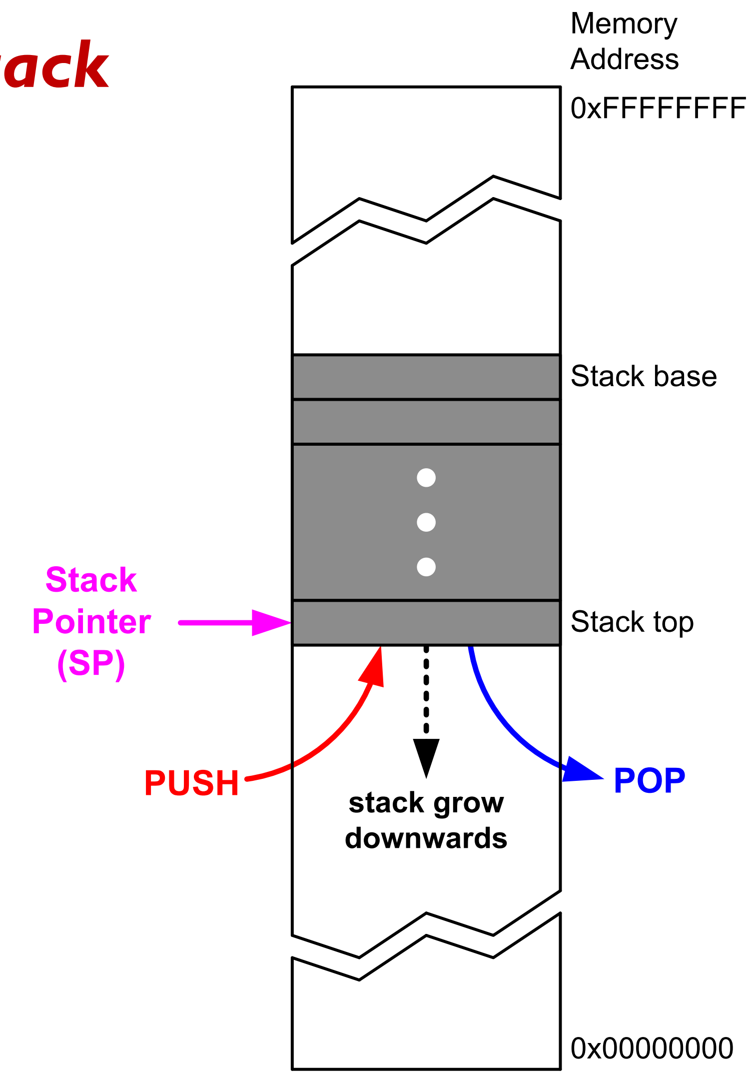
Empty stack: SP points to the next free space on the stack

Cortex-M Stack

- Cortex-M uses ***full descending stack***
- Example:

    PUSH/POP {r0,r6,r3}

- Stack pointer (SP, aka R13)
    - decremented on **PUSH**
        - **SP = SP – 4 * # of registers**
    - incremented on **POP**
        - **SP = SP + 4 * # of registers**
    - SP starts at **0x20000200** for STM32-Discovery by default (can be changed in startup.s)



Review

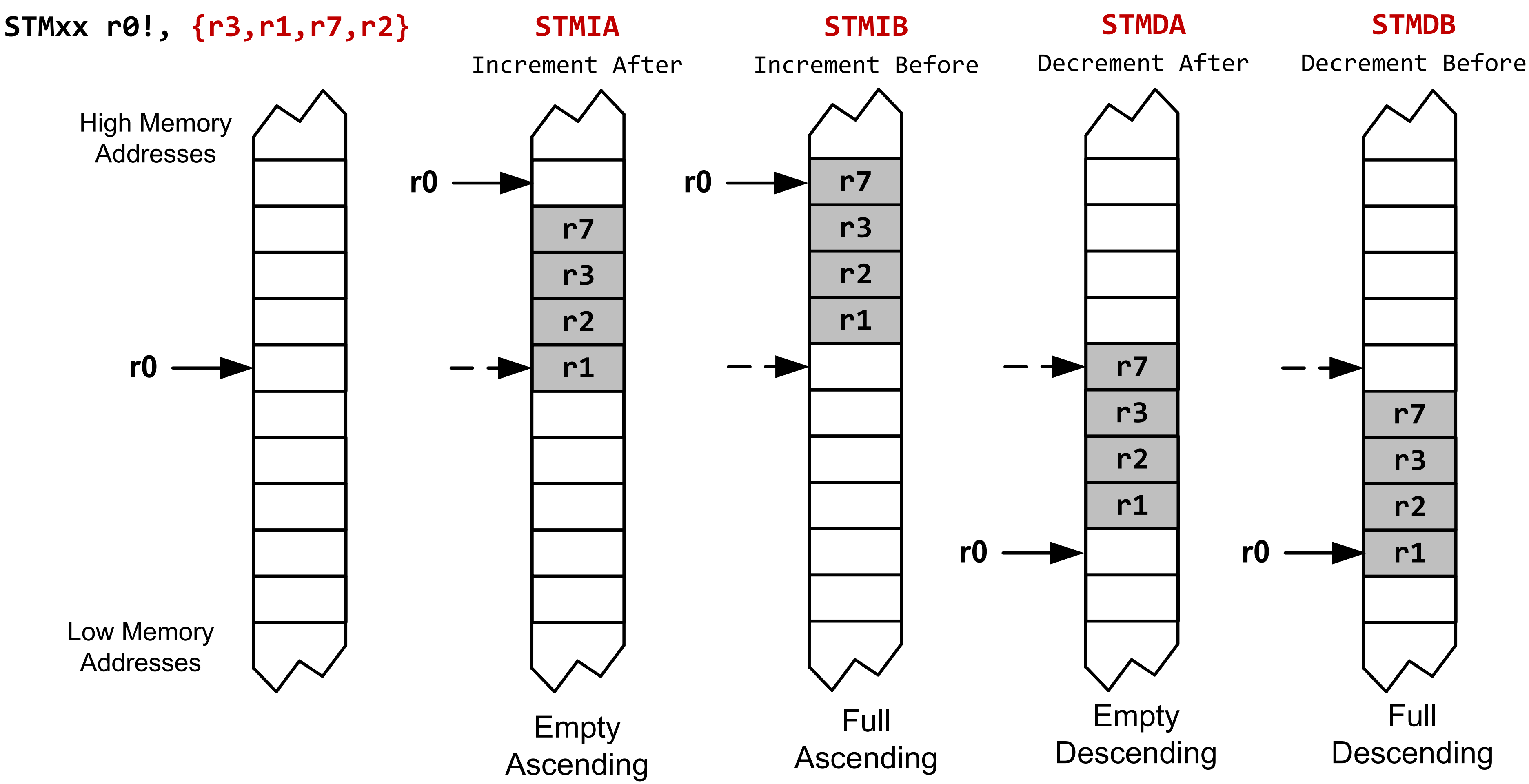
Load/Store Multiple Registers

- The following are synonyms.
  - **STM** = **STMIA** (Increment After) = **STMEA** (Empty Ascending)
  - **LDM** = **LDMIA** (Increment After) = **LDMFD** (Full Descending)

- The order in which registers are listed does not matter
  - For STM/LDM, the lowest-numbered register is stored/loaded at the lowest memory address.

Review

Store Multiple Registers

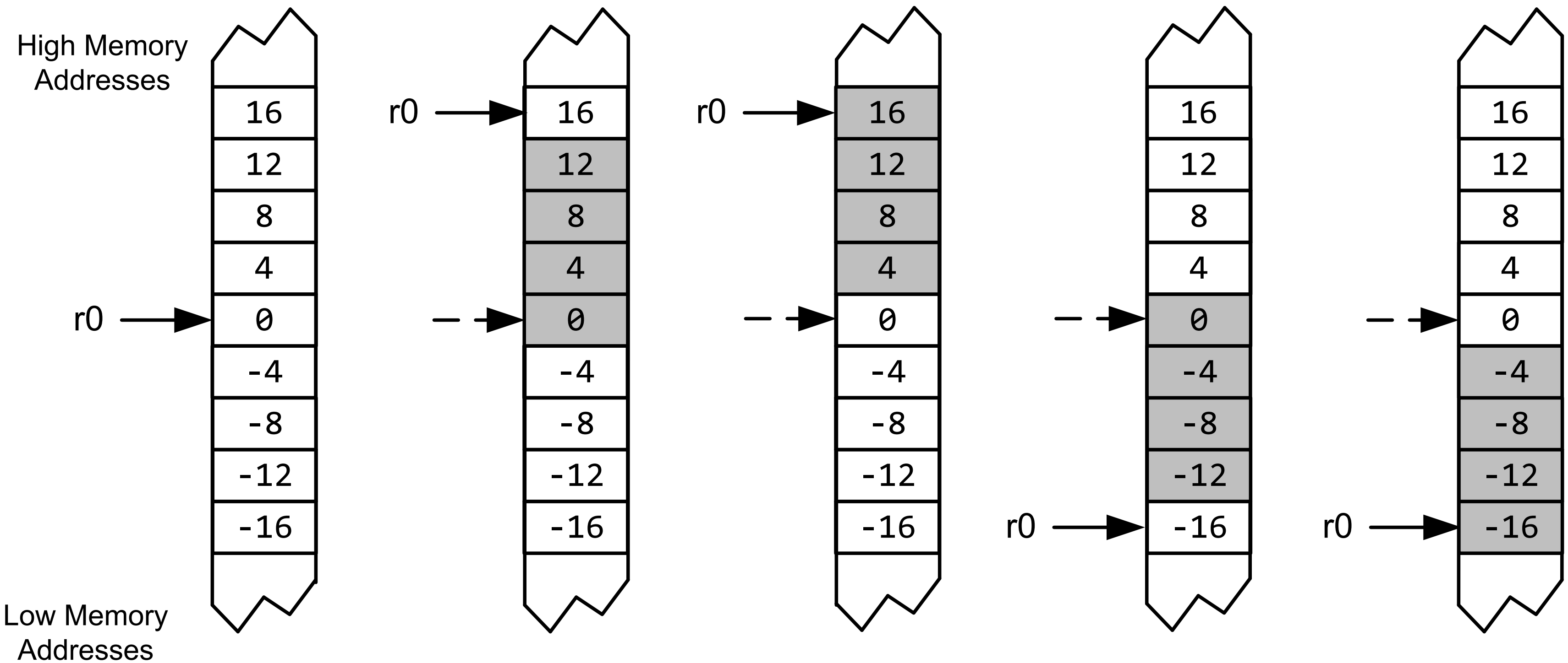


Review

Load Multiple Registers

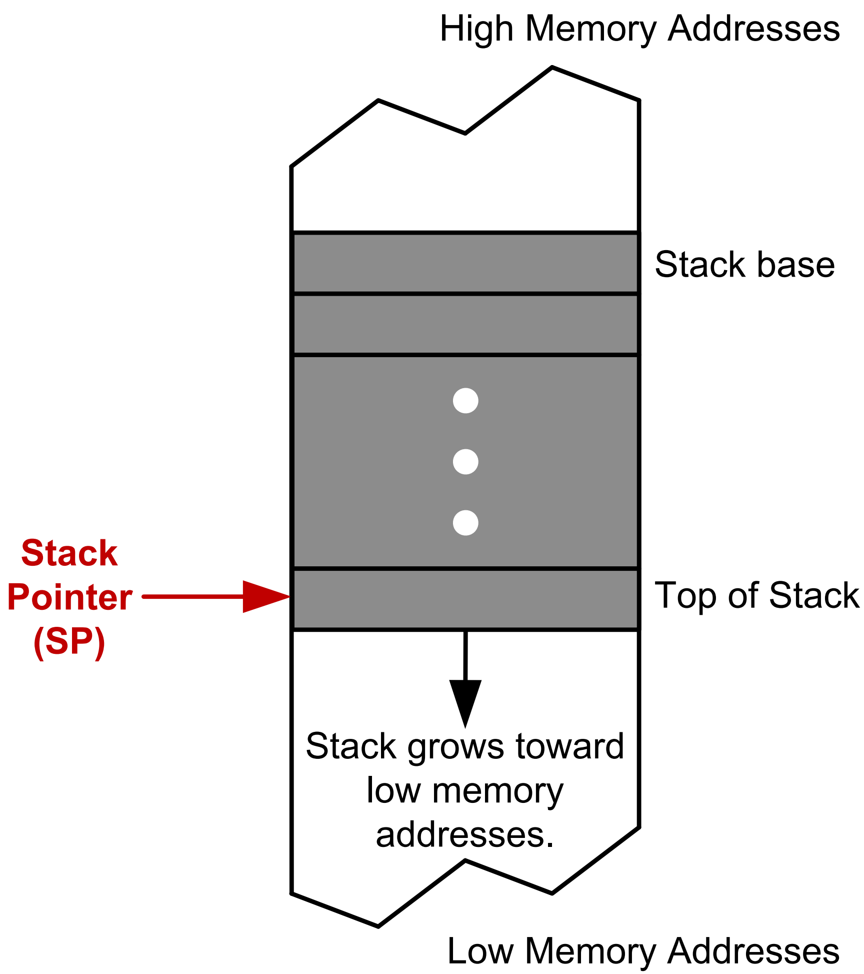
`LDMxx r0!, {r3,r1,r7,r2}`

	LDMIA	LDMIB	LDMDA	LDMDB
	Increment After	Increment Before	Decrement After	Decrement Before



LDMIA	LDMIB	LDMDA	LDMDB
r1 = 0	r1 = 4	r1 = -12	r1 = -16
r2 = 4	r2 = 8	r2 = -8	r2 = -12
r3 = 8	r3 = 12	r3 = -4	r3 = -8
r7 = 12	r7 = 16	r7 = -0	r7 = -4

Full Descending Stack



PUSH {register_list}
equivalent to:
STMDB SP!, {register_list}

DB: Decrement Before

POP {register_list}
equivalent to:
LDMIA SP!, {register_list}

IA: Increment After

Stack Implementation

Stock Name	Push		Pop	
	Equivalent	Alternative	Equivalent	Alternative
Full Descending(FD)	STMFD SP!,list	STMDB SP!,list	LDMFD SP!,list	LDMIA SP!,list
Empty Descending(ED)	STMED SP!,list	STMDA SP!,list	LDMED SP!,list	LDMIB SP!,list
Full Ascending(FA)	STMFA SP!,list	STMIB SP!,list	LDMFA SP!,list	LDMDA SP!,list
Empty Ascending(EA)	STMEA SP!,list	STMIA SP!,list	LDMEA SP!,list	LDMDB SP!,list

Typical Usage of Stack

- Why need stack?
  - Saving the original contents of processor's registers at the beginning a subroutine (Contents are restored at the end of a subroutine)
  - Storing local variables in a subroutine
  - Passing extra arguments to a subroutine
  - Saving processor's registers upon an interrupt

Stack

PUSH {*Rd*}

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

Push multiple registers

They are equivalent.

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

Stack

POP {*Rd*}

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

Pop multiple registers

They are equivalent.

