

ARM Subroutines

Lecture 8

Yeongpil Cho

Hanyang University

Topics

- Subroutine Calls
- Preserve Environment via Stack
- Stack and Recursive Functions

Subroutines Calls

Subroutine

- A subroutine, 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.

Calling and exiting a Subroutine

BL *label*

- Step 1: LR = the next inst.
- Step 2: PC = label

BX *LR*

- PC = LR

• Notes:

- *label* is name of subroutine
- Compiler translates label to memory address
- After call, LR holds return address (the instruction following the call)
- MOVS PC, LR / POP {PC} also can be used for return

Caller Program

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

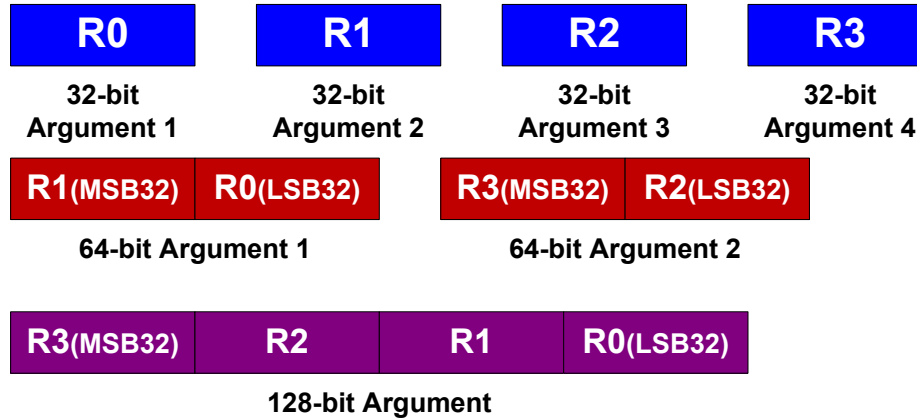
Subroutine/Callee

```
foo:
...
MOV    r4, #10
...
BX     LR
```

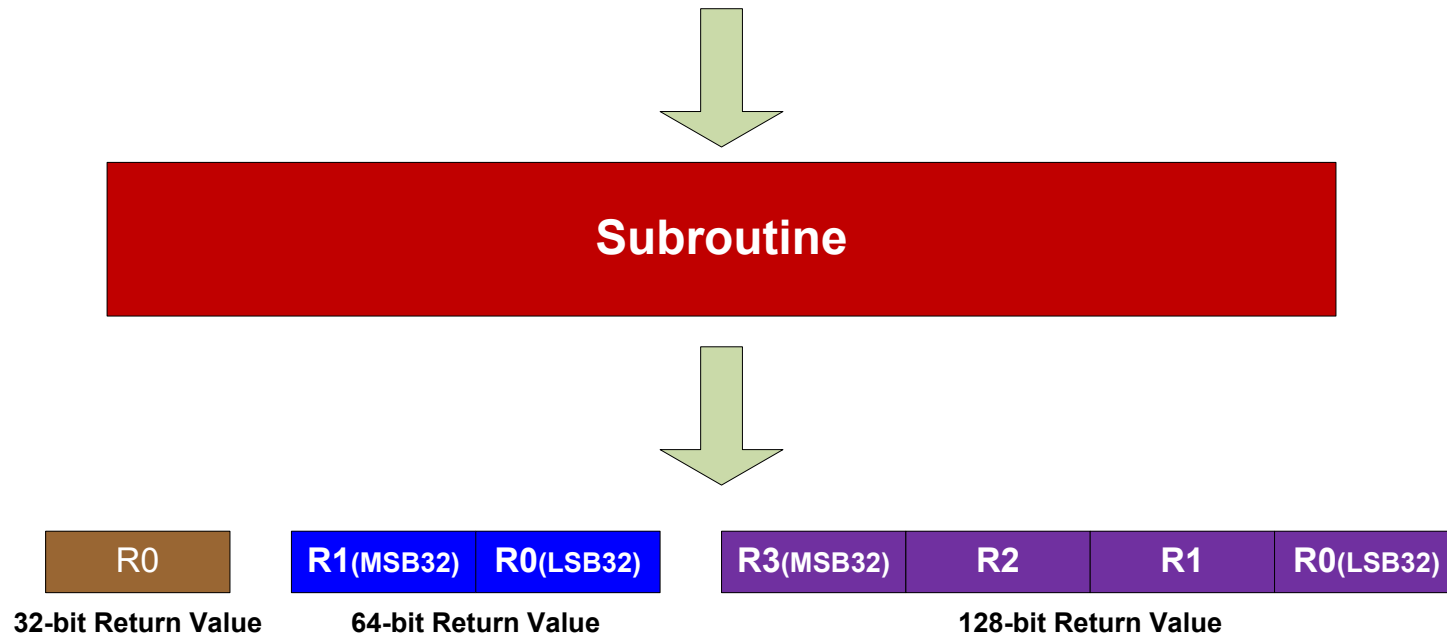
ARM Calling Convention

Register	Usage	Preservation	Notes
r0	Argument 1 and return value	Caller-saved	If return has 64 bits, then r0:r1 hold it. If argument 1 has 64 bits, r0:r1 hold it.
r1	Argument 2	Caller-saved	
r2	Argument 3	Caller-saved	If the return has 128 bits, r0-r3 hold it.
r3	Argument 4	Caller-saved	If more than 4 arguments, use the stack
r4	General-purpose V1	Callee-saved	Variable register 1 holds a local variable.
r5	General-purpose V2	Callee-saved	Variable register 2 holds a local variable.
r6	General-purpose V3	Callee-saved	Variable register 3 holds a local variable.
r7	General-purpose V4	Callee-saved	Variable register 4 holds a local variable.
r8	General-purpose V5	Callee-saved	Variable register 5 holds a local variable.
r9	Platform specific/V6	Caller-saved	Usage is platform-dependent.
r10	General-purpose V7	Callee-saved	Variable register 7 holds a local variable.
r11	General-purpose V8	Callee-saved	Variable register 8 holds a local variable.
r12 (IP)	Intra-procedure-call register	Caller-saved	It holds intermediate values between a procedure and the sub-procedure it calls.
r13 (SP)	Stack pointer	Callee-saved	SP has to be the same after a subroutine has completed
r14 (LR)	Link register	Caller-saved	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

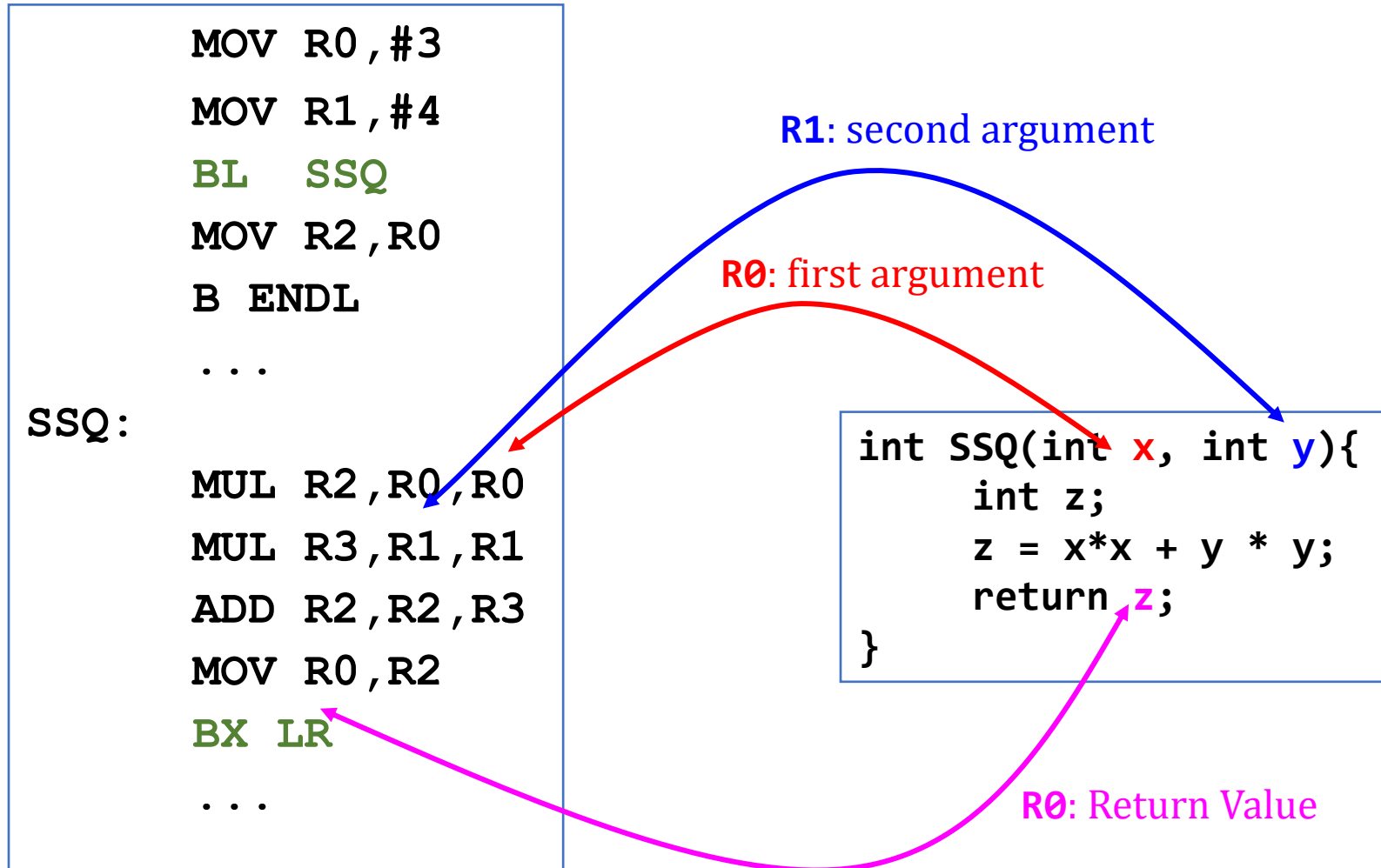
Passing Arguments via Registers



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.



Example: $R2 = R0 * R0 + R1 * R1$



Example: $R2 = R0 * R0 + R1 * R1$

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

R0	
R1	
R2	
R3	
LR	
PC	0x08000128

SSQ

In fact, PC is 0x08000129 because bit 0 of PC should always be 1 for ARM Cortex-M to indicate thumb mode.

	Memory Address
MOV R0,#3	0x08000128
MOV R1,#4	0x0800012C
BL SSQ	0x08000130
MOV R2,R0	0x08000134
B ENDL	0x08000136
MUL R2,...	0x0800013A
MUL R3,...	0x0800013C
ADD R2,R3	0x08000140
MOV R0,R2	0x08000142
BX LR	0x08000144

Example: $R2 = R0 * R0 + R1 * R1$

MOV R0, #3

MOV R1, #4

BL SSQ

MOV R2, R0

B ENDL

...

SSQ:

MUL R2, R0, R0

MUL R3, R1, R1

ADD R2, R2, R3

MOV R0, R2

BX LR

ENDL: ...

R0	3
R1	
R2	
R3	
LR	
PC	0x08000128

Memory Address	
MOV R0, #3	0x08000128
MOV R1, #4	0x0800012C
BL SSQ	0x08000130
MOV R2, R0	0x08000134
B ENDL	0x08000136
MUL R2, ...	0x0800013A
MUL R3, ...	0x0800013C
ADD R2, R3	0x08000140
MOV R0, R2	0x08000142
BX LR	0x08000144

Example: $R2 = R0 * R0 + R1 * R1$

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

R0	3
R1	4
R2	
R3	
LR	
PC	0x0800012C

	Memory Address
MOV R0,#3	0x08000128
MOV R1,#4	0x0800012C
BL SSQ	0x08000130
MOV R2,R0	0x08000134
B ENDL	0x08000136
MUL R2,...	0x0800013A
MUL R3,...	0x0800013C
ADD R2,R3	0x08000140
MOV R0,R2	0x08000142
BX LR	0x08000144

SSQ

Example: $R2 = R0 * R0 + R1 * R1$

MOV R0,#3

MOV R1,#4

BL SSQ

MOV R2,R0

B ENDL

...

SSQ:

MUL R2,R0,R0

MUL R3,R1,R1

ADD R2,R2,R3

MOV R0,R2

BX LR

ENDL: ...

R0	3
R1	4
R2	
R3	

LR	
PC	0x08000130

	Memory Address
MOV R0,#3	0x08000128
MOV R1,#4	0x0800012C
BL SSQ	0x08000130
MOV R2,R0	0x08000134
B ENDL	0x08000136
MUL R2,...	0x0800013A
MUL R3,...	0x0800013C
ADD R2,R3	0x08000140
MOV R0,R2	0x08000142
BX LR	0x08000144