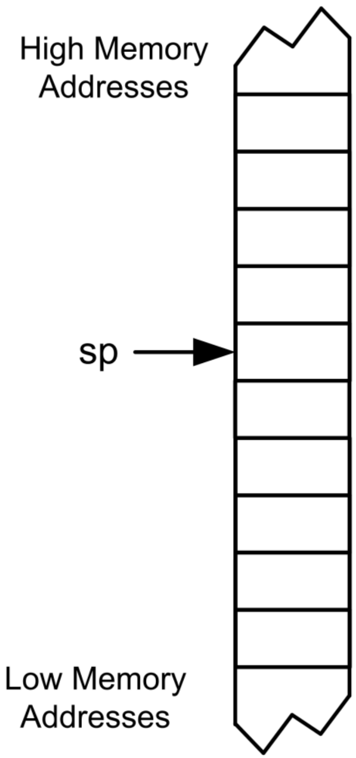
`POP {r6, r7, r8}` ⟺ `POP {r8, r7, r6}` ⟺

```
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.

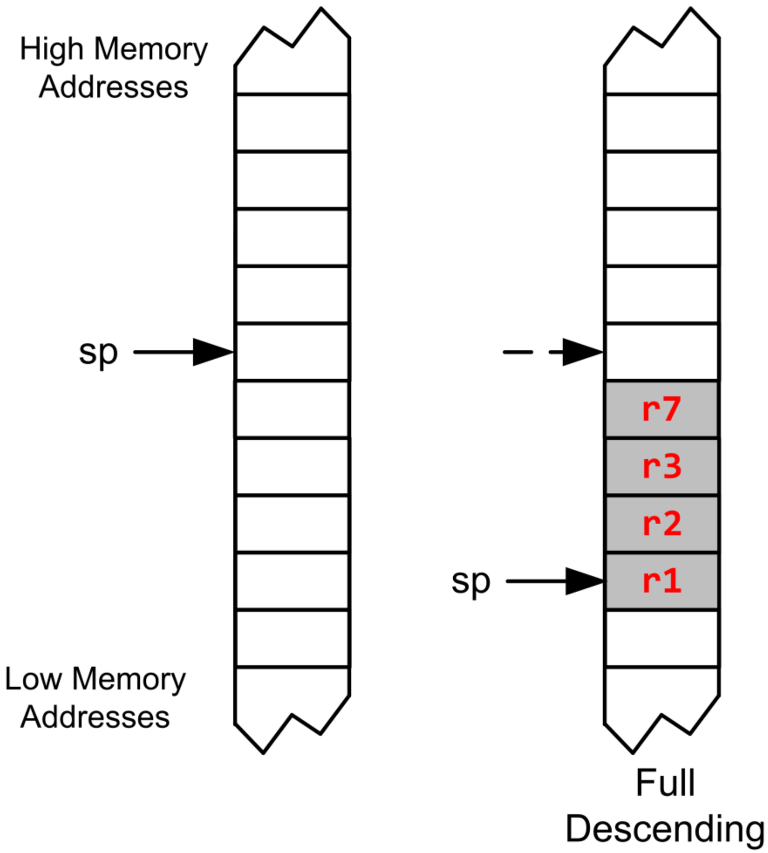
Full Descending Stack

PUSH {r3, r1, r7, r2}



Full Descending Stack

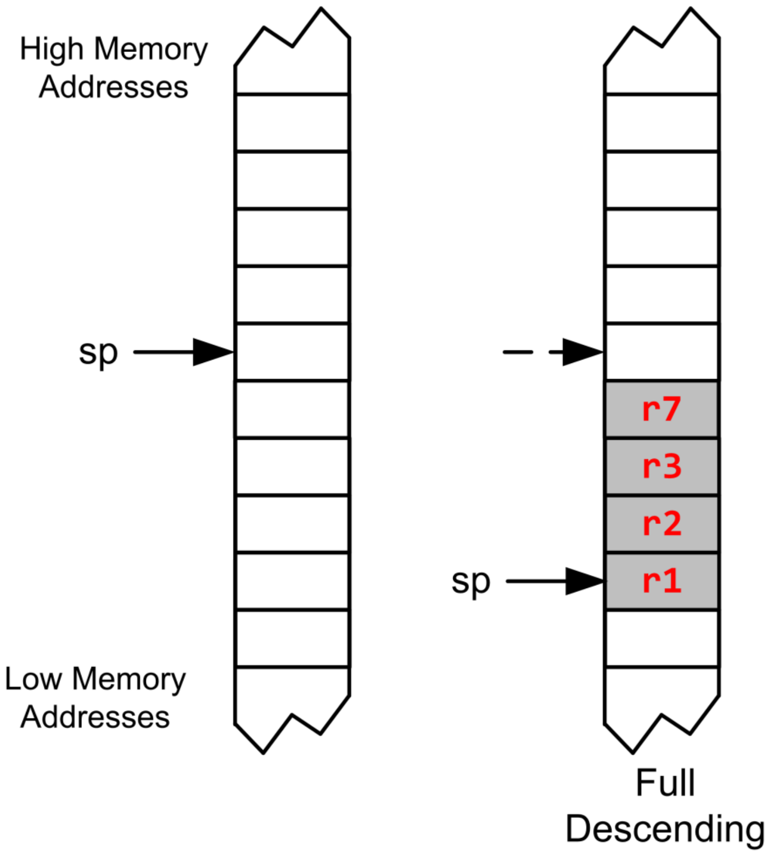
PUSH {r3, r1, r7, r2}



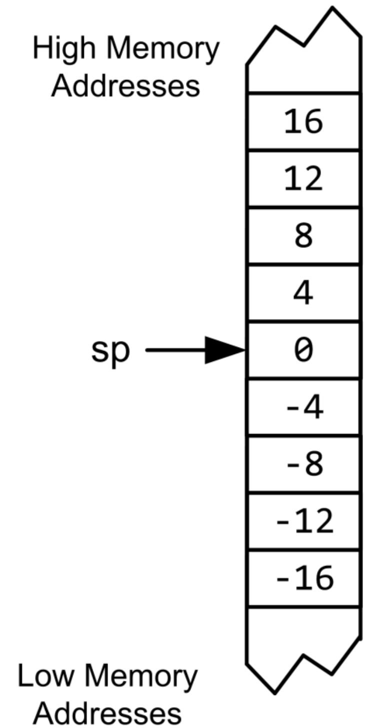
Largest-numbered register is pushed first.

Full Descending Stack

PUSH {r3, r1, r7, r2}

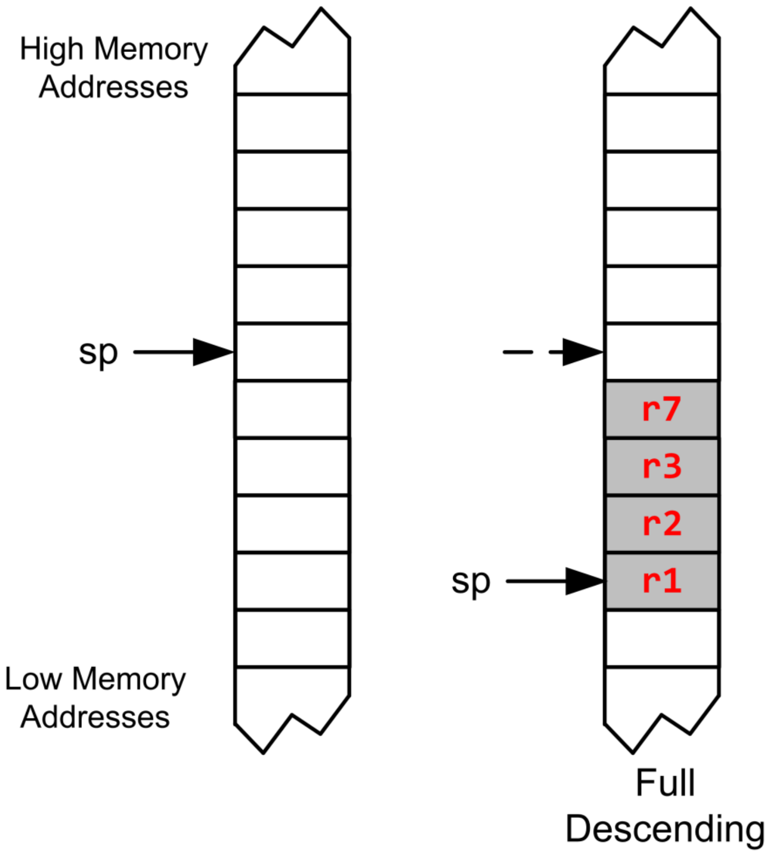


POP {r3, r1, r7, r2}

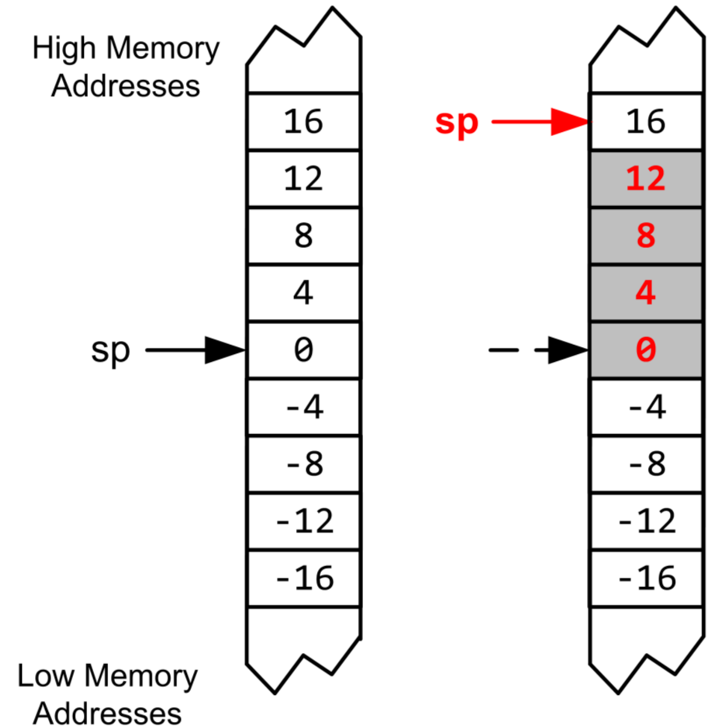


Full Descending Stack

PUSH {r3, r1, r7, r2}



POP {r3, r1, r7, r2}

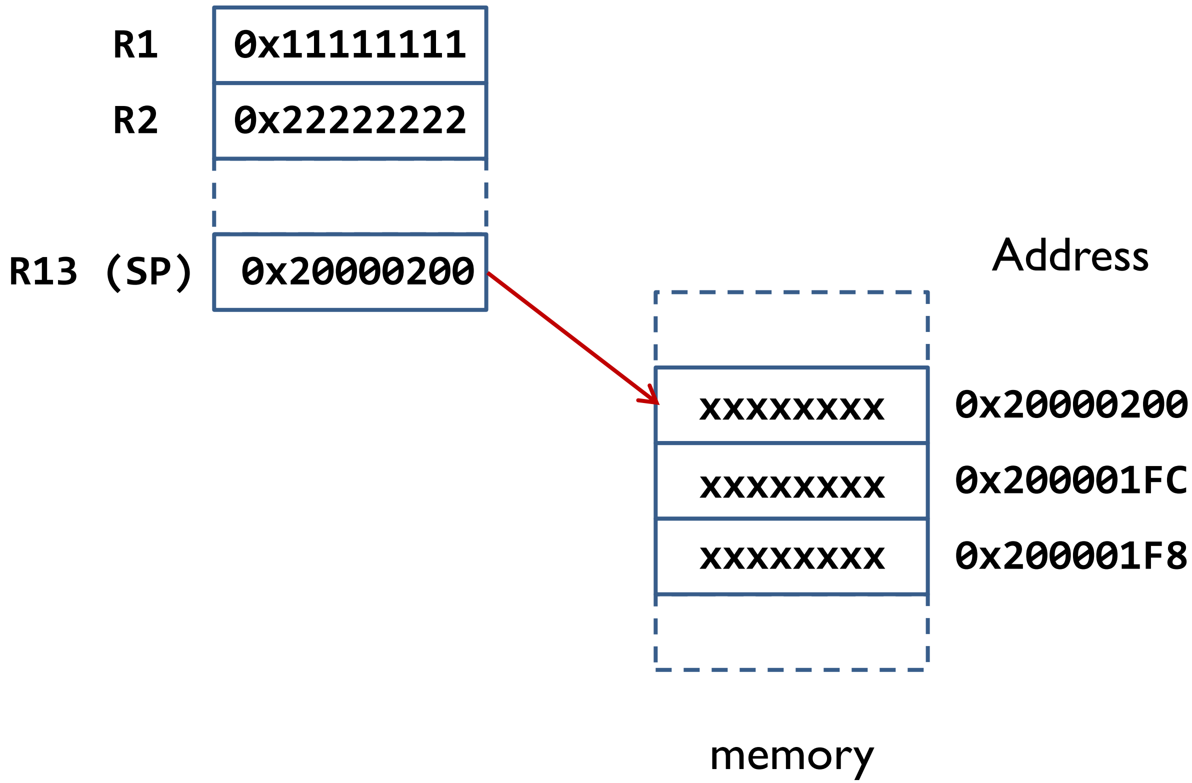


Smallest-numbered register is popped first.

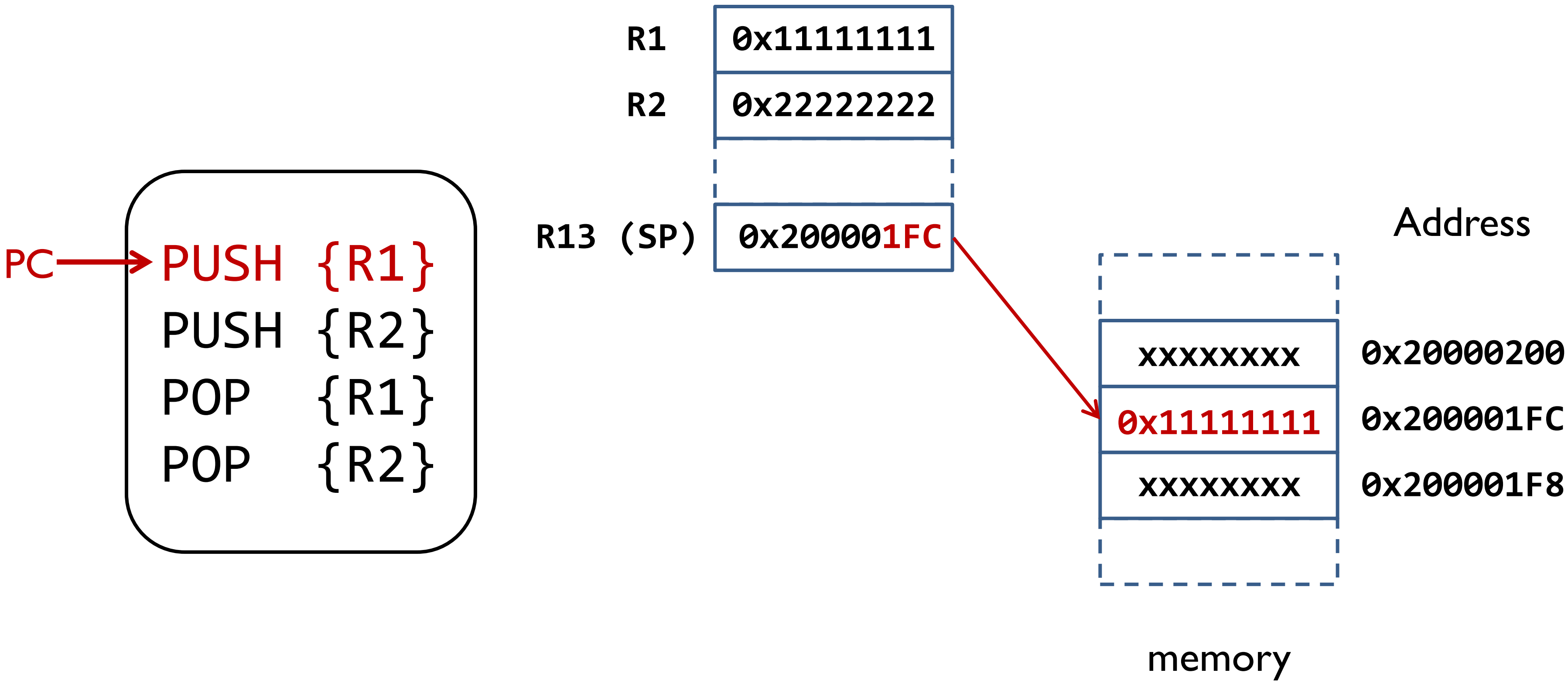
```
r1 = 0
r2 = 4
r3 = 8
r7 = 12
```

Example: swap R1 & R2

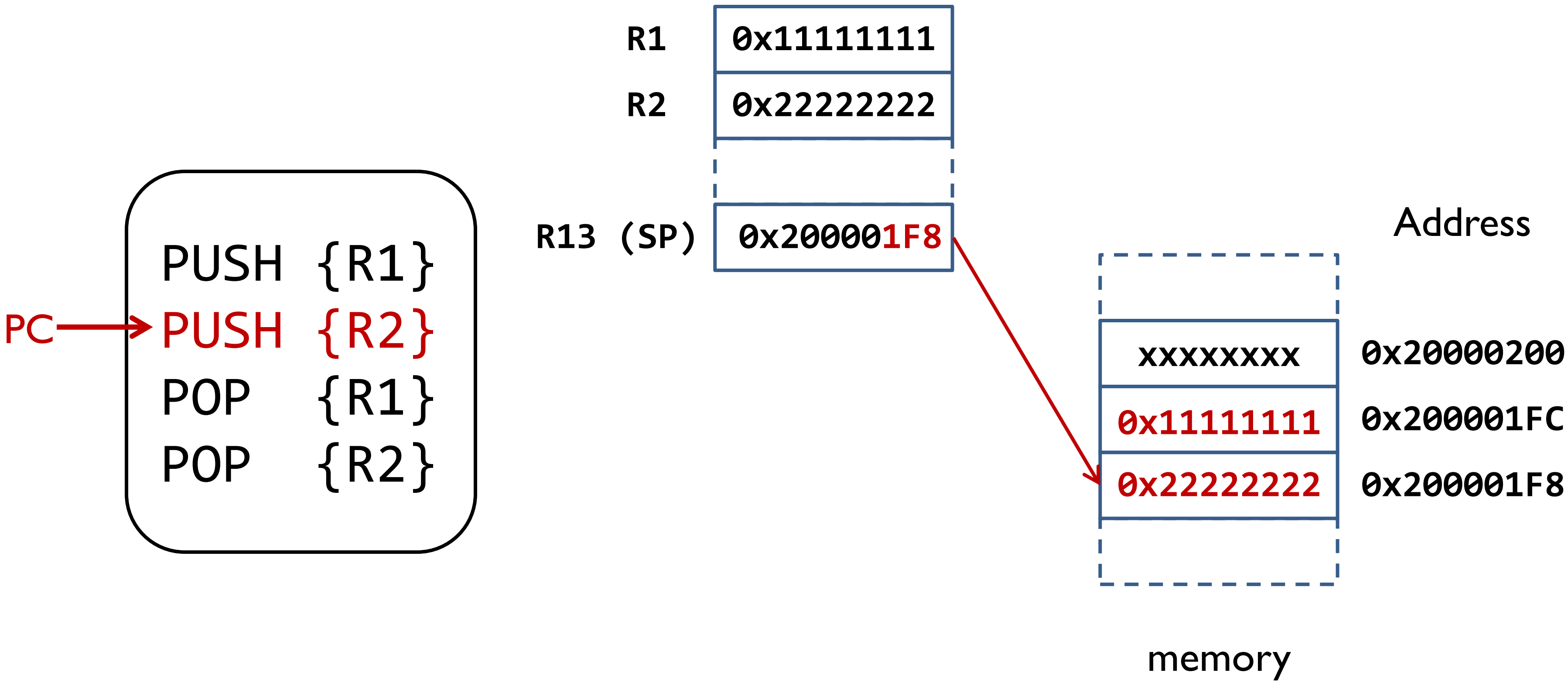
```
PUSH {R1}
PUSH {R2}
POP  {R1}
POP  {R2}
```



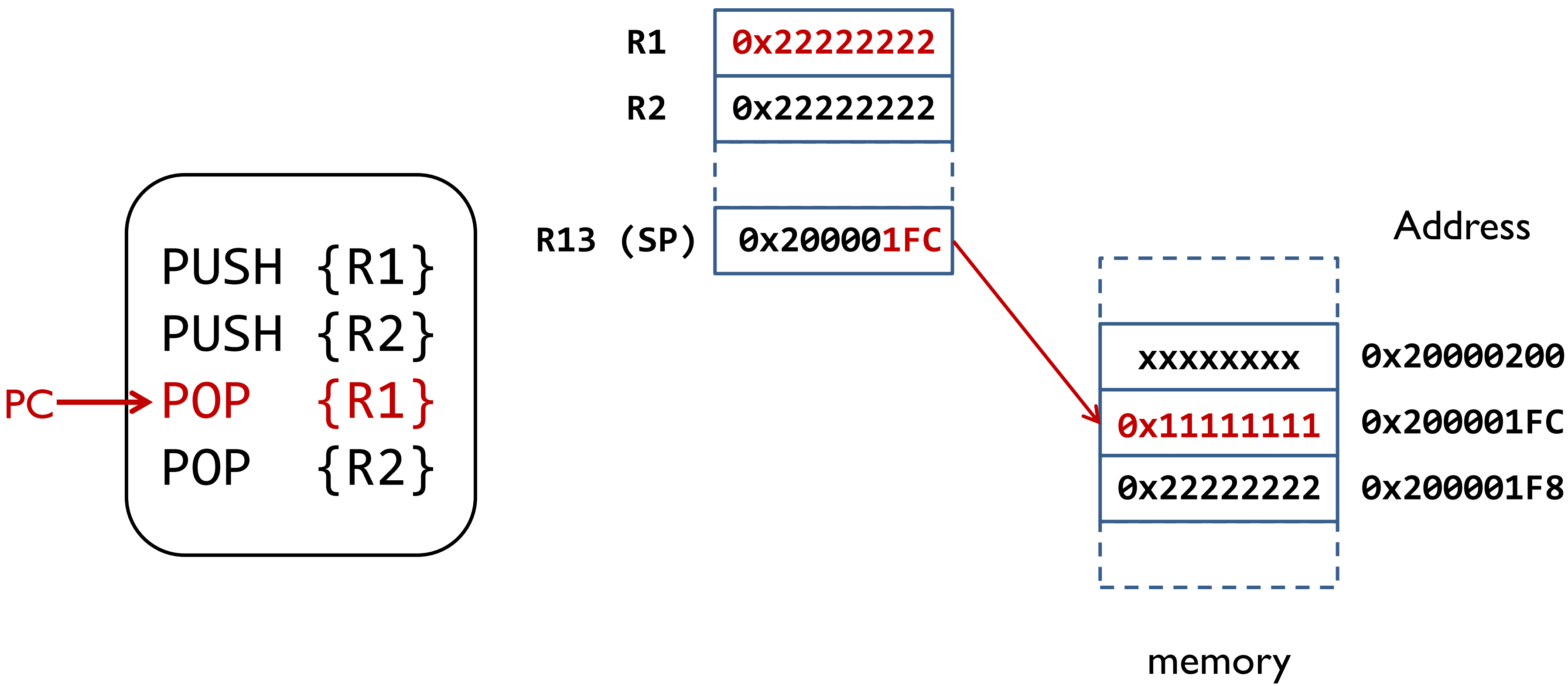
Example: swap R1 & R2



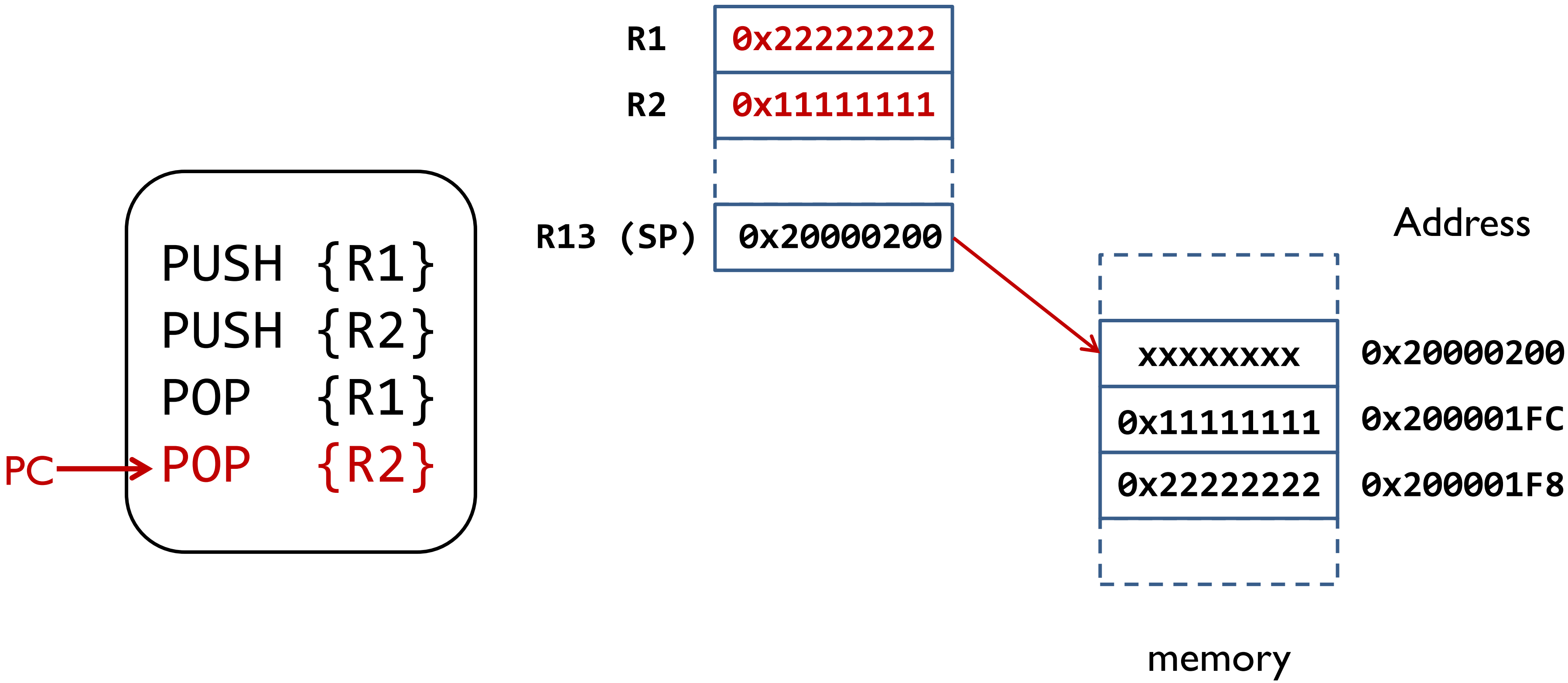
Example: swap R1 & R2



Example: swap R1 & R2



Example: swap R1 & R2



Quiz

Are the values of R1 and R2 swapped? Why?

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

Quiz ANS

Are the values of R1 and R2 swapped? Why?

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

Answer: No. It is equivalent to below, which preserves values of R1 and R2.

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

These are correct:

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

equivalent

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

or

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

equivalent

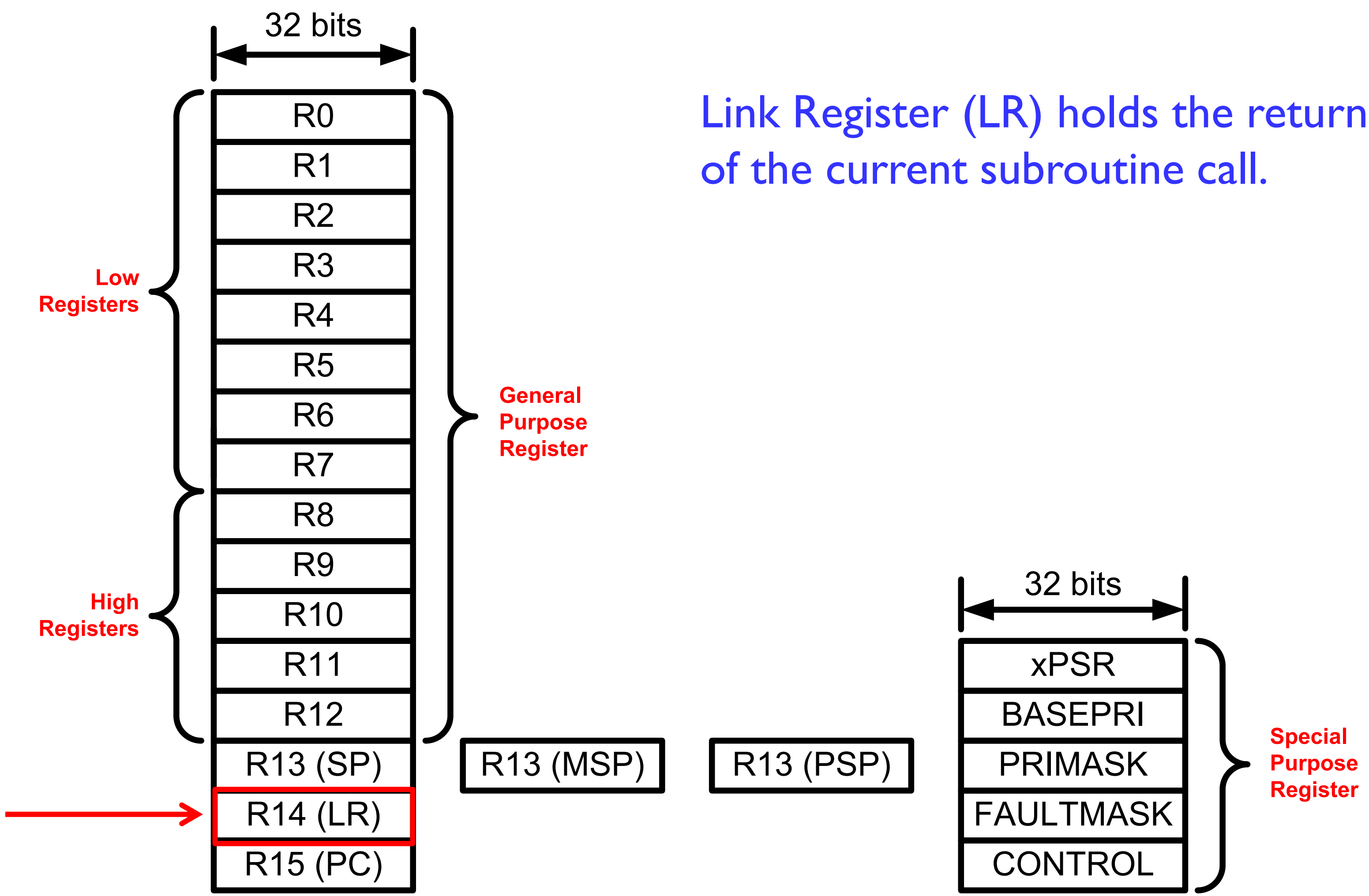
or

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

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



Link Register (LR) holds the return address of the current subroutine call.

Call a Subroutine

<table>
<tr><th>Caller Program</th><th>Subroutine/Callee</th></tr>
<tr><td><pre>
    MOV r4, #100
    ...
    BL   foo
    ...
    ADD r4, r4, #1   ; r4 = 101, not 11
</pre></td><td><pre>
foo PROC
    ...
    MOV   r4, #10 ; foo changes r4
    ...
    BX   LR
    ENDP
</pre></td></tr>
</table>

Calling a Subroutine

BL *label*

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

- Notes:
  - *label* is name of subroutine
  - 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

Caller Program

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

Subroutine/Callee

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

Branch and Exchange

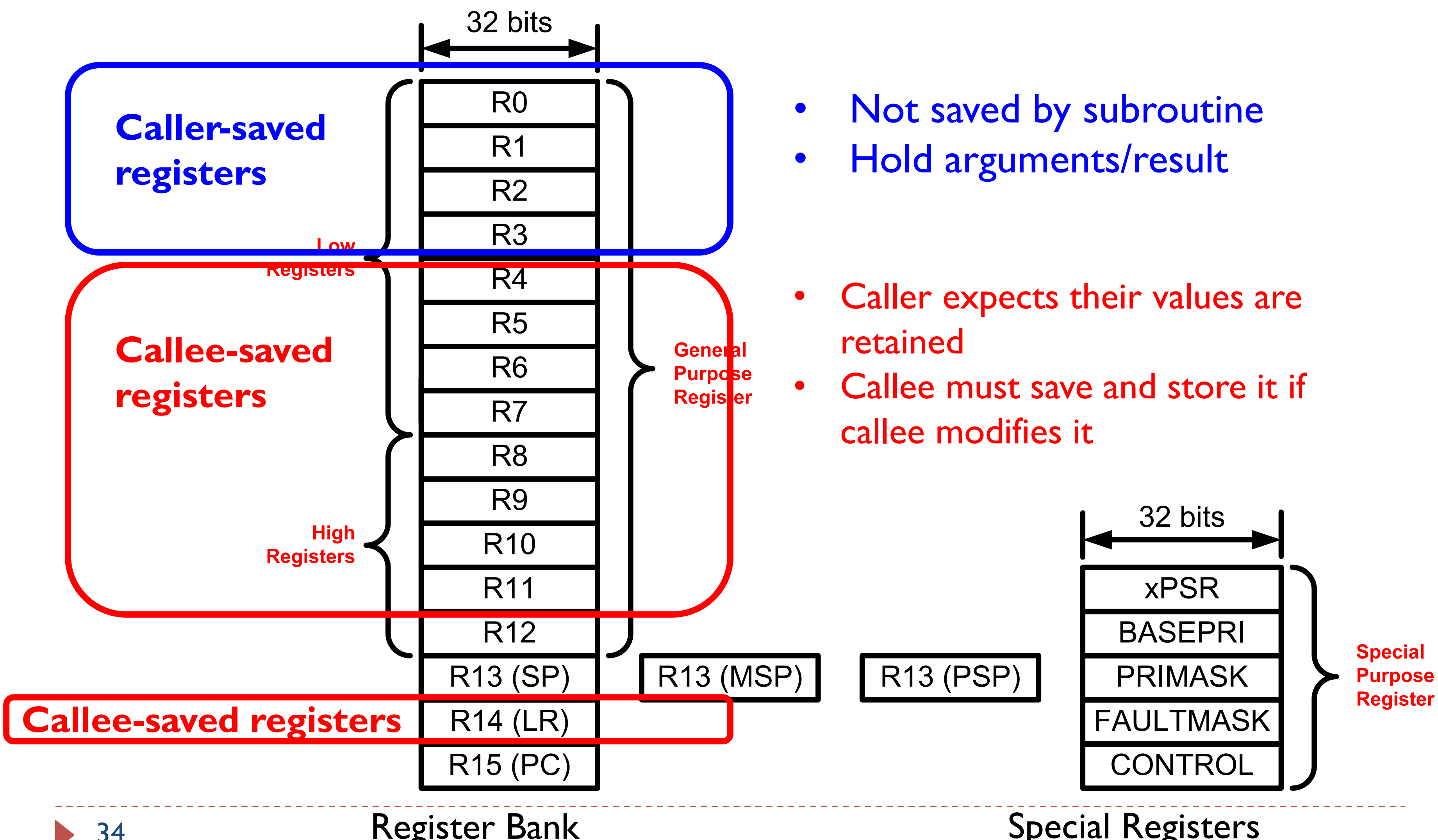
BX *LR*

- PC = 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	Yes/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

Caller-saved Registers *vs* Callee-saved Registers



- Not saved by subroutine
- Hold arguments/result

- Caller expects their values are retained
- Callee must save and store it if callee modifies it

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

Caller expects callee does not modify r4!

Callee should preserve r4!

Preserve Runtime Environment via Stack

Caller Program	Subroutine/Callee
Caller should save these registers if callers needs to re-use their original values: • R0 – R3 • R12 • CPSR	Callee should Preserve • R4 – R11 • R14 (LR) • R13 (SP)

Preserve Runtime Environment via Stack

What is wrong in foo()?

Caller Program	Subroutine foo	Subroutine bar

MOV r4, #100
...
BL foo
...
ADD r4, r4, #1
	foo PROC
	PUSH {r4}
	...
	MOV r4, #10
	...
	BL bar
	...
	POP {r4}
	BX LR
	ENDP
		bar PROC
		...
		BX LR
		ENDP

Preserve Runtime Environment via Stack

What is wrong in foo()?

Caller Program	Subroutine foo	Subroutine bar

MOV r4, #100
...
BL foo
...
ADD r4, r4, #1
	foo PROC
	PUSH {r4}
	...
	MOV r4, #10
	...
	BL bar
	...
	POP {r4}
	BX LR
	ENDP
		bar PROC
		...
		BX LR
		ENDP

The code shows a caller program calling subroutine foo, which pushes register r4 to preserve its value, modifies r4, calls another subroutine bar, then restores r4 and returns. The problem is that foo does not preserve LR, which holds the return address. When foo calls bar with BL bar, LR is overwritten. This means after bar returns, when foo tries to return with BX LR, it uses the LR value overwritten by bar, causing incorrect behavior.

Preserve Runtime Environment via Stack

What is wrong in foo()? Solution #1

Caller Program	Subroutine foo	Subroutine bar

`MOV r4, #100`
`...`
`BL foo`
`...`
`ADD r4, r4, #1`
	`foo PROC`
	`    PUSH {r4, LR}`
	`    ...`
	`    MOV  r4, #10`
	`    ...`
	`    BL   bar`
	`    ...`
	`    POP  {r4, LR}`
	`    BX   LR`
	`ENDP`
		`bar PROC`
		`    ...`
		`    BX   LR`
		`ENDP`

Here foo pushes both r4 and LR at the start and pop them before returning. This preserves the original return address in LR across the call to bar. The subroutine foo now: Saves r4 and LR on entry; Modifies r4, calls bar; Restores r4 and LR; Returns correctly via the restored LR. This ensures both the callee-saved register (r4) and the link register are preserved properly.

Preserve Runtime Environment via Stack

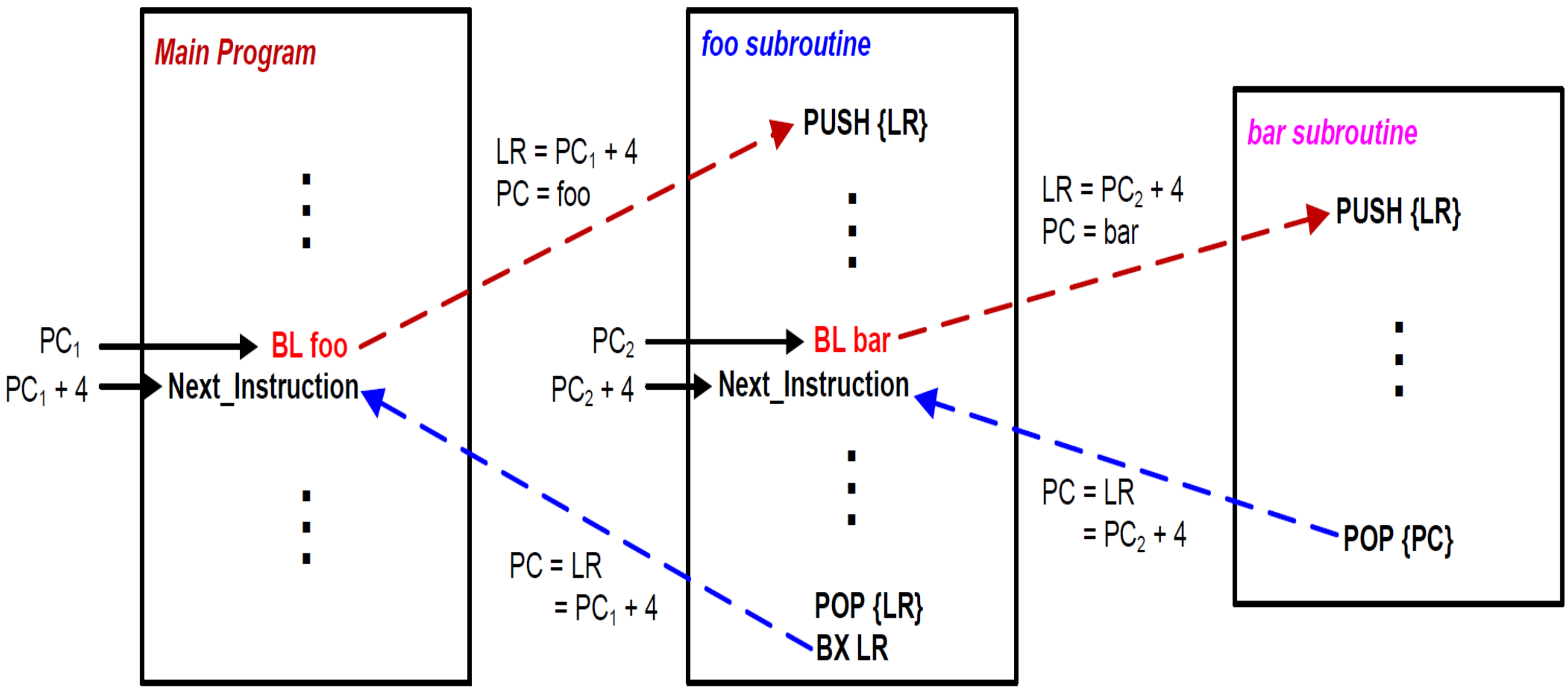
What is wrong in foo()? Solution #2

Caller Program	Subroutine foo	Subroutine bar

MOV r4, #100
...
BL foo
...
ADD r4, r4, #1
	foo PROC
	PUSH {r4, LR}
	...
	MOV r4, #10
	...
	BL bar
	...
	POP {r4, PC}
	~~BX LR~~
	ENDP
		bar PROC
		...
		BX LR
		ENDP

Here foo pushes {r4, LR} but pops {r4, PC} instead of {r4, LR}. Popping into PC (Program Counter) directly performs the return by loading the return address into PC. This method eliminates the need for an explicit BX LR instruction because popping PC causes an immediate return.

Stacks and Subroutines



Subroutine Calling Another Subroutine