SSQ

Example: $R2 = R0 * R0 + R1 * R1$

```

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

R0	3
R1	4
R2	
R3	

LR	0x08000134
PC	0x0800013B

In fact, LR is 0x08000135 because bit 0 of PC should always be 1 for ARM Cortex-M to indicate thumb mode.

	Memory Address
MOV R0,#3	0x08000128
MOV R1,#4	0x0800012C
BL SSQ	0x08000130
MOV R2,R0	0x08000134
B ENDL	0x08000136
MUL R2,...	0x0800013A
MUL R3,...	0x0800013C
ADD R2,R3	0x08000140
MOV R0,R2	0x08000142
BX LR	0x08000144

Address of the next instruction after the branch is saved into

Example: $R2 = R0 * R0 + R1 * R1$

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

R0	3
R1	4
R2	9
R3	

LR	0x08000134
PC	0x0800013B

	Memory Address
MOV R0,#3	0x08000128
MOV R1,#4	0x0800012C
BL SSQ	0x08000130
MOV R2,R0	0x08000134
B ENDL	0x08000136
MUL R2,...	0x0800013A
MUL R3,...	0x0800013C
ADD R2,R3	0x08000140
MOV R0,R2	0x08000142
BX LR	0x08000144

SSQ

Example: $R2 = R0 * R0 + R1 * R1$

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

R0	3
R1	4
R2	9
R3	16

LR	0x08000134
PC	0x0800013C

	Memory Address
MOV R0,#3	0x08000128
MOV R1,#4	0x0800012C
BL SSQ	0x08000130
MOV R2,R0	0x08000134
B ENDL	0x08000136
MUL R2,...	0x0800013A
MUL R3,...	0x0800013C
ADD R2,R3	0x08000140
MOV R0,R2	0x08000142
BX LR	0x08000144

SSQ

Example: $R2 = R0 * R0 + R1 * R1$

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

R0	3
R1	4
R2	25
R3	16

LR	0x08000134
PC	0x08000140

	Memory Address
MOV R0,#3	0x08000128
MOV R1,#4	0x0800012C
BL SSQ	0x08000130
MOV R2,R0	0x08000134
B ENDL	0x08000136
MUL R2,...	0x0800013A
MUL R3,...	0x0800013C
ADD R2,R3	0x08000140
MOV R0,R2	0x08000142
BX LR	0x08000144

SSQ

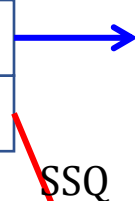
Example: $R2 = R0 * R0 + R1 * R1$

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

R0	25
R1	4
R2	25
R3	16

LR	0x08000134
PC	0x08000142

	Memory Address
MOV R0,#3	0x08000128
MOV R1,#4	0x0800012C
BL SSQ	0x08000130
MOV R2,R0	0x08000134
B ENDL	0x08000136
MUL R2,...	0x0800013A
MUL R3,...	0x0800013C
ADD R2,R3	0x08000140
MOV R0,R2	0x08000142
BX LR	0x08000144



Example: $R2 = R0 * R0 + R1 * R1$

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

R0	25
R1	4
R2	25
R3	16

LR	0x08000134
PC	0x08000144

	Memory Address
MOV R0,#3	0x08000128
MOV R1,#4	0x0800012C
BL SSQ	0x08000130
MOV R2,R0	0x08000134
B ENDL	0x08000136
MUL R2,...	0x0800013A
MUL R3,...	0x0800013C
ADD R2,R3	0x08000140
MOV R0,R2	0x08000142
BX LR	0x08000144



Example: $R2 = R0 * R0 + R1 * R1$

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

R0	25
R1	4
R2	25
R3	16

LR	0x08000134
PC	0x08000134

SSQ

	Memory Address
MOV R0,#3	0x08000128
MOV R1,#4	0x0800012C
BL SSQ	0x08000130
MOV R2,R0	0x08000134
B ENDL	0x08000136
MUL R2,...	0x0800013A
MUL R3,...	0x0800013C
ADD R2,R3	0x08000140
MOV R0,R2	0x08000142
BX LR	0x08000144

Copy LR to PC when returning from a subroutine!

Example: $R2 = R0 * R0 + R1 * R1$

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

R0	25
R1	4
R2	25
R3	16

LR	0x08000134
PC	0x08000134

	Memory Address
MOV R0,#3	0x08000128
MOV R1,#4	0x0800012C
BL SSQ	0x08000130
MOV R2,R0	0x08000134
B ENDL	0x08000136
MUL R2,...	0x0800013A
MUL R3,...	0x0800013C
ADD R2,R3	0x08000140
MOV R0,R2	0x08000142
BX LR	0x08000144

SSQ

Example: $R2 = R0 * R0 + R1 * R1$

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

R0	25
R1	4
R2	25
R3	16

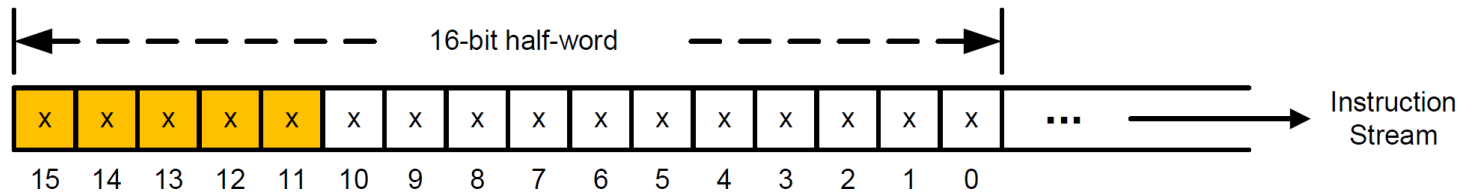
LR	0x08000134
PC	0x08000136

	Memory Address
MOV R0,#3	0x08000128
MOV R1,#4	0x0800012C
BL SSQ	0x08000130
MOV R2,R0	0x08000134
B ENDL	0x08000136
MUL R2,...	0x0800013A
MUL R3,...	0x0800013C
ADD R2,R3	0x08000140
MOV R0,R2	0x08000142
BX LR	0x08000144

SSQ

Realities

- In the previous example, PC is incremented by 2 or 4.
- but, PC is always incremented by **4**.
 - Each time, 4 bytes are fetched from the instruction memory
 - It is either two 16-bit instructions or one 32-bit instruction

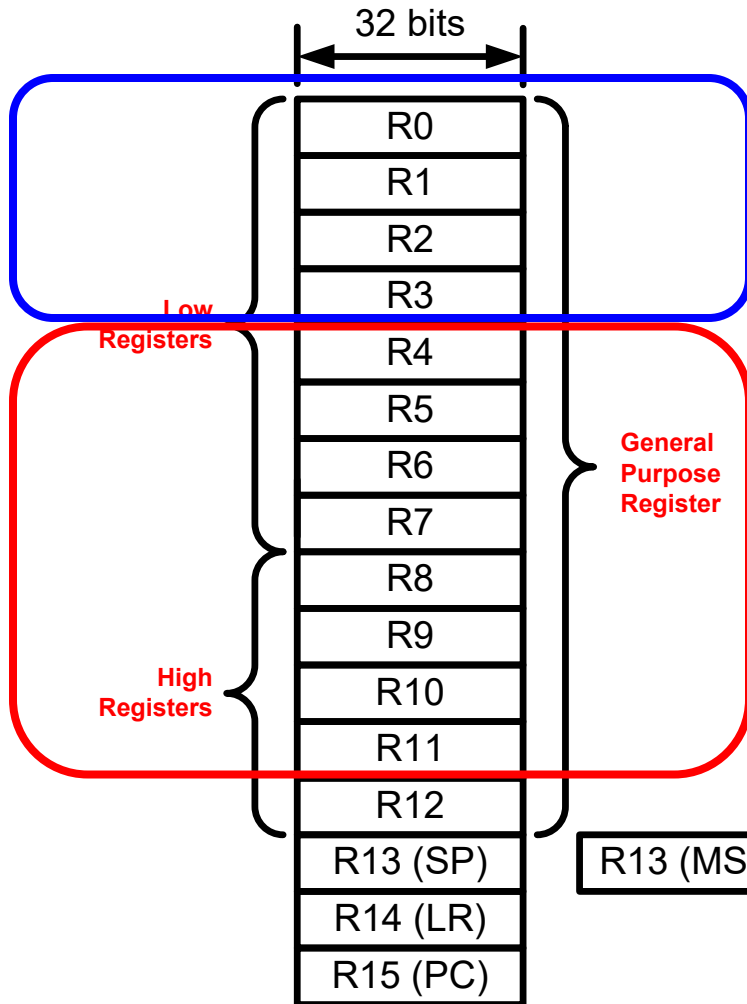


If bit [15-11] = **11101**, **11110**, or **11111**, then, it is the first half-word of a 32-bit instruction. Otherwise, it is a 16-bit instruction.

ARM Calling Convention

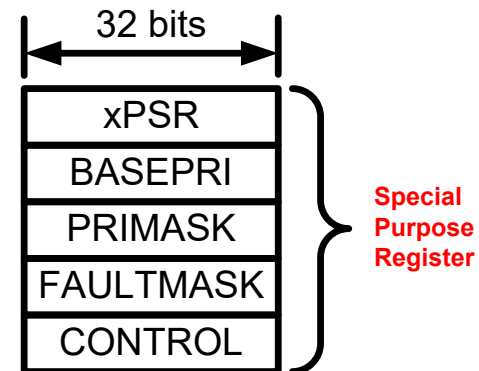
Register	Usage	Preservation	Notes
r0	Argument 1 and return value	Caller-saved	If return has 64 bits, then r0:r1 hold it. If argument 1 has 64 bits, r0:r1 hold it.
r1	Argument 2	Caller-saved	
r2	Argument 3	Caller-saved	If the return has 128 bits, r0-r3 hold it.
r3	Argument 4	Caller-saved	If more than 4 arguments, use the stack
r4	General-purpose V1	Callee-saved	Variable register 1 holds a local variable.
r5	General-purpose V2	Callee-saved	Variable register 2 holds a local variable.
r6	General-purpose V3	Callee-saved	Variable register 3 holds a local variable.
r7	General-purpose V4	Callee-saved	Variable register 4 holds a local variable.
r8	General-purpose V5	Callee-saved	Variable register 5 holds a local variable.
r9	Platform specific/V6	Caller-saved	Usage is platform-dependent.
r10	General-purpose V7	Callee-saved	Variable register 7 holds a local variable.
r11	General-purpose V8	Callee-saved	Variable register 8 holds a local variable.
r12 (IP)	Intra-procedure-call register	Caller-saved	It holds intermediate values between a procedure and the sub-procedure it calls.
r13 (SP)	Stack pointer	Callee-saved	SP has to be the same after a subroutine has completed
r14 (LR)	Link register	Caller-saved	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 // Callee Saved registers



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

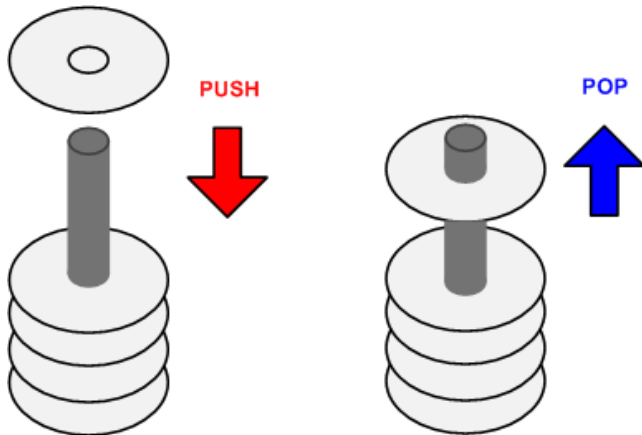
Callee must save them. Caller expects these values are retained.



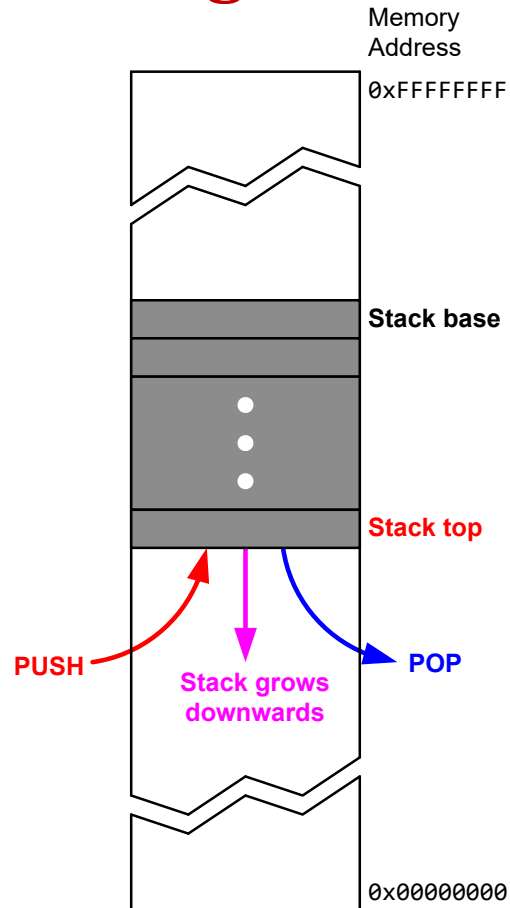
Preserve Environment via Stack

Stack

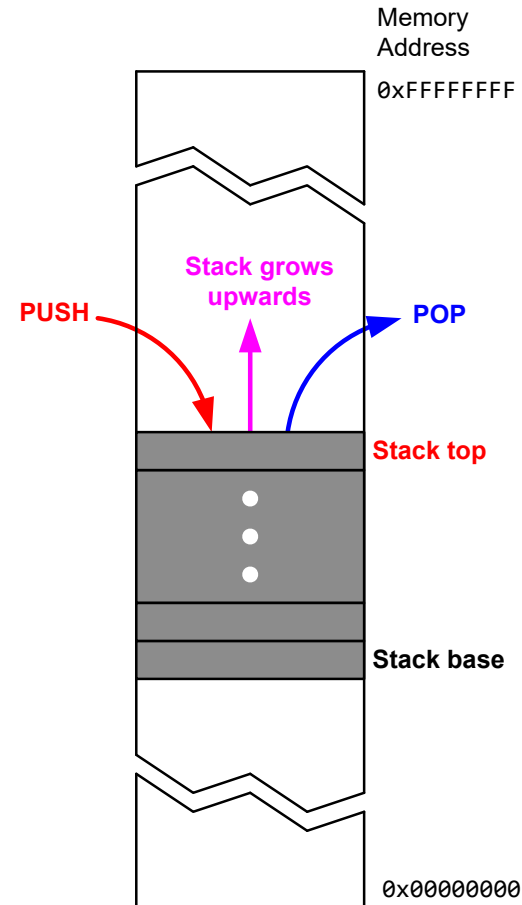
- A **Last-In-First-Out** data structure
- Only allow to access the most recently added item
 - Also called the top of the stack
- Key operations:
 - push (add item to stack)
 - pop (remove top item from stack)



Stack Growth Convention: Ascending vs Descending

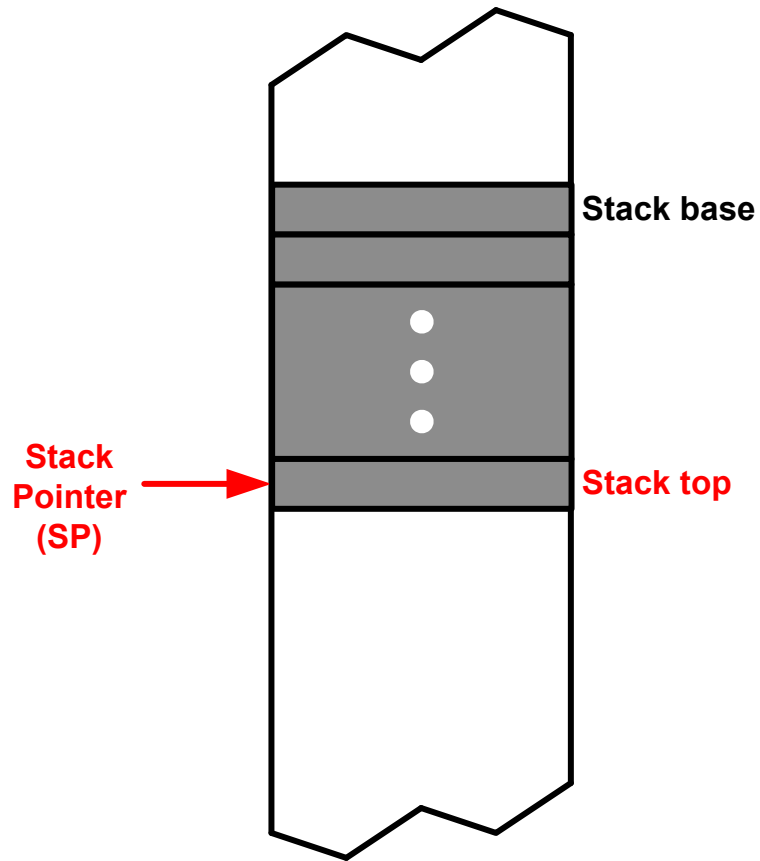


Descending stack: Stack grows towards low memory address

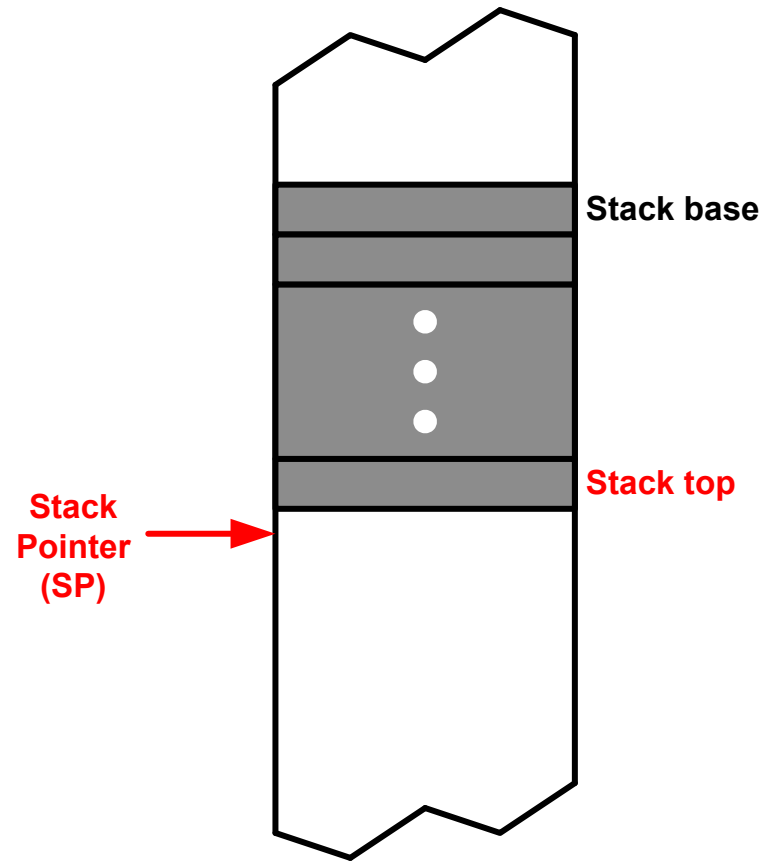


Ascending stack: Stack grows towards high memory address

Stack Growth Convention: Full vs Empty



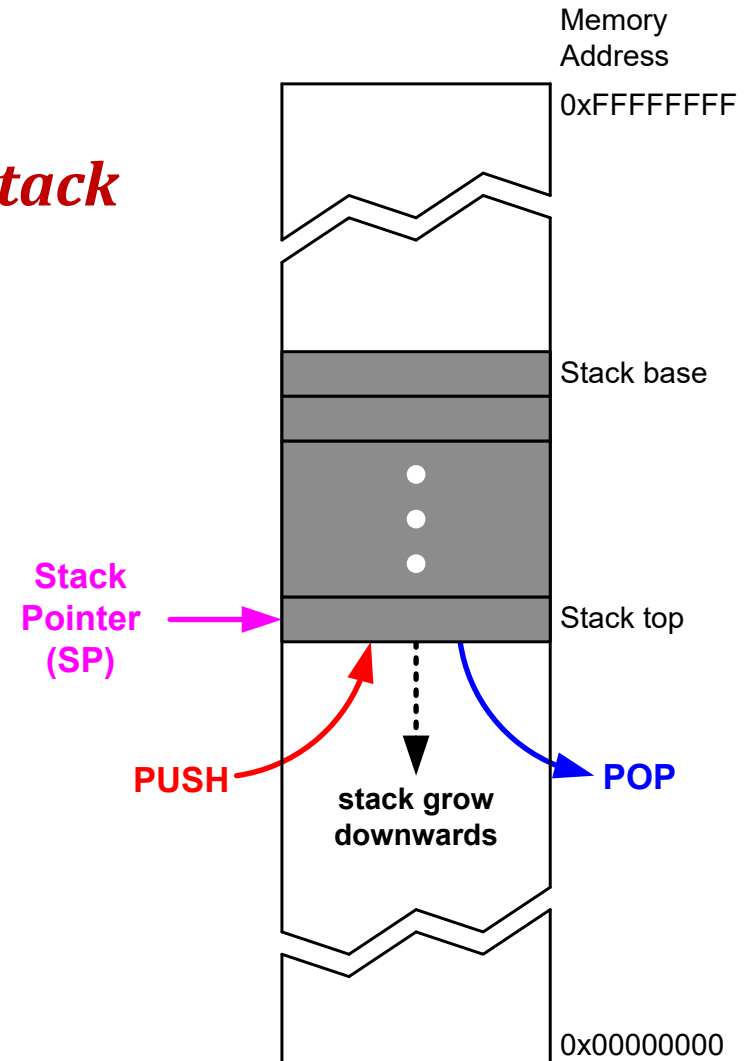
Full stack: SP points to the last item pushed onto the stack



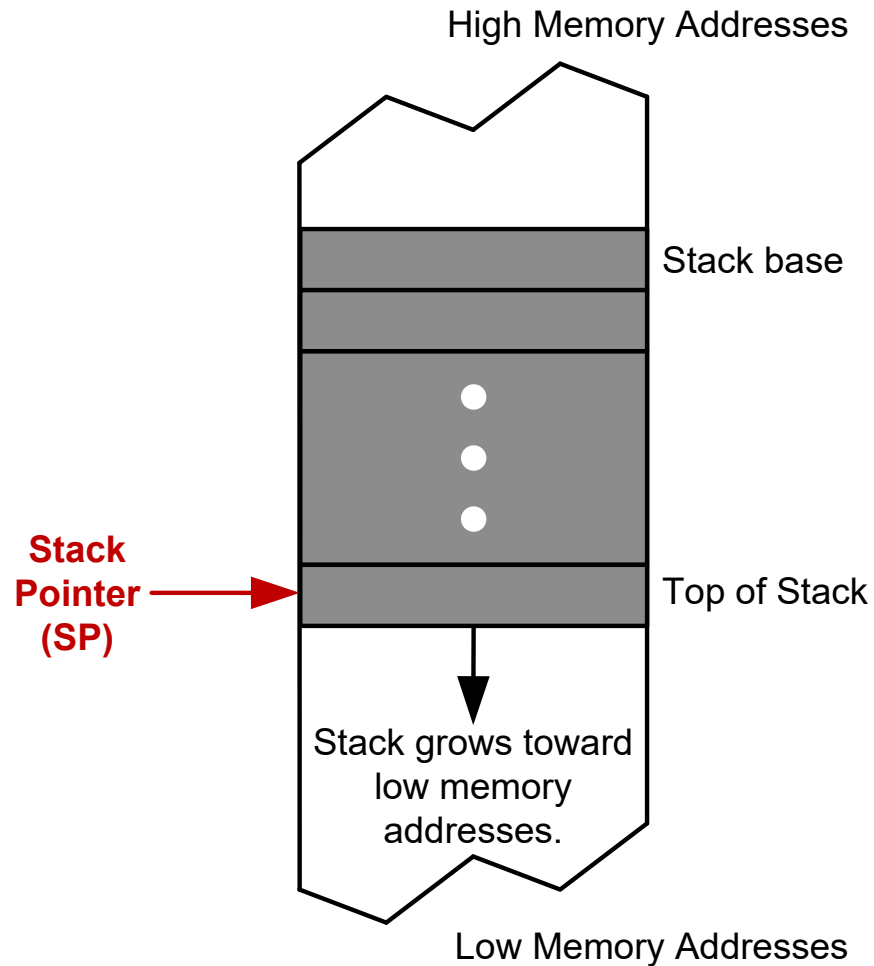
Empty stack: SP points to the next free space on the stack

Cortex-M Stack

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



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

PUSH {*Rd*}

- $SP = SP - 4 \rightarrow$ descending stack
- $(*SP) = Rd \rightarrow$ 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}

- $Rd = (*SP) \rightarrow$ full stack
- $SP = SP + 4 \rightarrow$ Stack shrinks

Pop multiple registers

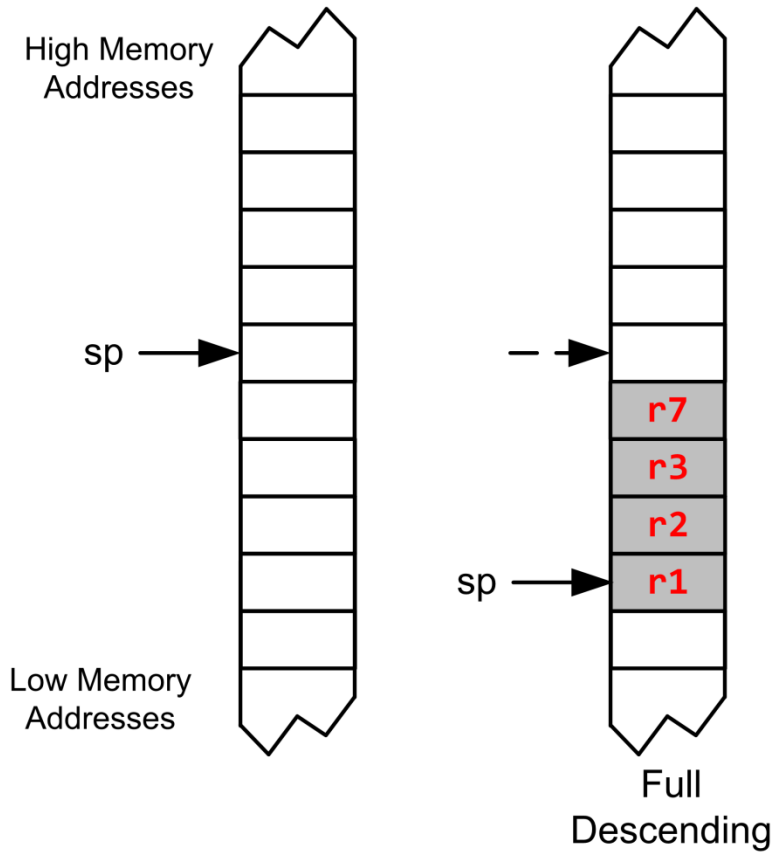
They are equivalent.

POP {r6, r7, r8} $\longleftrightarrow$ POP {r8, r7, r6} $\longleftrightarrow$
 POP {r6}
 POP {r7}
 POP {r8}

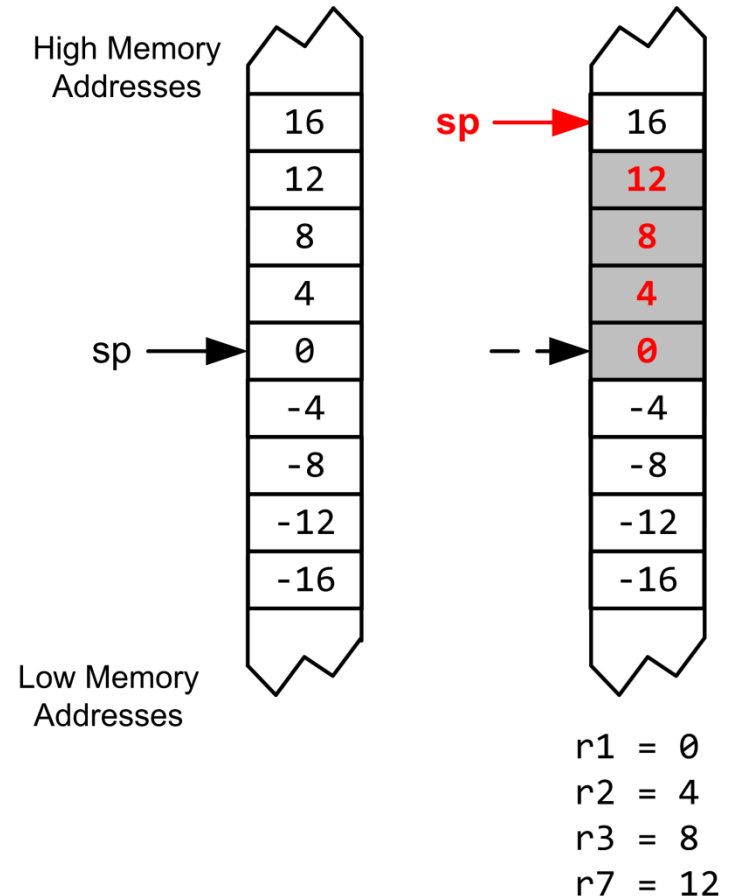
- The order in which registers listed in the register list does not matter.
- When popping multiple registers, these registers are automatically **sorted by name** and **the lowest-numbered register** is loaded from the lowest memory address, *i.e.* **is loaded first**.

Full Descending Stack

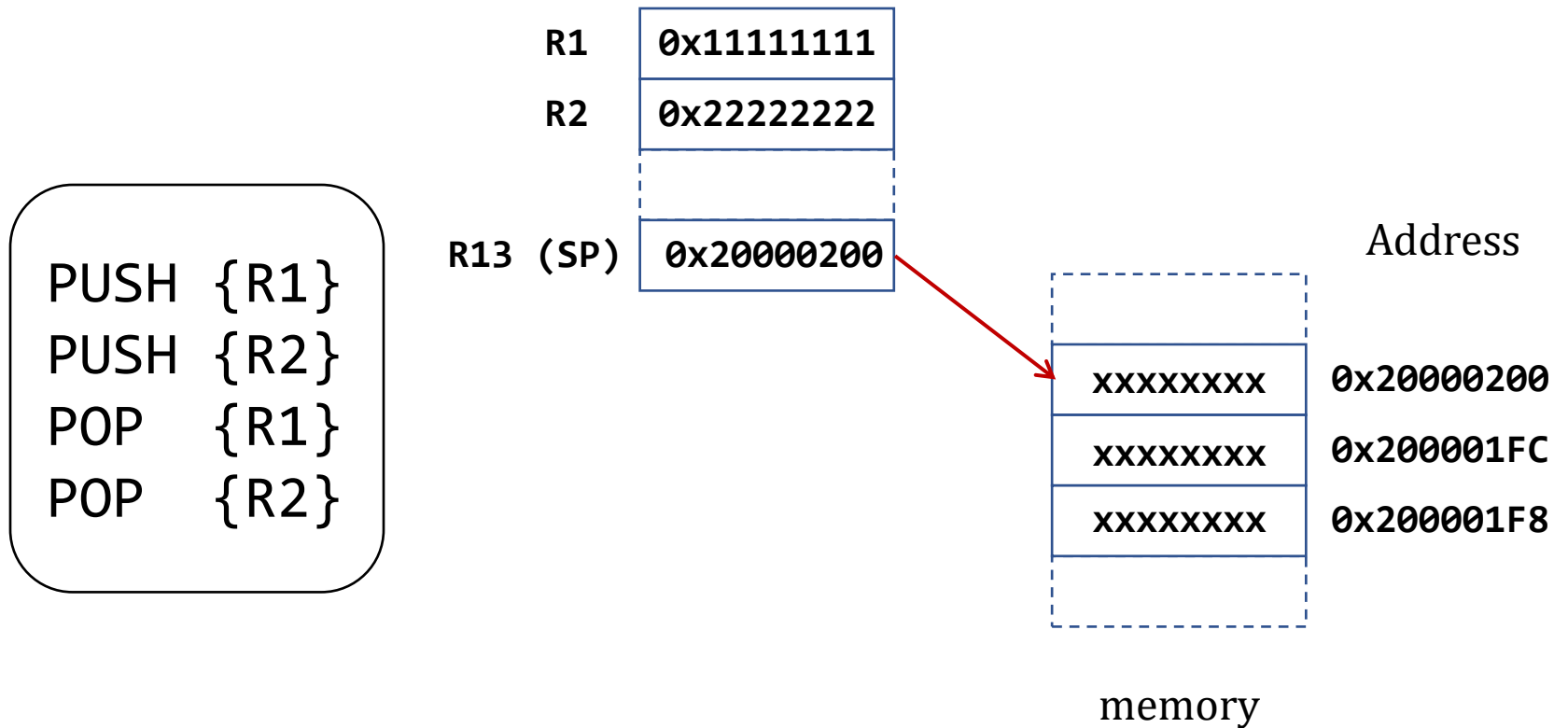
PUSH {r3, r1, r7, r2}



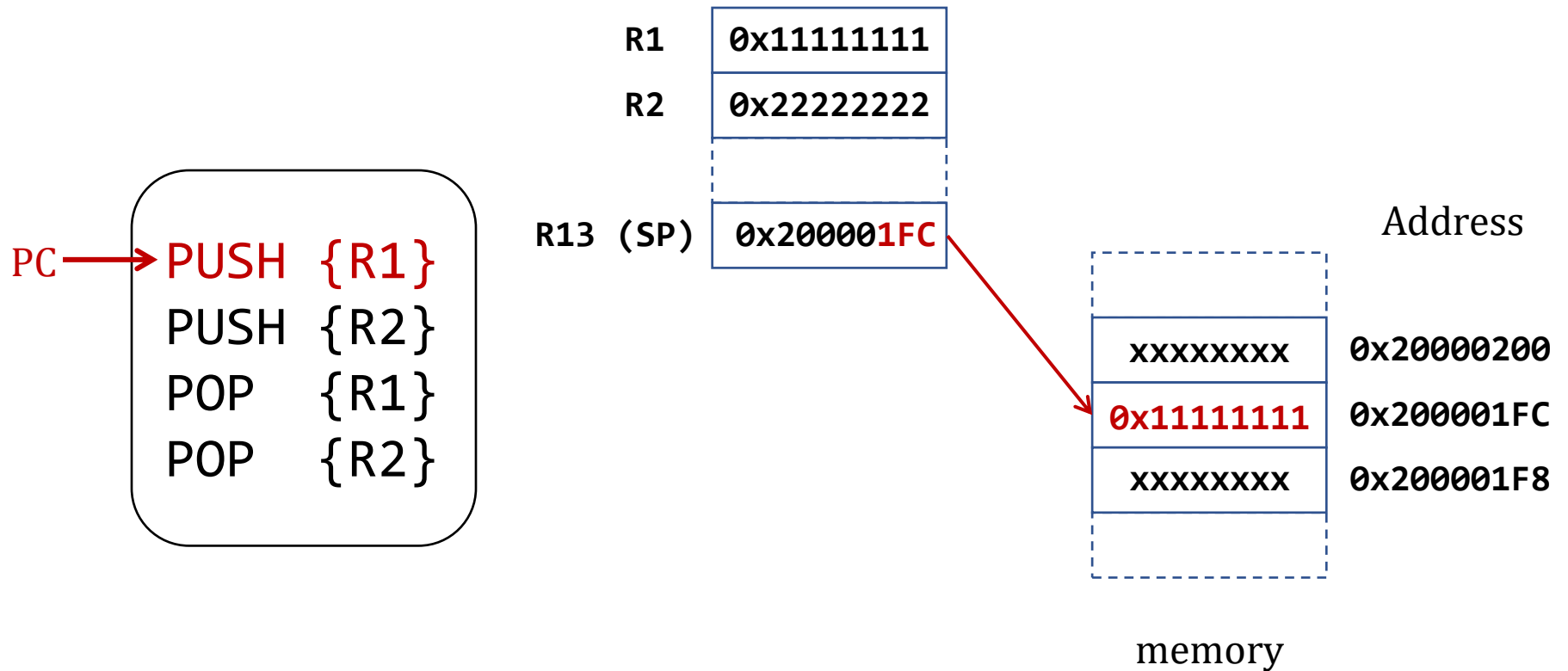
POP {r3, r1, r7, r2}



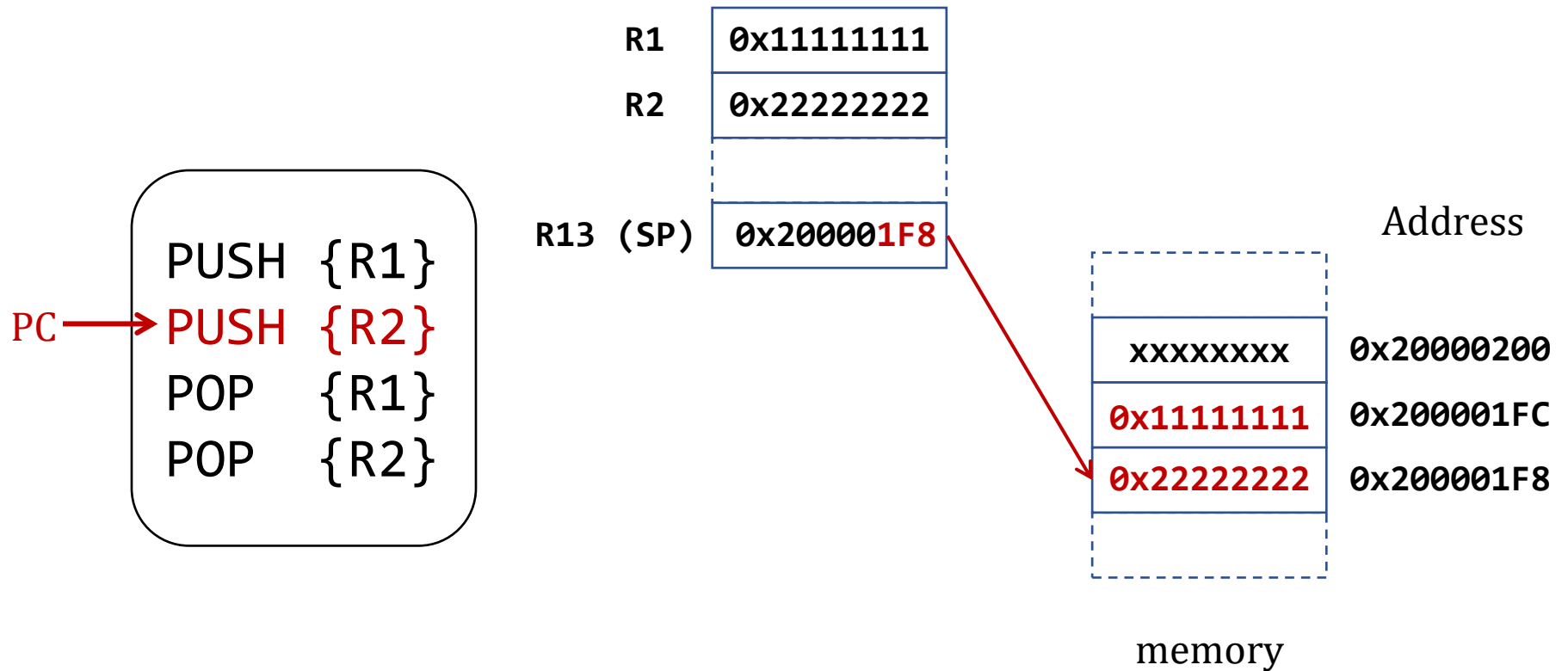
Example: swap R1 & R2



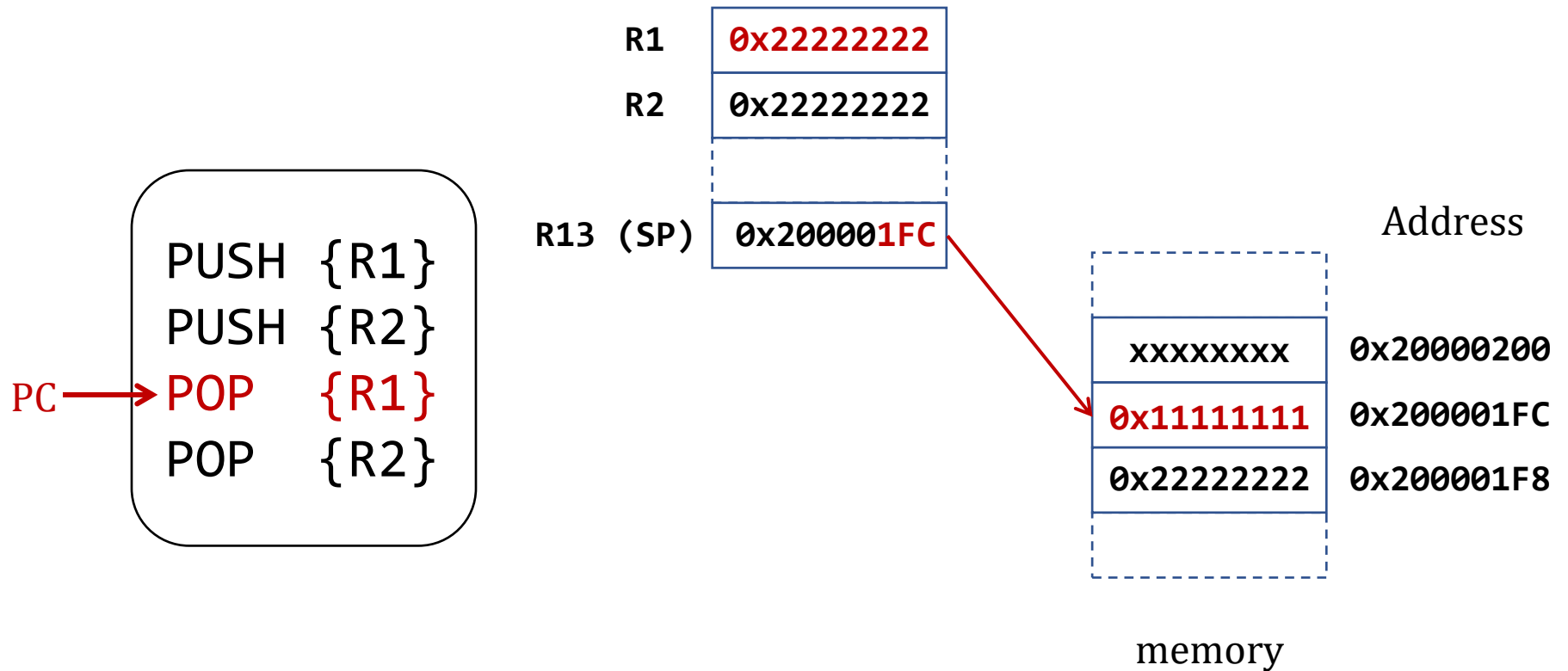
Example: swap R1 & R2



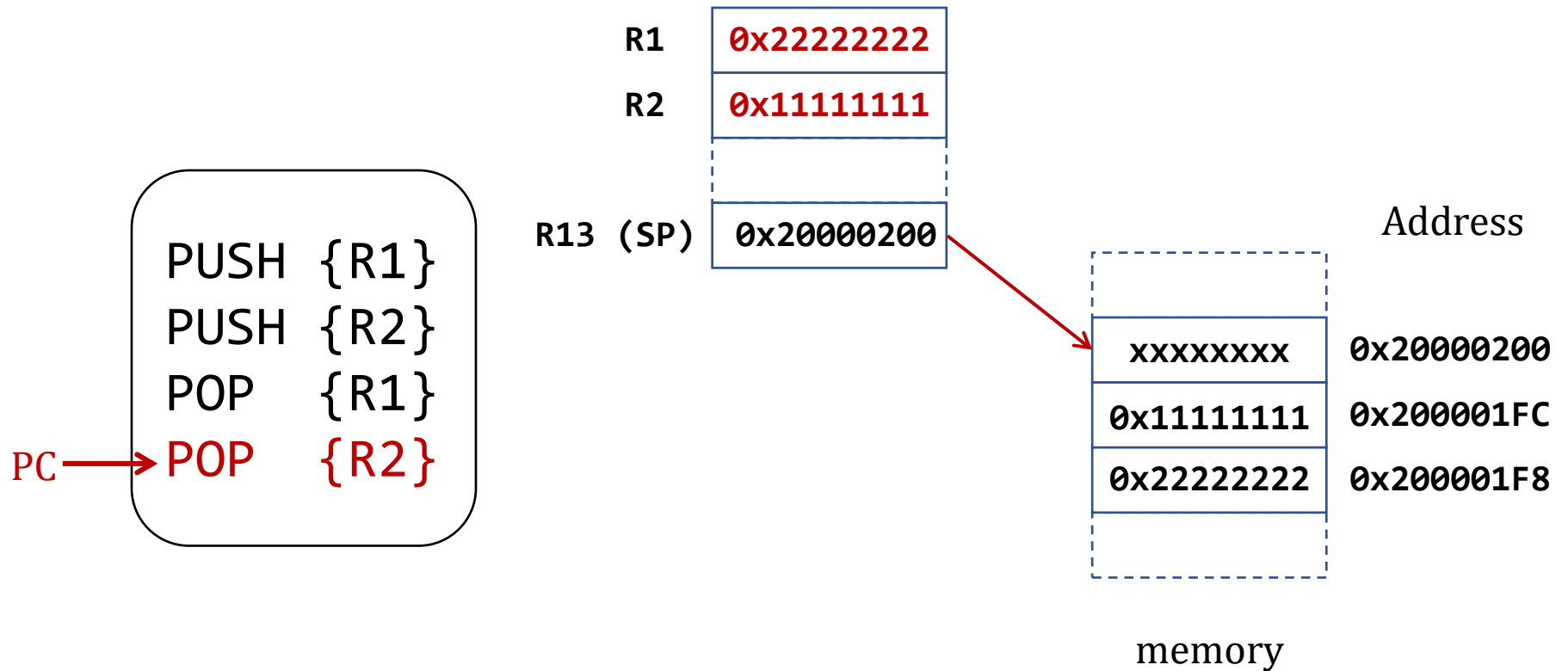
Example: swap R1 & R2



Example: swap R1 & R2



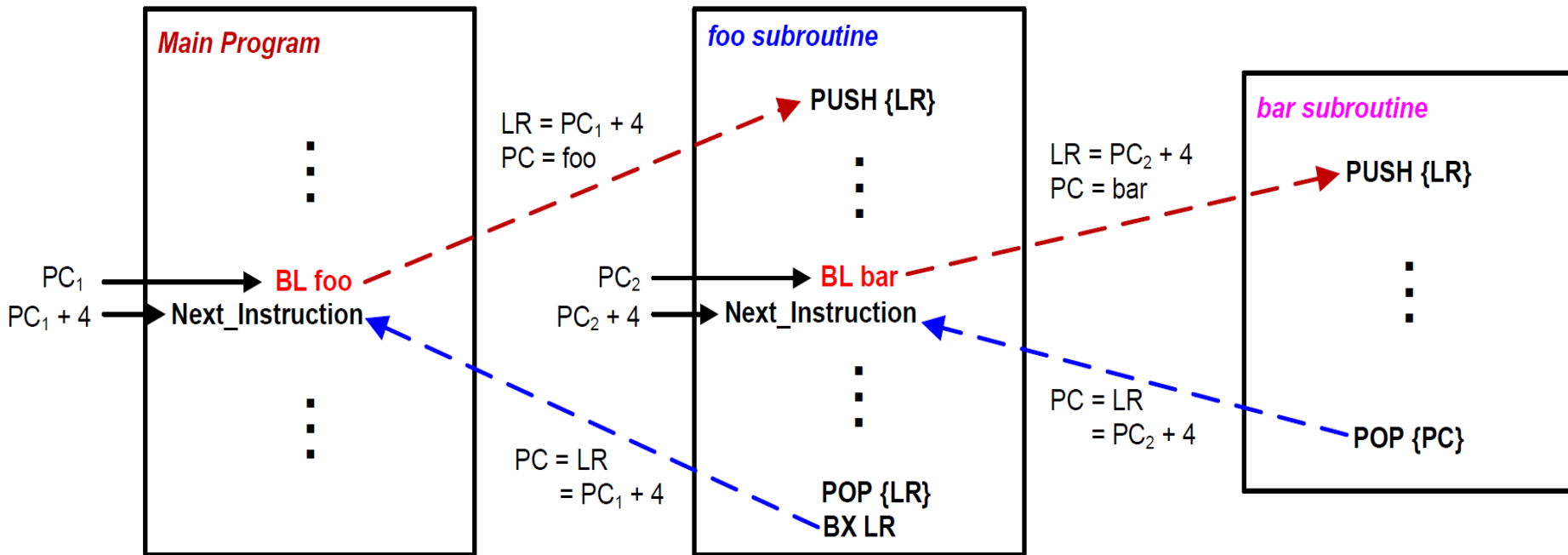
Example: swap R1 & R2



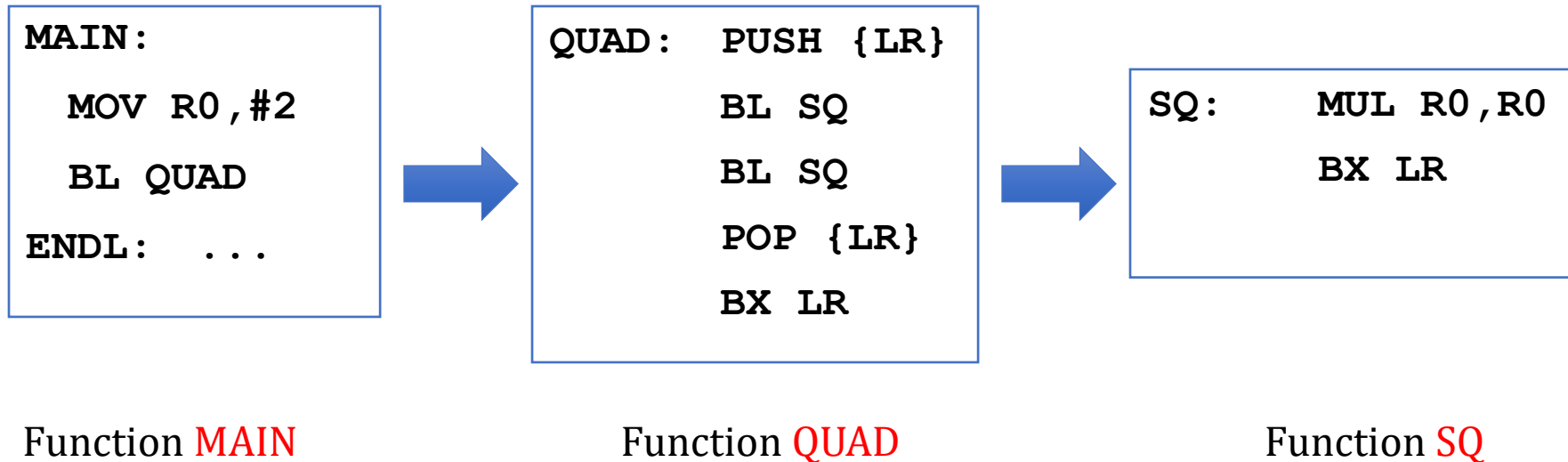
Preserve Runtime Environment via Stack

Caller Program	Subroutine/Callee
<pre>MOV r4, #100 ... BL foo ... ADD r4, r4, #1 ; r4 = 101, not 11</pre>	<pre>foo: PUSH {r4} ; preserve r4 ... MOV r4, #10 ; foo changes r4 ... POP {r4} ; Recover r4 BX LR</pre>

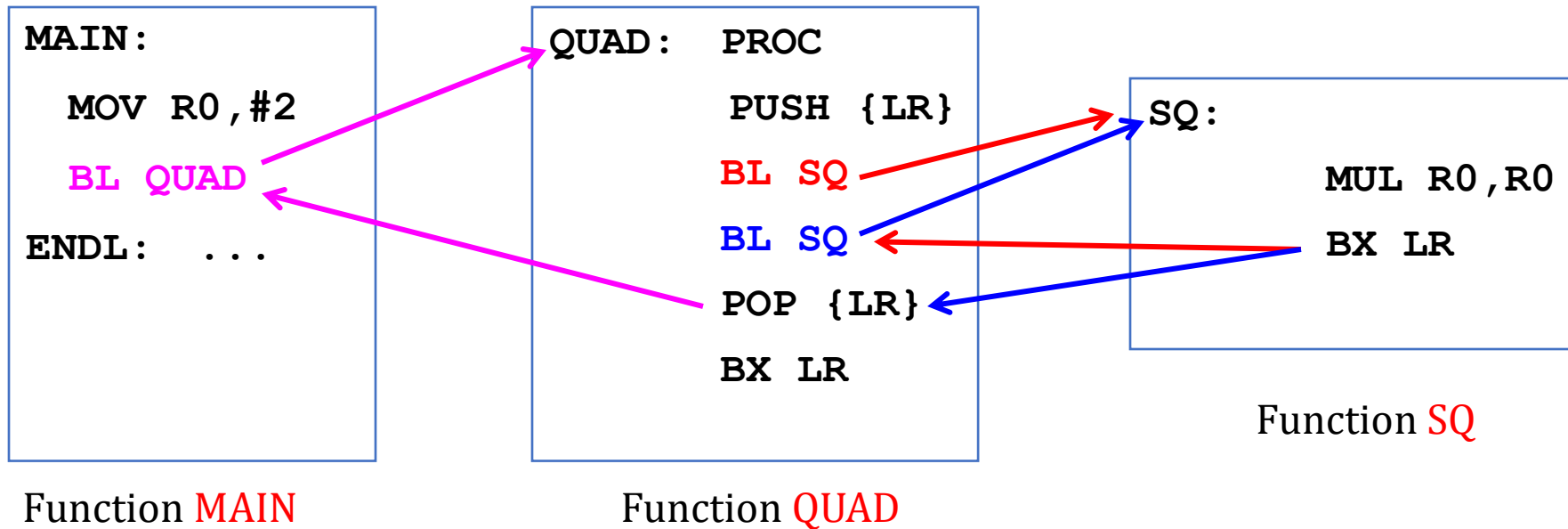
Stacks and Subroutines



Subroutine Calling Another Subroutine



Subroutine Calling Another Subroutine



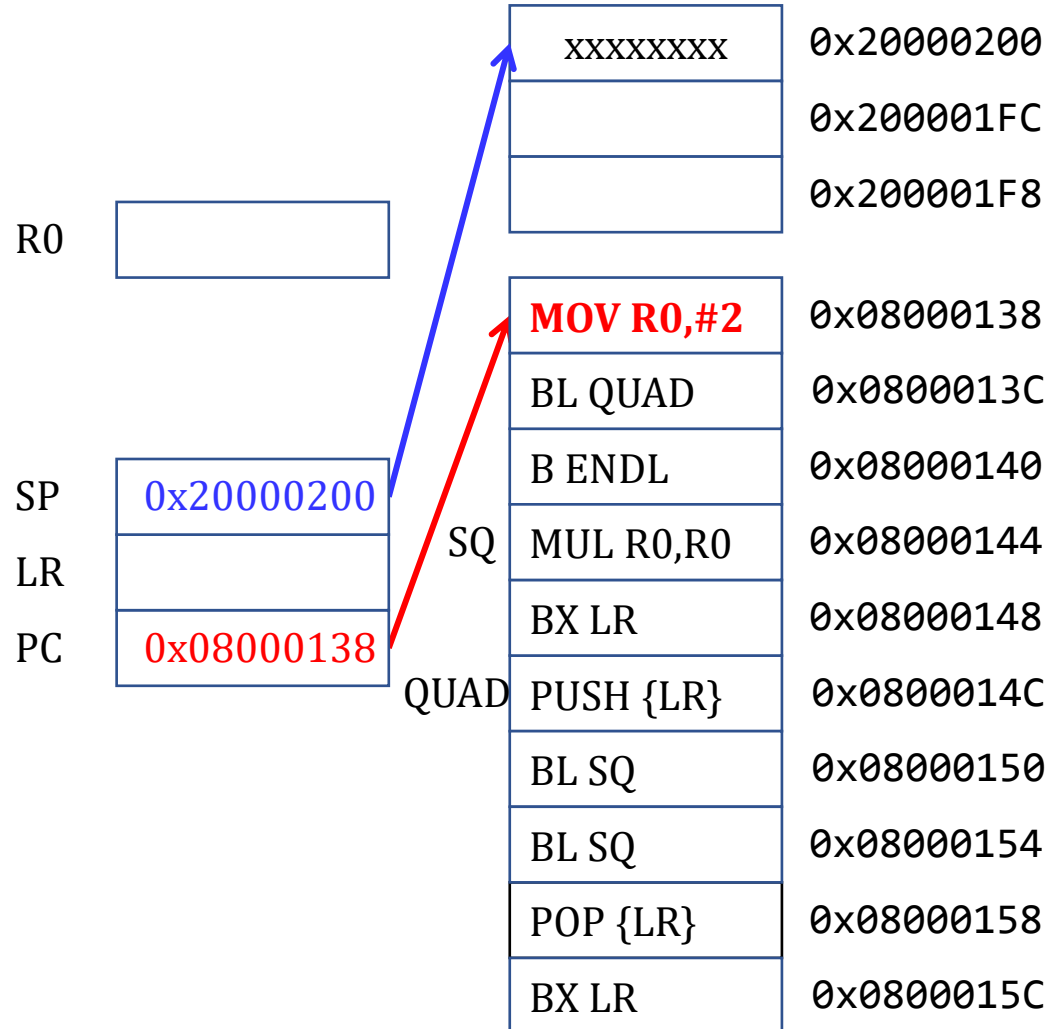
Example: R0 = R0⁴