```
MAIN
   MOV R0,#2
   BL QUAD
ENDL    ...
```

Function MAIN

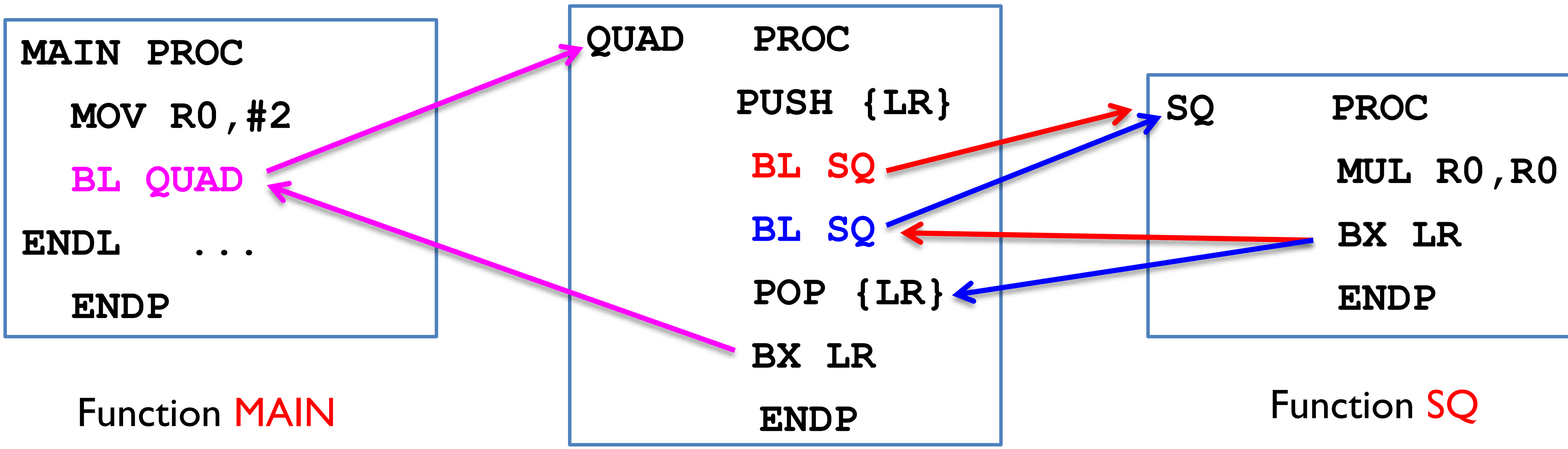
```
QUAD      PUSH {LR}
          BL SQ
          BL SQ
          POP {LR}
          BX LR
```

Function QUAD

```
SQ        MUL R0,R0
          BX LR
```

Function SQ

Subroutine Calling Another Subroutine



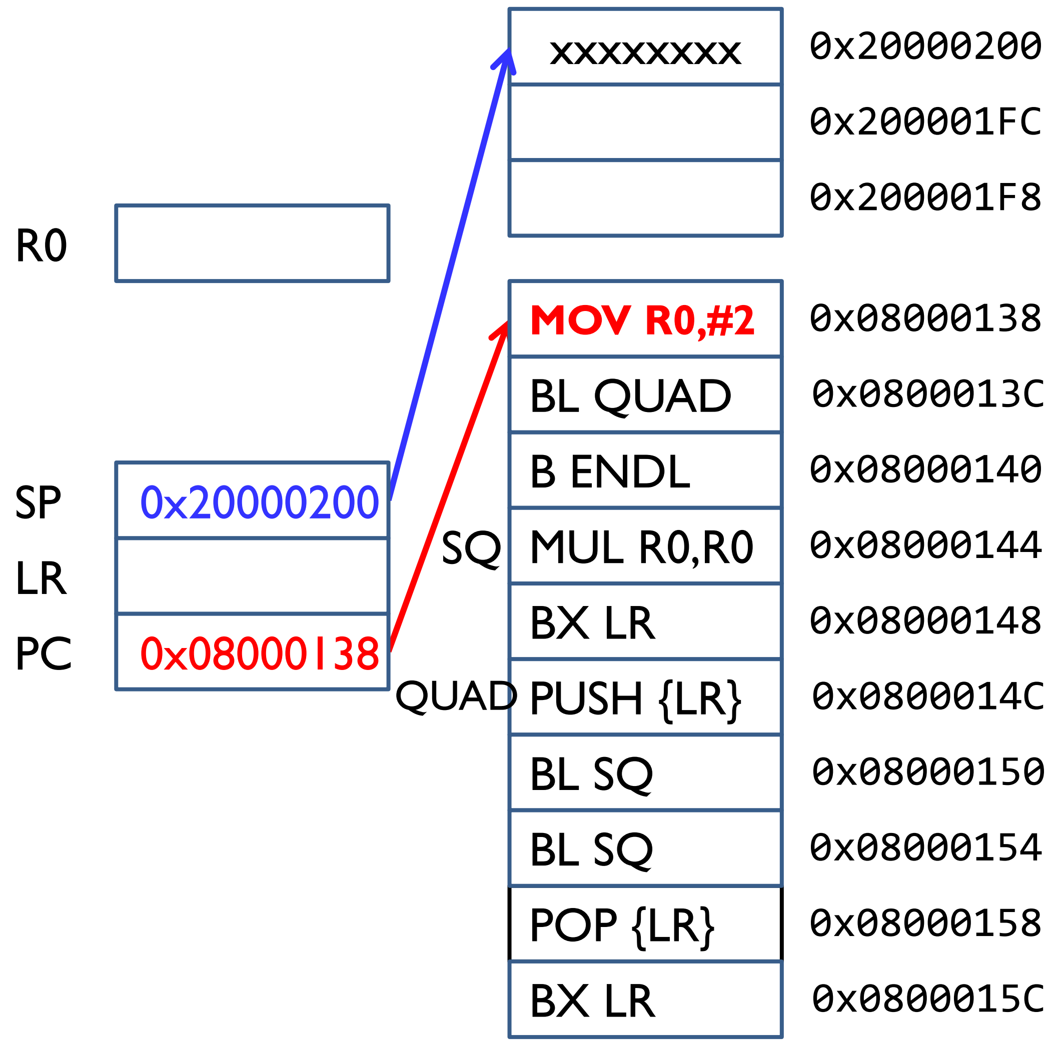
Function MAIN

Function QUAD

Function SQ

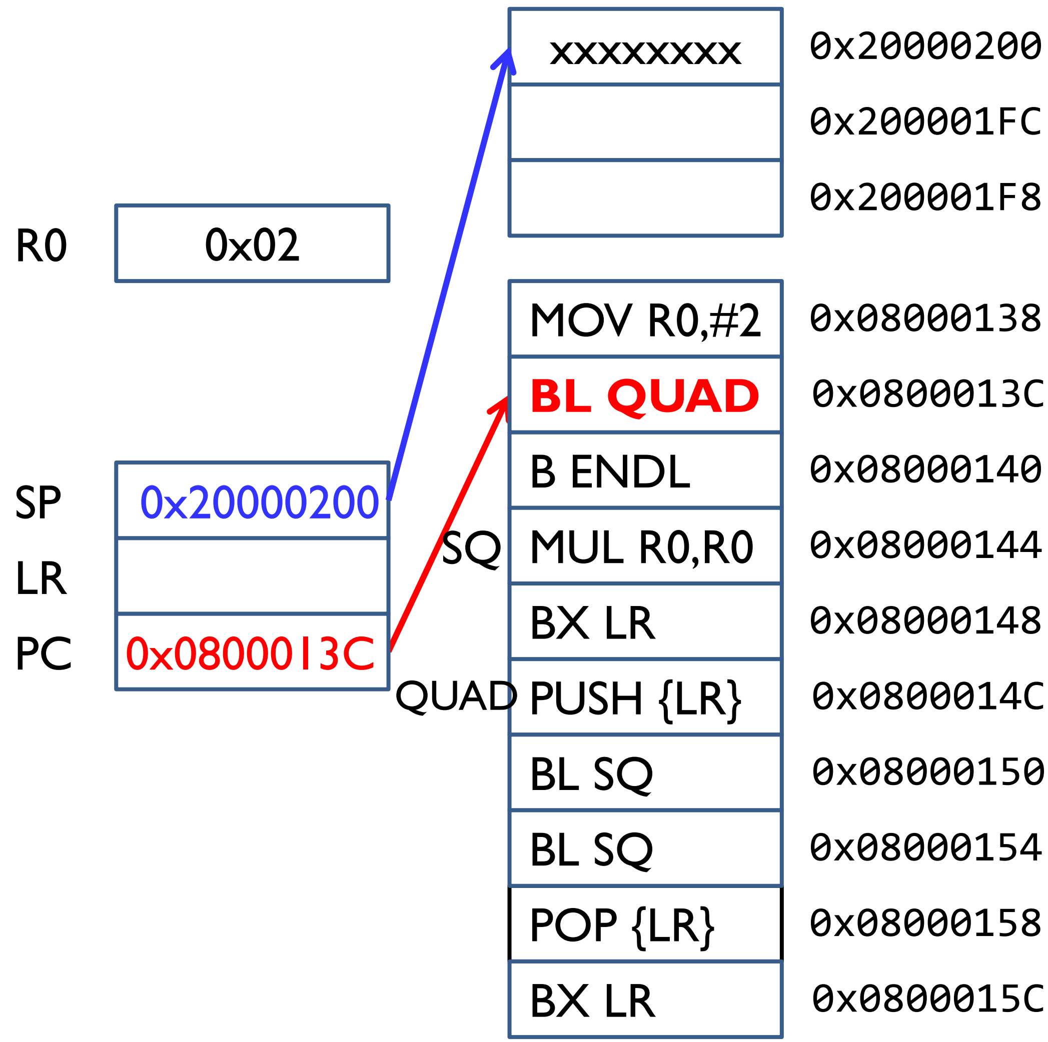
Example: $R0 = R0^4$