```
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:  ...
```



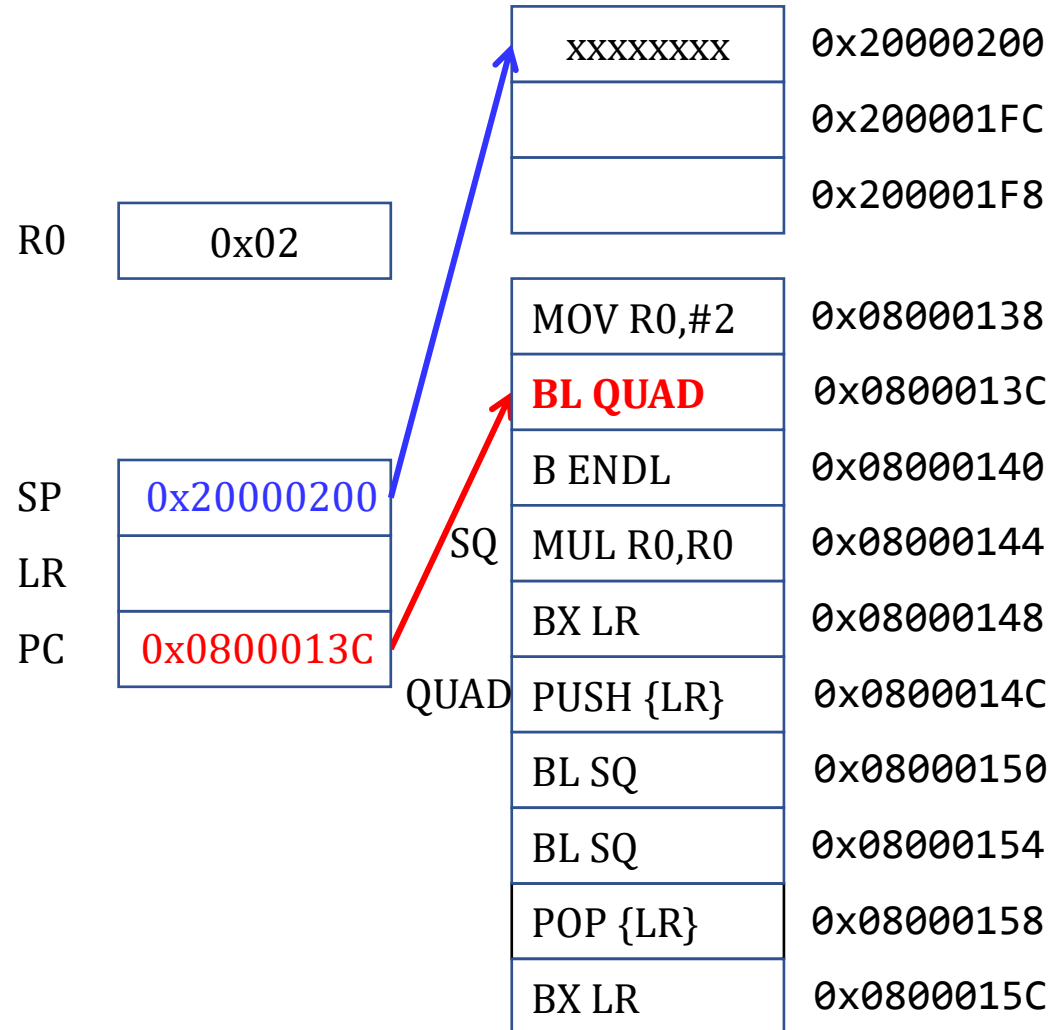
Example: R0 = R0⁴

```
        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:    ...
```



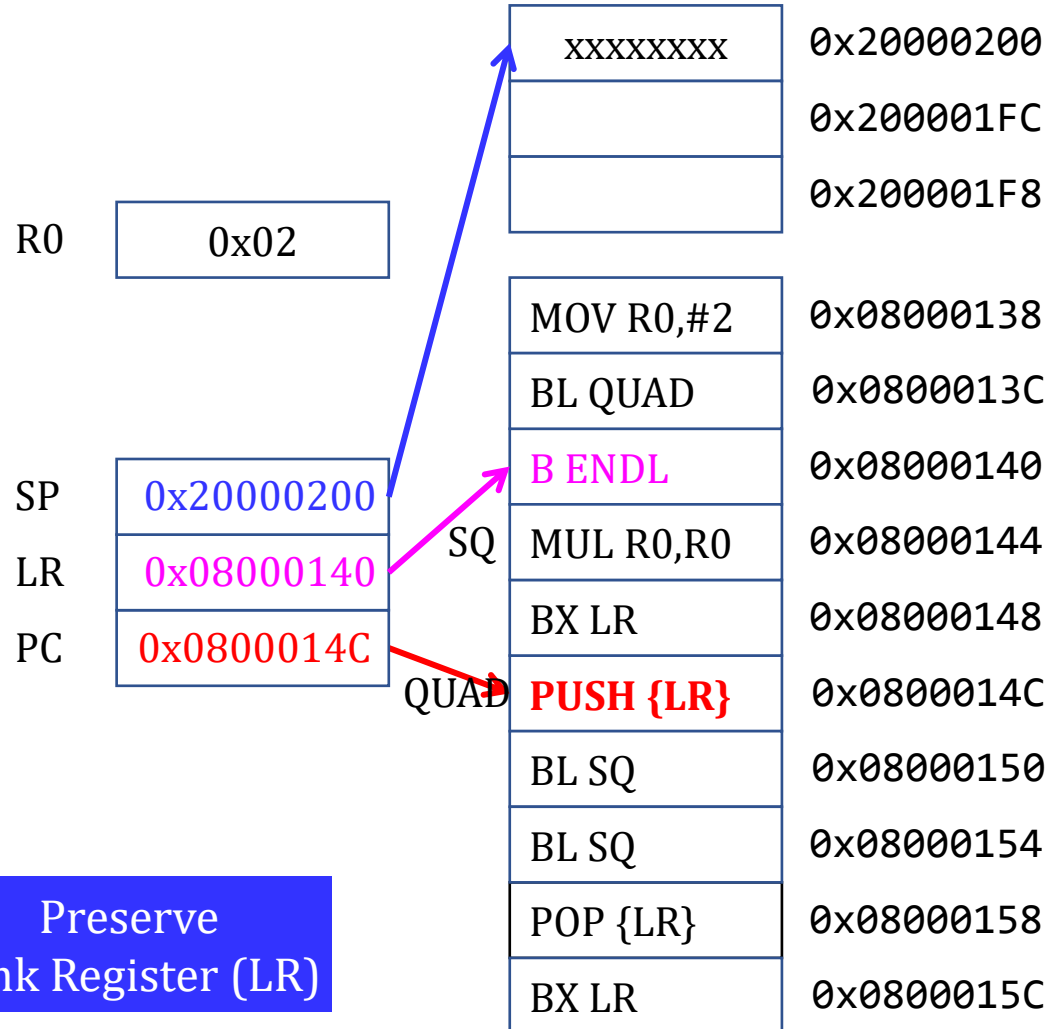
Example: R0 = R0⁴