```
          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      ...
```



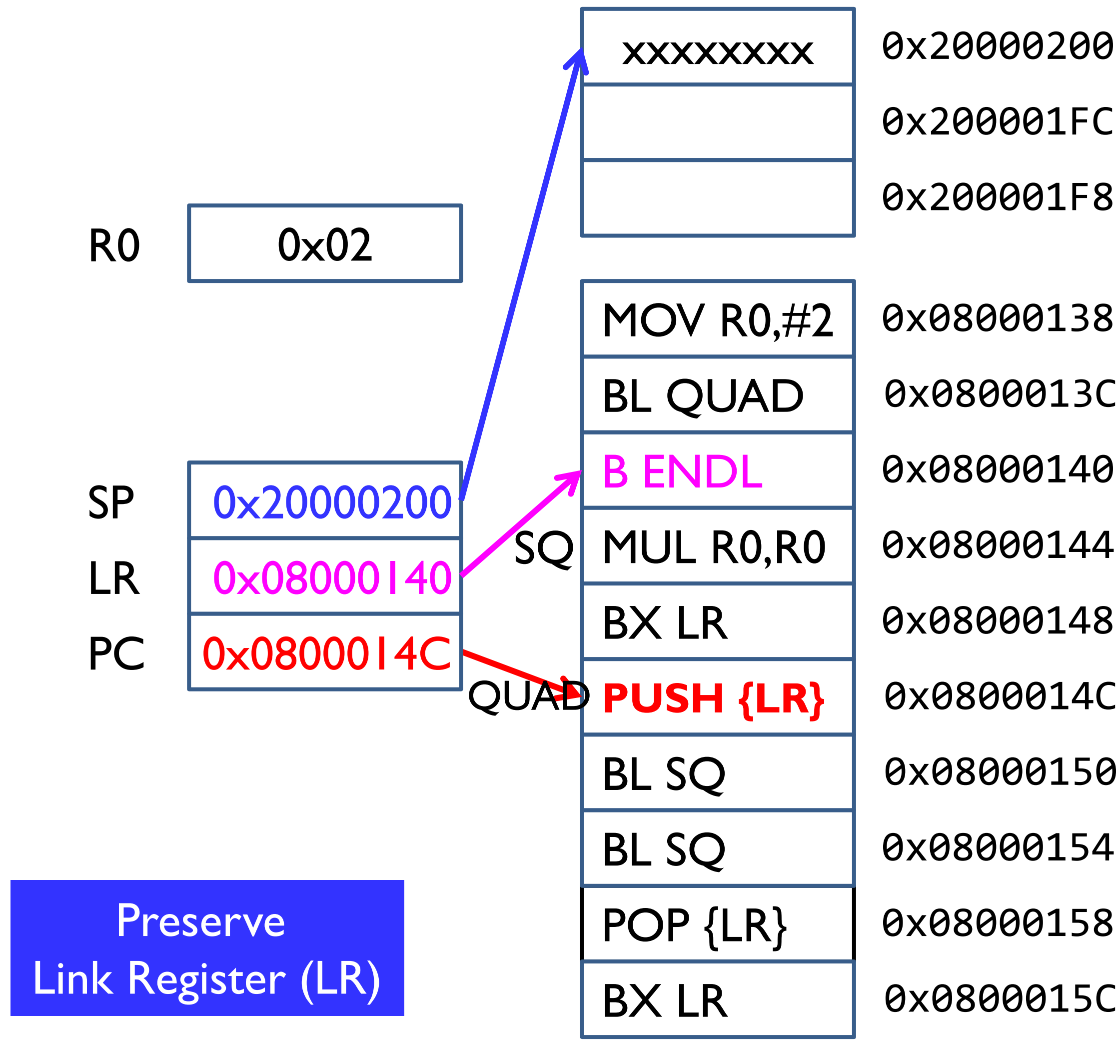
Example: R0 = $R0^4$

```
          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      ...
```



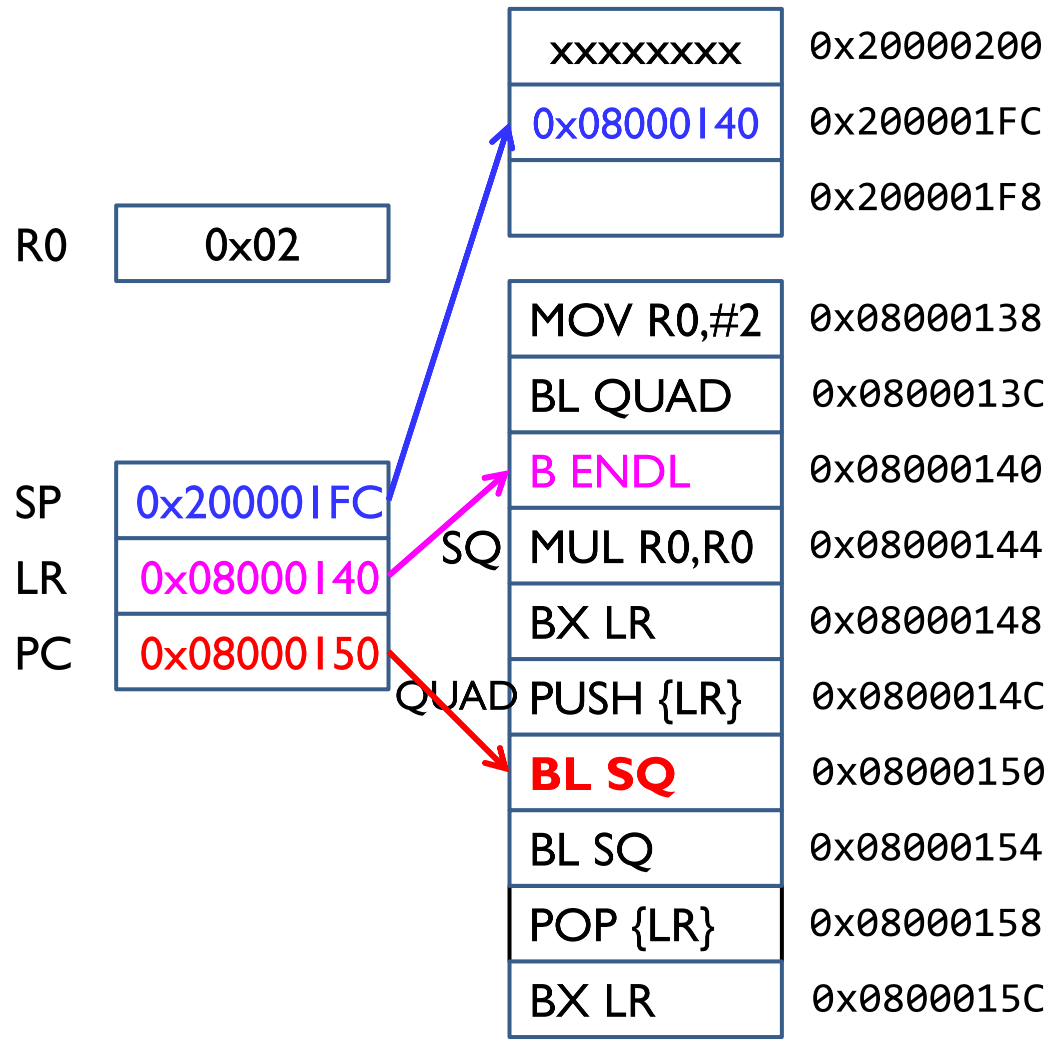
Example: R0 = $R0^4$