```
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:  ...
```



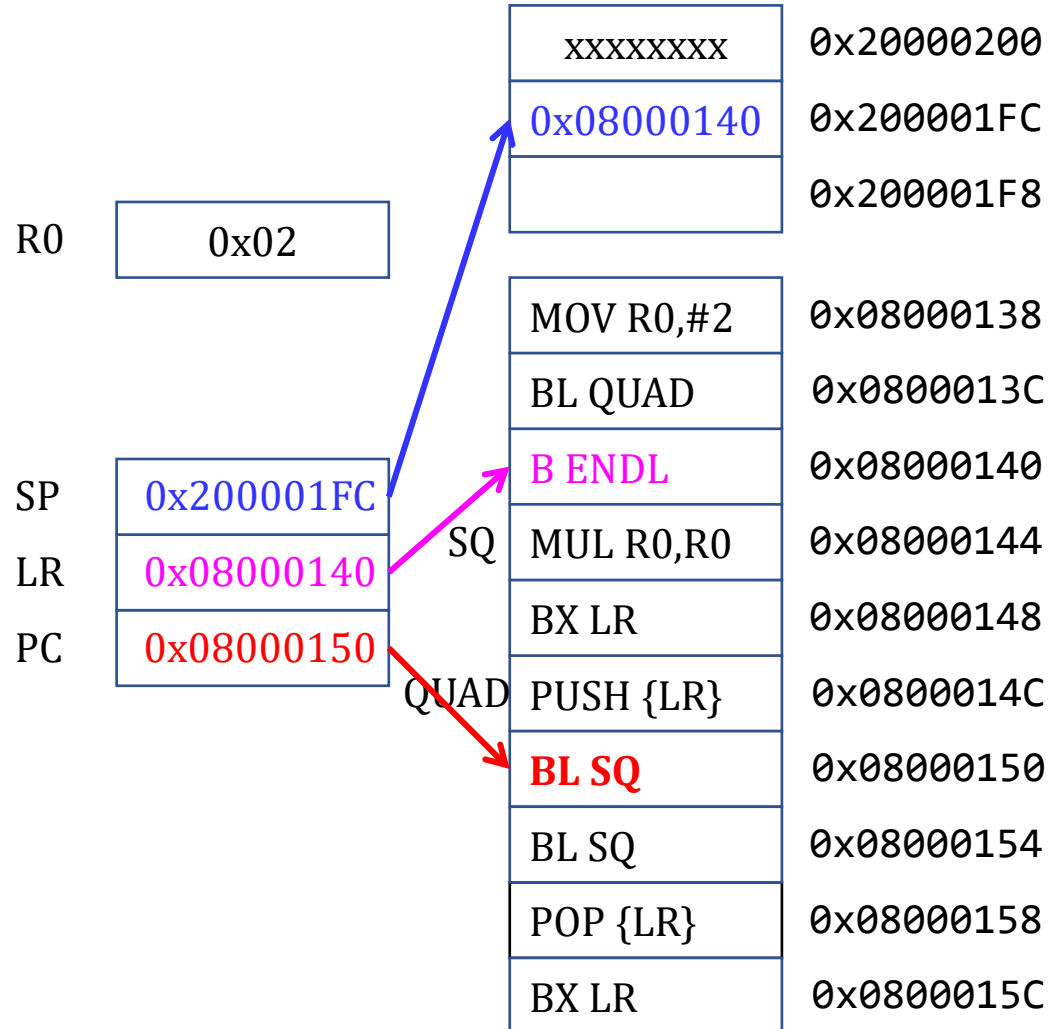
Example: R0 = R0⁴