```
        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    ...
```



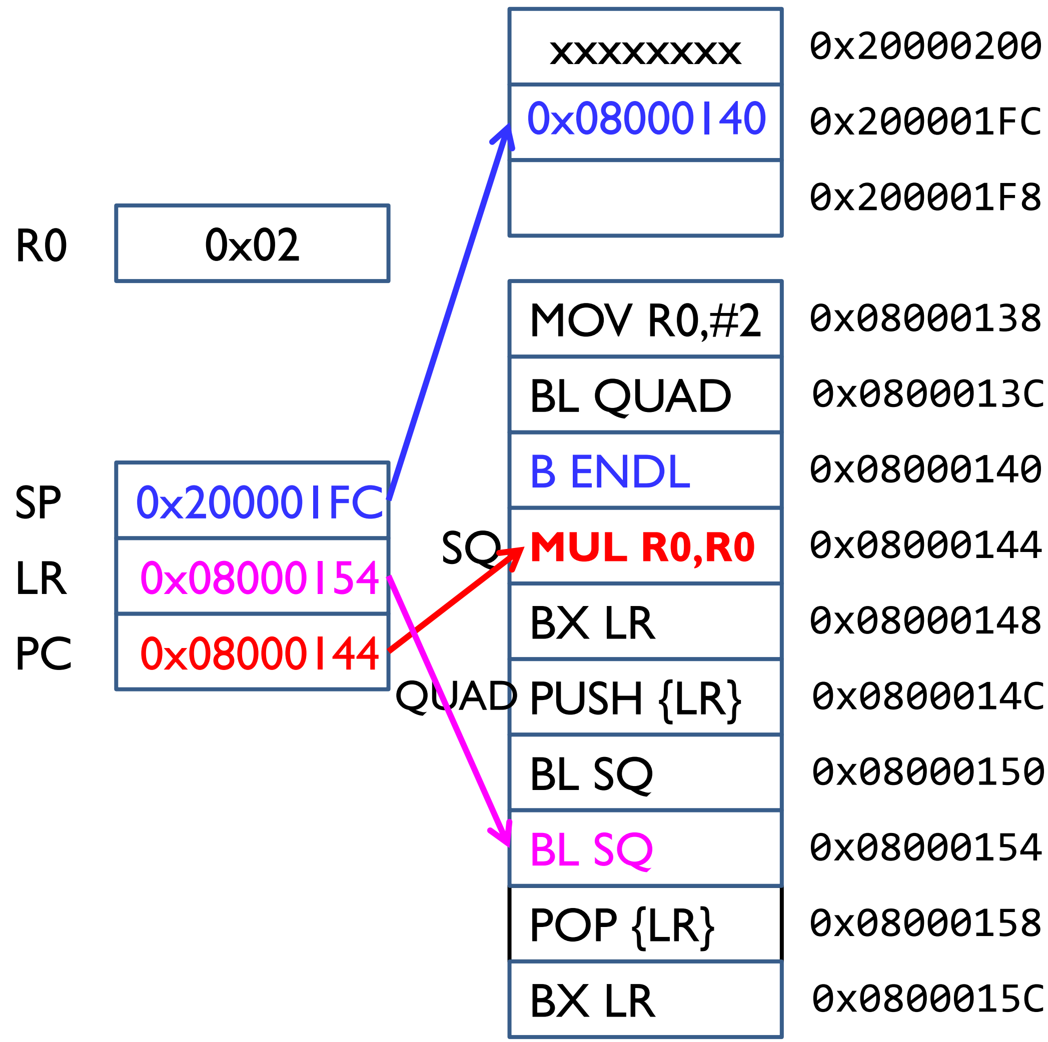
Example: $R0 = R0^4$

```
        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    ...
```



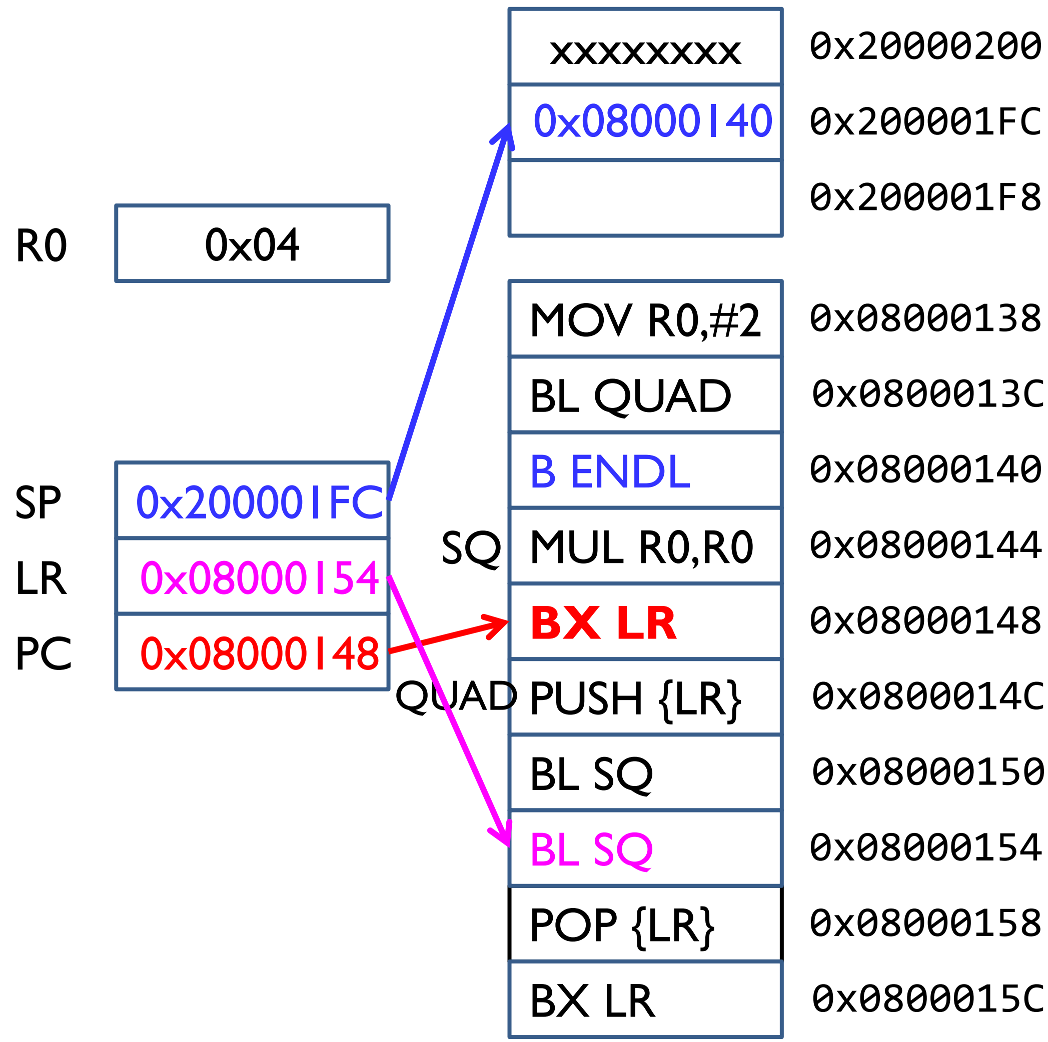
Example: R0 = $R0^4$

```
        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    ...
```



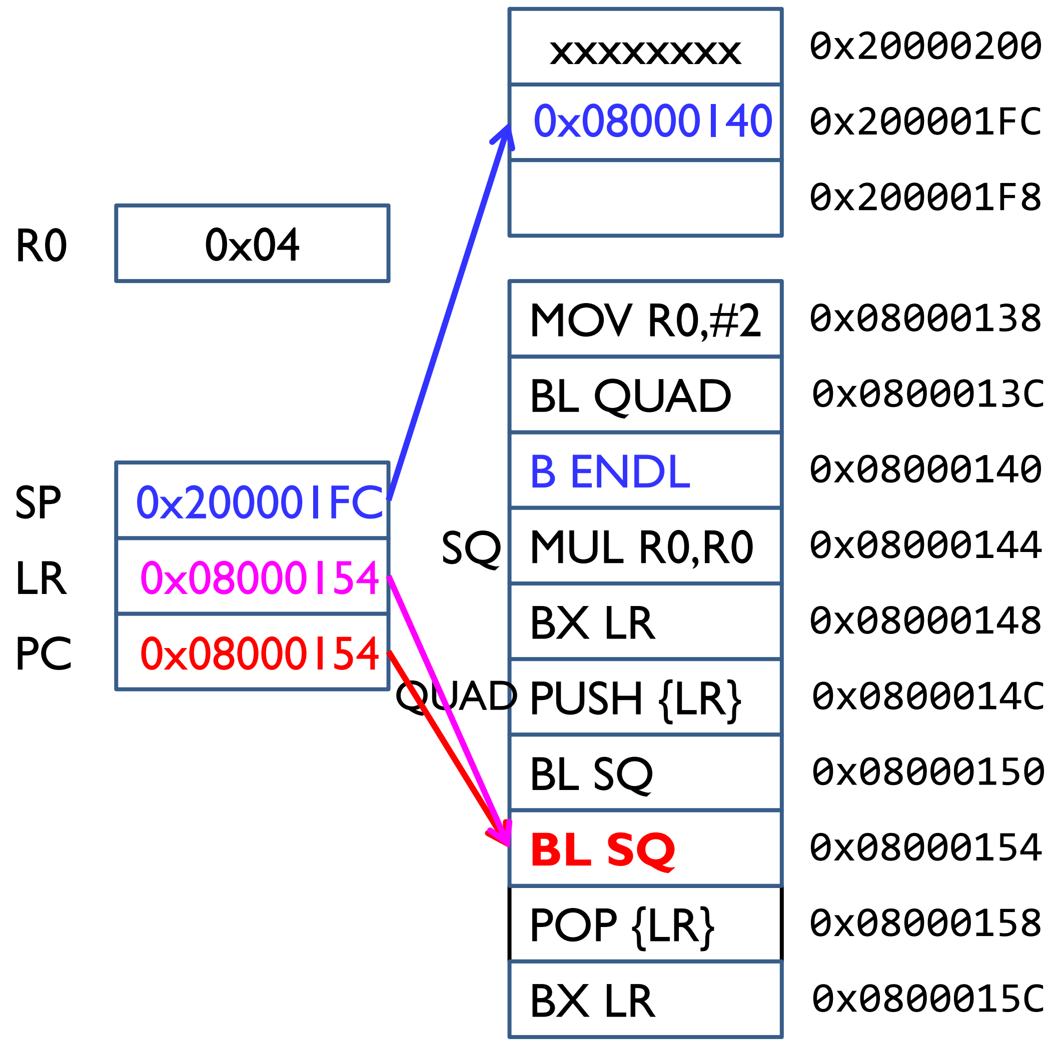
Example: $R0 = R0^4$

```
        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    ...
```



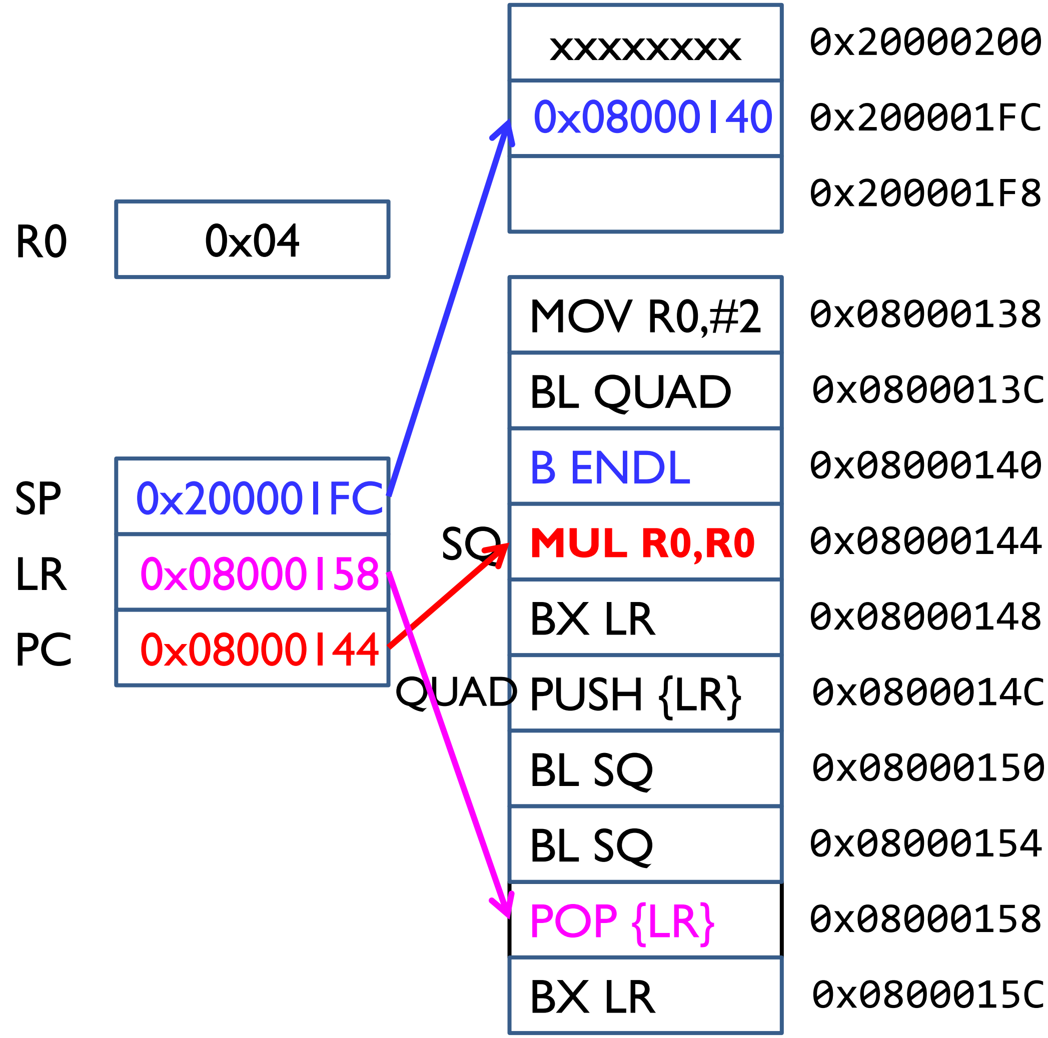
Example: $R0 = R0^4$

```
        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    ...
```



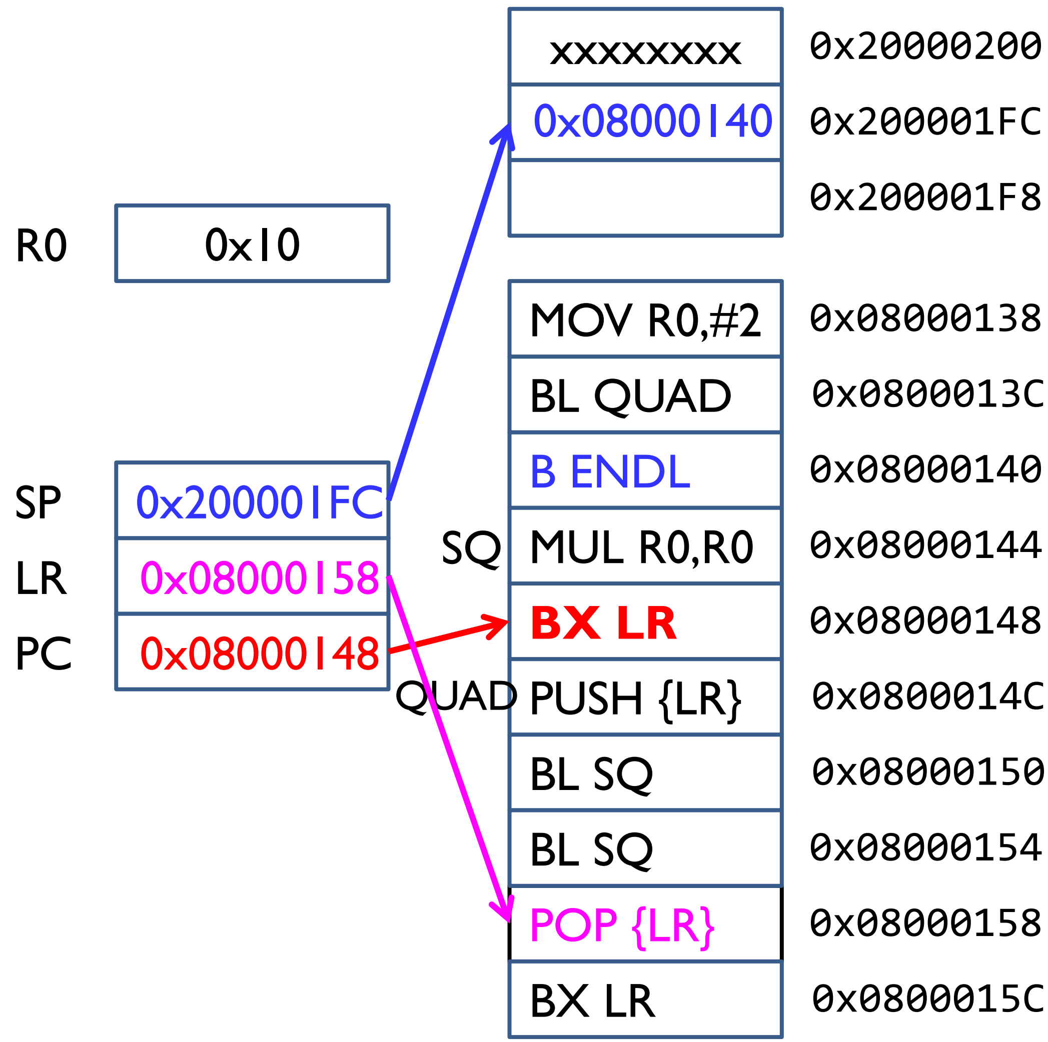
Example: R0 = R0^4

```
        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    ...
```



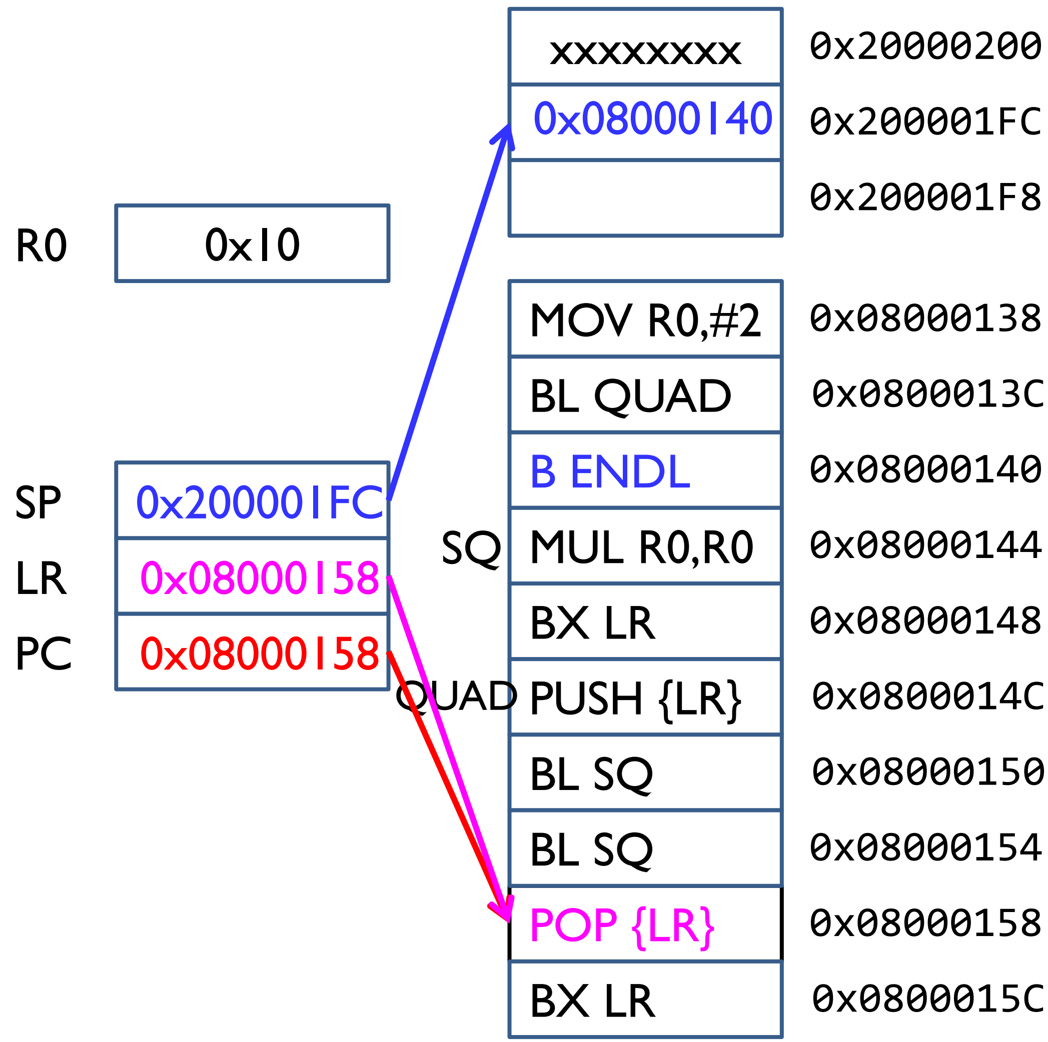
Example: $R0 = R0^4$

```
        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    ...
```



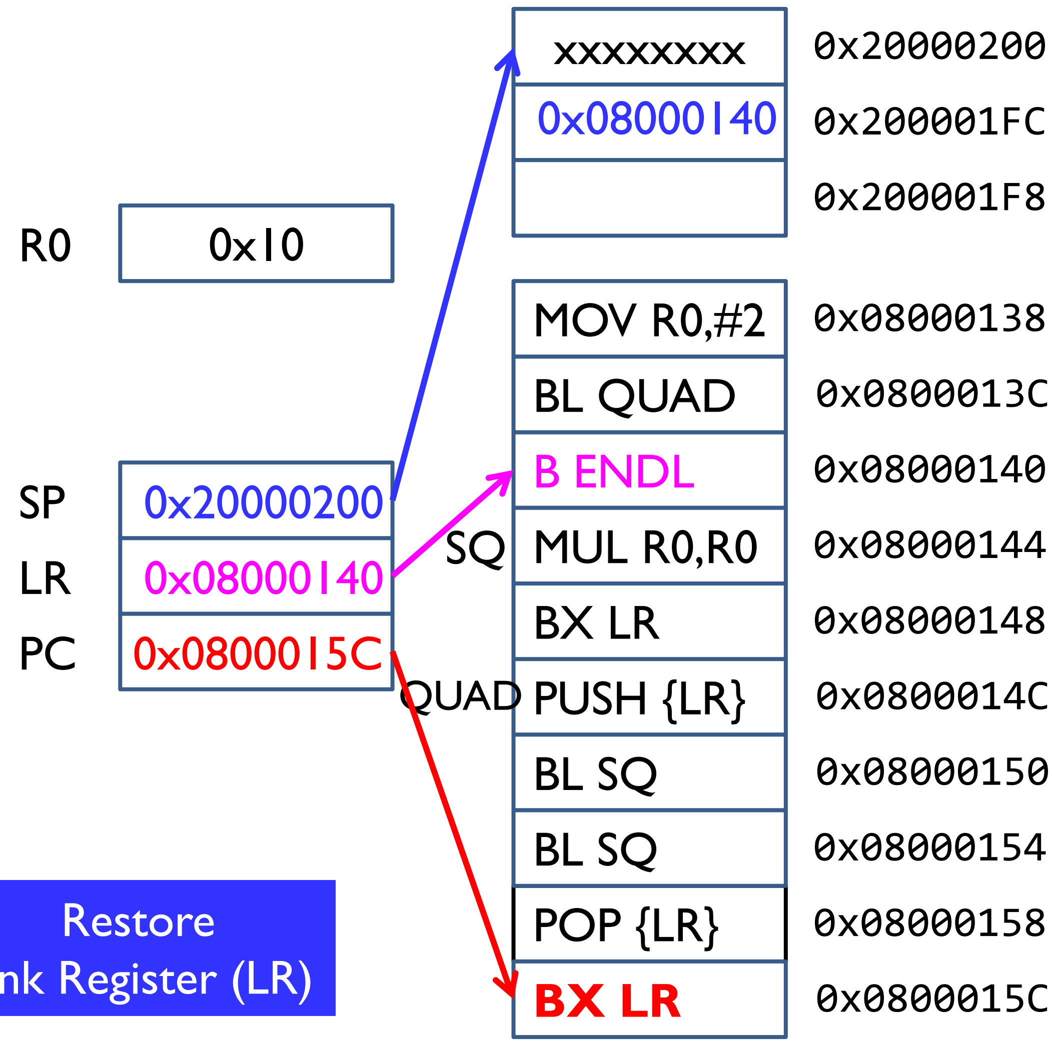
Example: R0 = R0^4

```
        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    ...
```



Example: R0 = R0^4

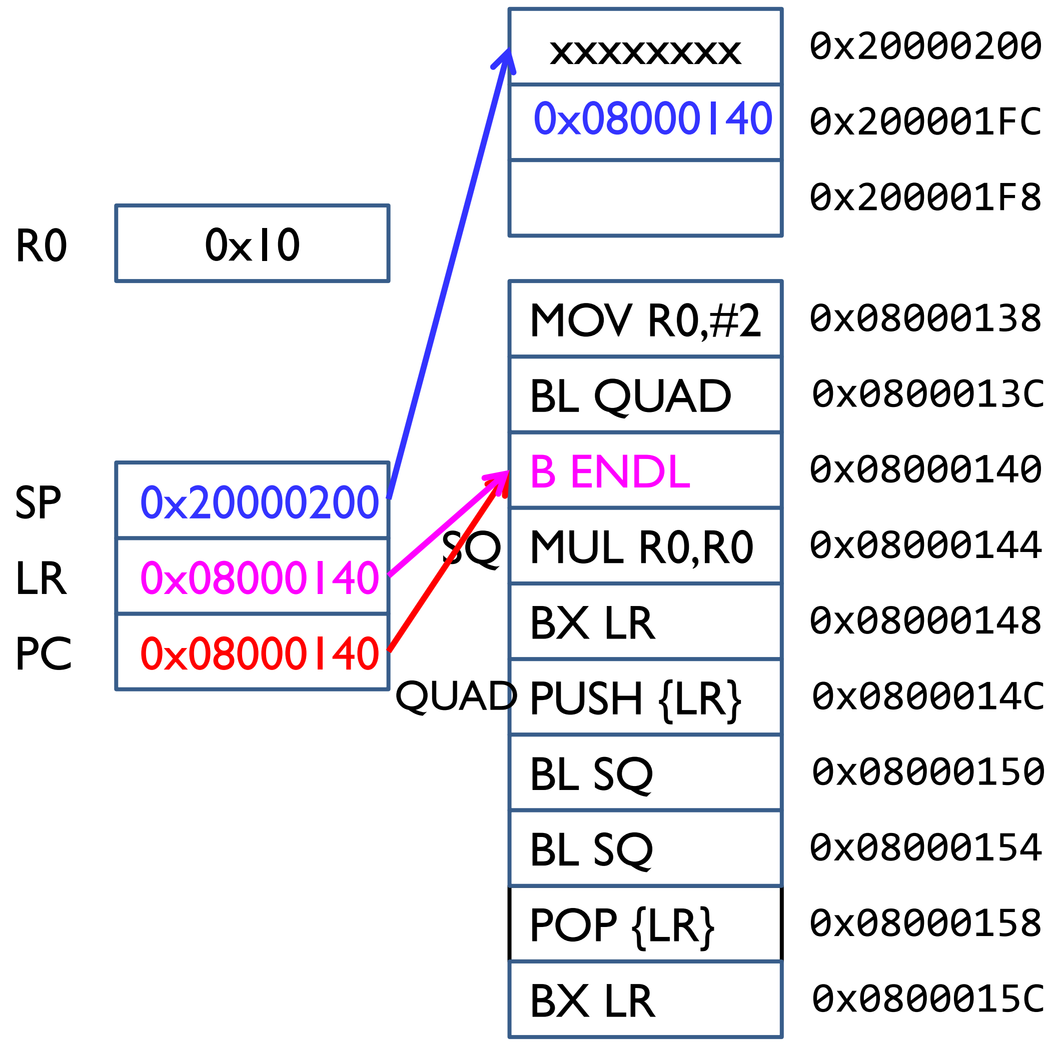
```
            MOV R0,#2
            BL QUAD
            B ENDL

SQ          MUL R0,R0
            BX LR

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



Restore Link Register (LR)

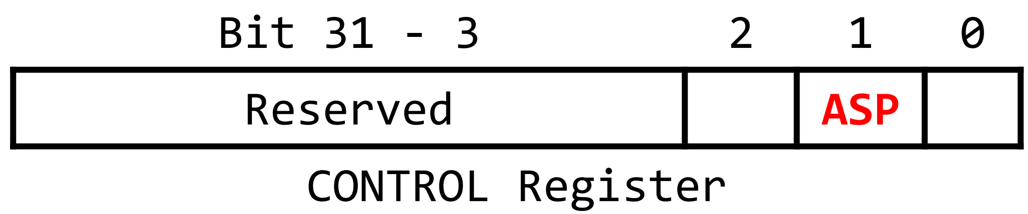
Example: R0 = $R0^4$

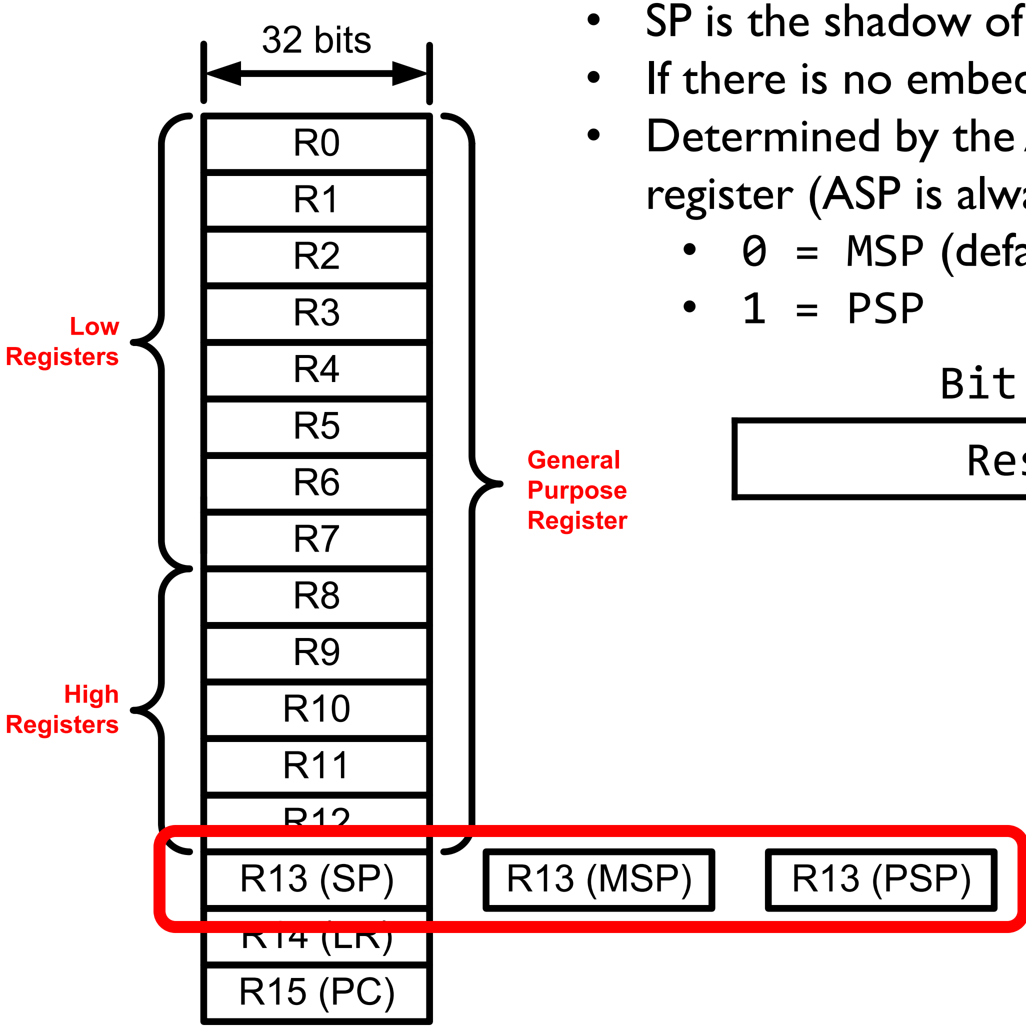
```
            MOV R0,#2
            BL QUAD
            B ENDL

SQ          MUL R0,R0
            BX LR

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



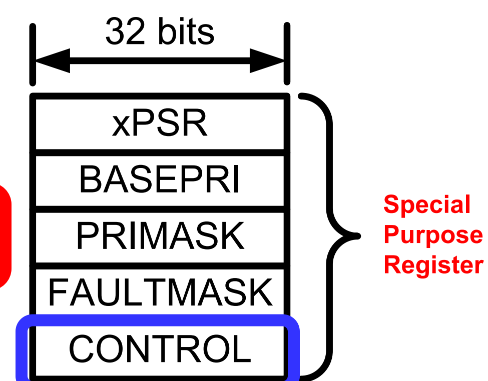
Stack Pointer (SP)

- SP is the shadow of MSP (Main SP) or PSP (Process SP)
- If there is no embedded OS, PSP is not used
- Determined by the ASP (Active SP) bit in the CONTROL register (ASP is always 0 in handler mode).
  - `0 = MSP` (default)
  - `1 = PSP`





Register Bank

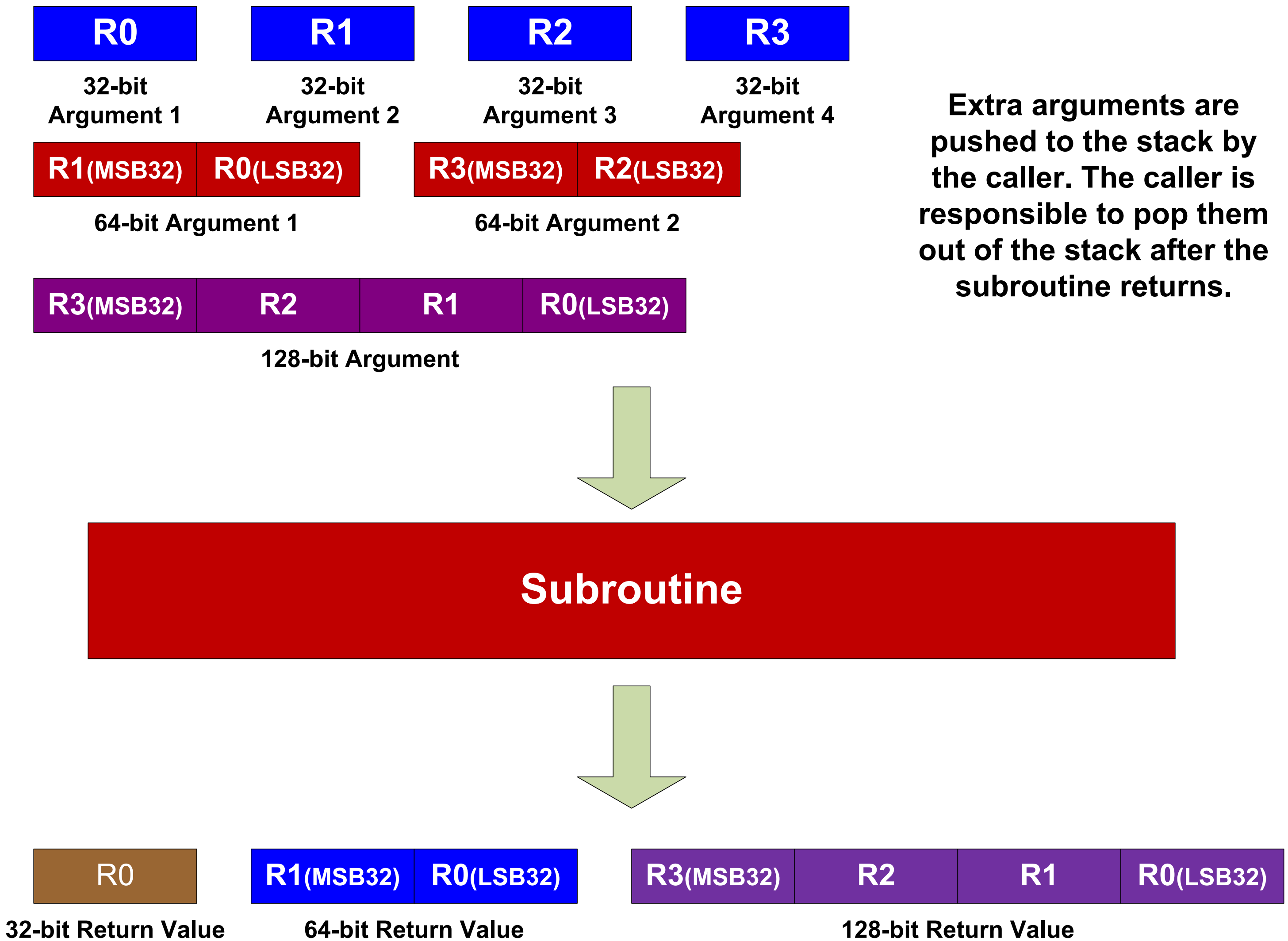


Special Registers

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.

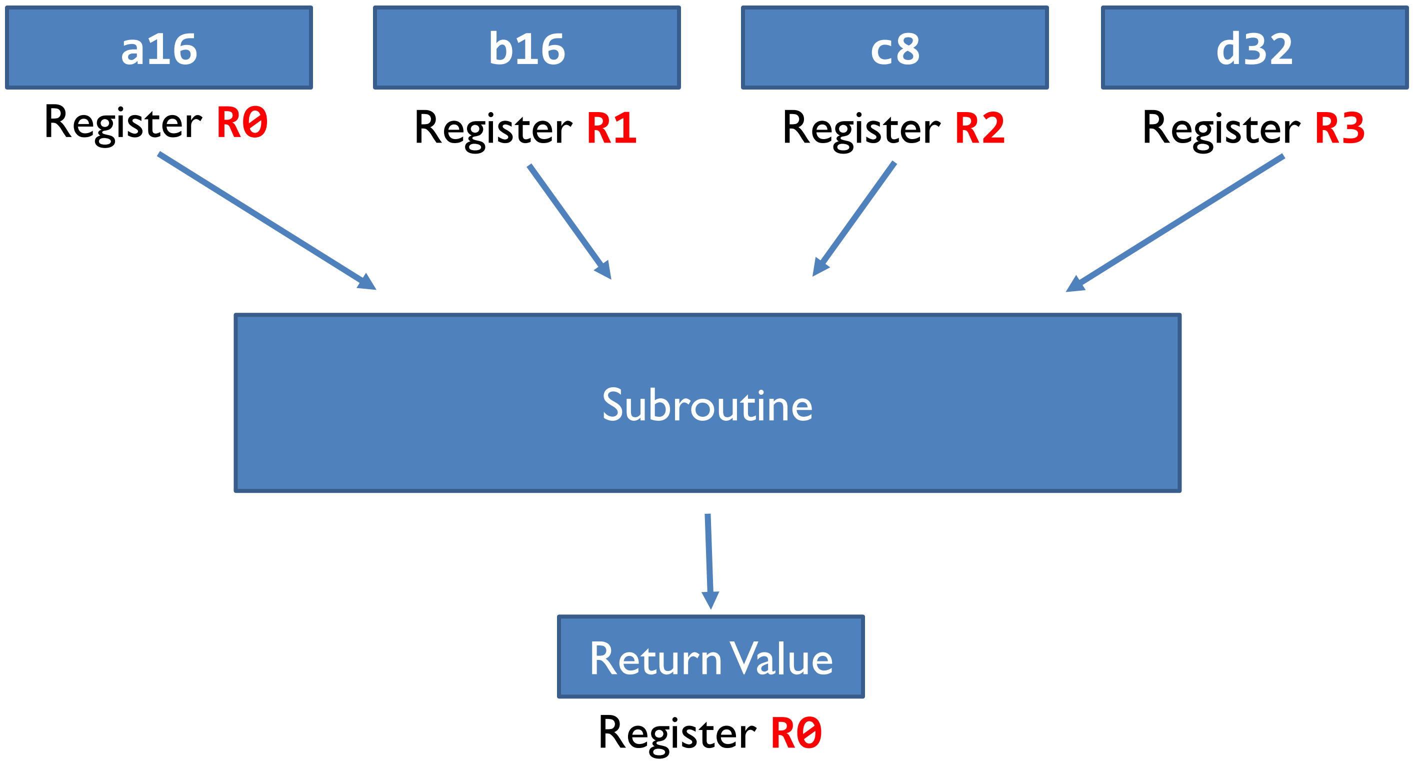
Passing Arguments into a Subroutine



Extra arguments are pushed to the stack by the caller. The caller is responsible to pop them out of the stack after the subroutine returns.

Passing Arguments into a Subroutine

```
int32_t sum(int16_t a16, int16_t b16, int8_t c8, int32_t d32);
```



Passing 4 Arguments

```
int32_t sum(int16_t a16, int16_t b16, int8_t c8, int32_t d32);
```

```
s = sum(1, 2, 3, 4);
```

Caller

```
   MOVS r0, #1 ; a16
   MOVS r1, #2 ; b16
   MOVS r2, #3 ; c8
   MOVS r3, #4 ; d32
   BL   sum
```

Callee

```
sum PROC
   ADD r0, r0, r1 ; a16 + b16
   ADD r0, r0, r2 ; add c8
   ADD r0, r0, r3 ; add d32
   BX  LR         ; return
   ENDP
```

Passing Extra Arguments via Stack

```
int32_t sum(int32_t a, int32_t b, int32_t c, int32_t d,
            int32_t h, int32_t i, int32_t j, int32_t k);
```

```
s = sum(1, 2, 3, 4, 5, 6, 7, 8);
```

8 arguments are passed: Registers r0-r3 hold the first 4 arguments (a, b, c, d with values 1, 2, 3, 4). The last 4 arguments (h, i, j, k with values 5, 6, 7, 8) are pushed onto the stack by the caller before calling the function.

Caller

```
MOVS r0, #5
MOVS r1, #6
MOVS r2, #7
MOVS r3, #8
PUSH {r0, r1, r2, r3}
MOVS r0, #1
MOVS r1, #2
MOVS r2, #3
MOVS r3, #4
BL   sum
...
POP {r0, r1, r2, r3}
```

Callee

```
sum PROC
  EXPORT sum
  ADD r0, r0, r1    ; add a + b
  ADD r0, r0, r2    ; add c
  ADD r0, r0, r3    ; add d
  LDRD r1,r2, [sp]  ; r1=mem[sp],r2=mem[sp+4]
  ADD r0, r0, r1    ; add h
  ADD r0, r0, r2    ; add i
  LDRD r1,r2, [sp, #8] ; r1=mem[sp+8],r2=mem[sp+12]
  ADD r0, r0, r1    ; add j
  ADD r0, r0, r2    ; add k
  BX  LR
  ENDP
```

LDRD: Load Register Doubleword

Passing Extra Arguments via Stack

```
int32_t sum(int32_t a, int32_t b, int32_t c, int32_t d,
              int32_t h, int32_t i, int32_t j, int32_t k);
```

```
s = sum(1, 2, 3, 4, 5, 6, 7, 8);
```

Caller

```
MOVS r0, #5
MOVS r1, #6
MOVS r2, #7
MOVS r3, #8
PUSH {r0, r1, r2, r3}
MOVS r0, #1
MOVS r1, #2
MOVS r2, #3
MOVS r3, #4
BL   sum
...
POP {r0, r1, r2, r3}
```

Stack

Old SP	<xxxxxxxx>
SP+12	0x00000008
SP+8	0x00000007
SP+4	0x00000006
SP	0x00000005

Passing Extra Arguments via Stack

```
int32_t sum(int32_t a, int32_t b, int32_t c, int32_t d,
            int32_t h, int32_t i, int32_t j, int32_t k);
```

```
s = sum(1, 2, 3, 4, 5, 6, 7, 8);
```

In callee (subroutine) code, the subroutine saves registers r5, r6, and the link register LR via push at the start. It adds the first 4 arguments from registers r0-r3. It then loads the extra arguments from specific stack offsets with LDRD instructions (load register double), adding these to the result. The subroutine ends by popping saved registers and PC (program counter) to return.

Caller

```
MOVS r0, #5
MOVS r1, #6
MOVS r2, #7
MOVS r3, #8
PUSH {r0, r1, r2, r3}
MOVS r0, #1
MOVS r1, #2
MOVS r2, #3
MOVS r3, #4
BL   sum
...
POP {r0, r1, r2, r3}
```

Callee

```
sum PROC
  EXPORT sum
  PUSH {r5, r6, lr}
  ADD r0, r0, r1     ; add a + b
  ADD r0, r0, r2     ; add c
  ADD r0, r0, r3     ; add d
  LDRD r5,r6, [sp, #12] ;r5=mem[sp+12],r6=mem[sp+16]
  ADD r0, r0, r5     ; add h
  ADD r0, r0, r6     ; add i
  LDRD r5,r6, [sp, #20] ;r5=mem[sp+20],r6=mem[sp+24]
  ADD r0, r0, r5     ; add j
  ADD r0, r0, r6     ; add k
  POP {r5, r6, pc}
  ENDP
```

Passing Extra Arguments via Stack

```
int32_t sum(int32_t a, int32_t b, int32_t c, int32_t d,
            int32_t h, int32_t i, int32_t j, int32_t k);
```

```
s = sum(1, 2, 3, 4, 5, 6, 7, 8);
```

Stack

Old SP	<xxxxxxxx>
SP+24	0x00000008
SP+20	0x00000007
SP+16	0x00000006
SP+12	0x00000005
SP+8	lr
SP+4	r6
SP	r5

Callee

```
sum PROC
    EXPORT sum
    PUSH {r5, r6, lr}
    ADD r0, r0, r1    ; add a + b
    ADD r0, r0, r2    ; add c
    ADD r0, r0, r3    ; add d
    LDRD r5,r6, [sp, #12] ; r5=mem[sp+12],r6=mem[sp+16]
    ADD r0, r0, r5    ; add h
    ADD r0, r0, r6    ; add i
    LDRD r5,r6, [sp, #20] ; r5=mem[sp+20],r6=mem[sp+24]
    ADD r0, r0, r5    ; add j
    ADD r0, r0, r6    ; add k
    POP {r5, r6, pc}
    ENDP
```

Summary

- ARM Cortex-M uses full descending stack
- How to pass arguments into a subroutine?
  - Each 8-, 16- or 32-bit variables is passed via `r0`, `r1`, `r2`, `r3`
  - Extra parameters are passed via stack
- What registers should be preserved?
  - Caller-saved registers *vs* callee-saved registers
- How to preserve the running environment for the caller?
  - Via stack