```
        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:    ...
```



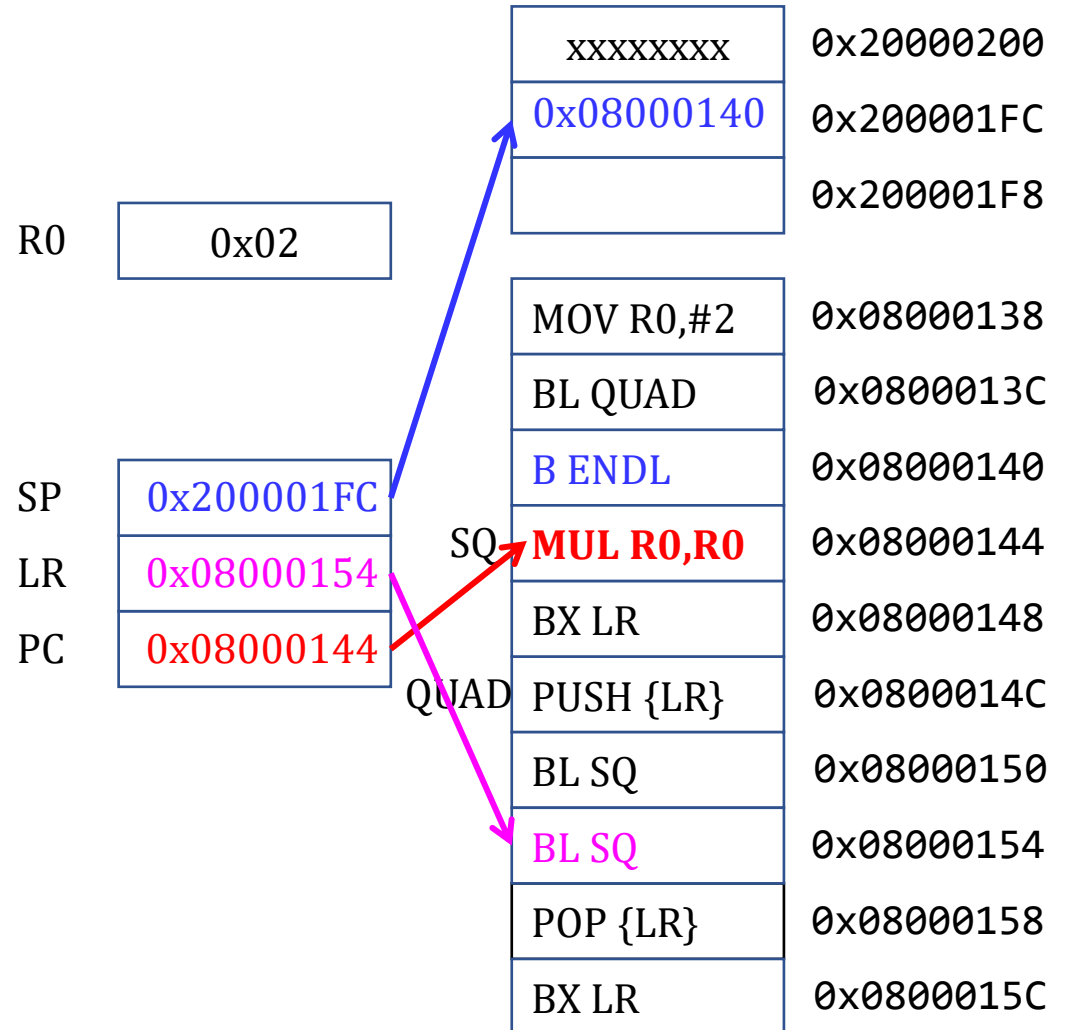
Example: R0 = R0⁴

```
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: ...
```



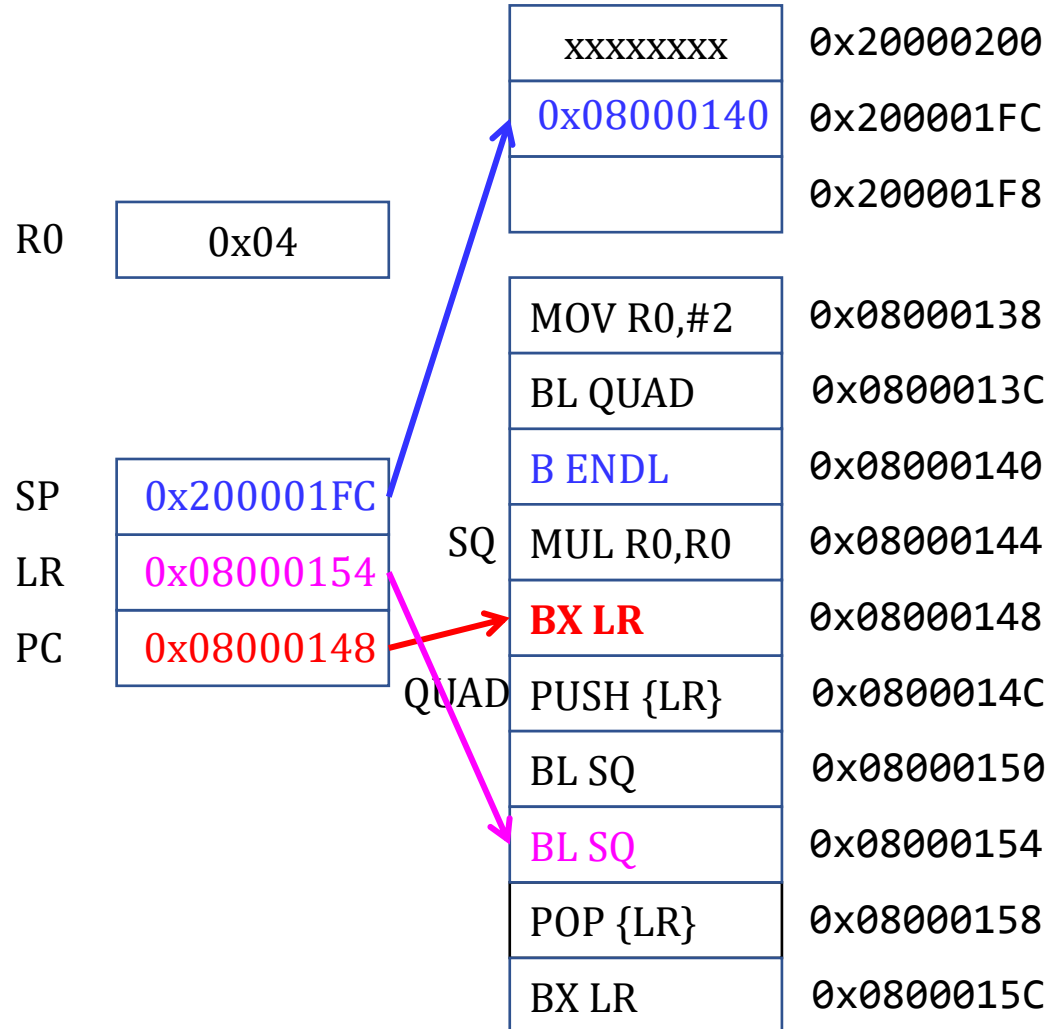
Example: $R0 = R0^4$

```
        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:    ...
```



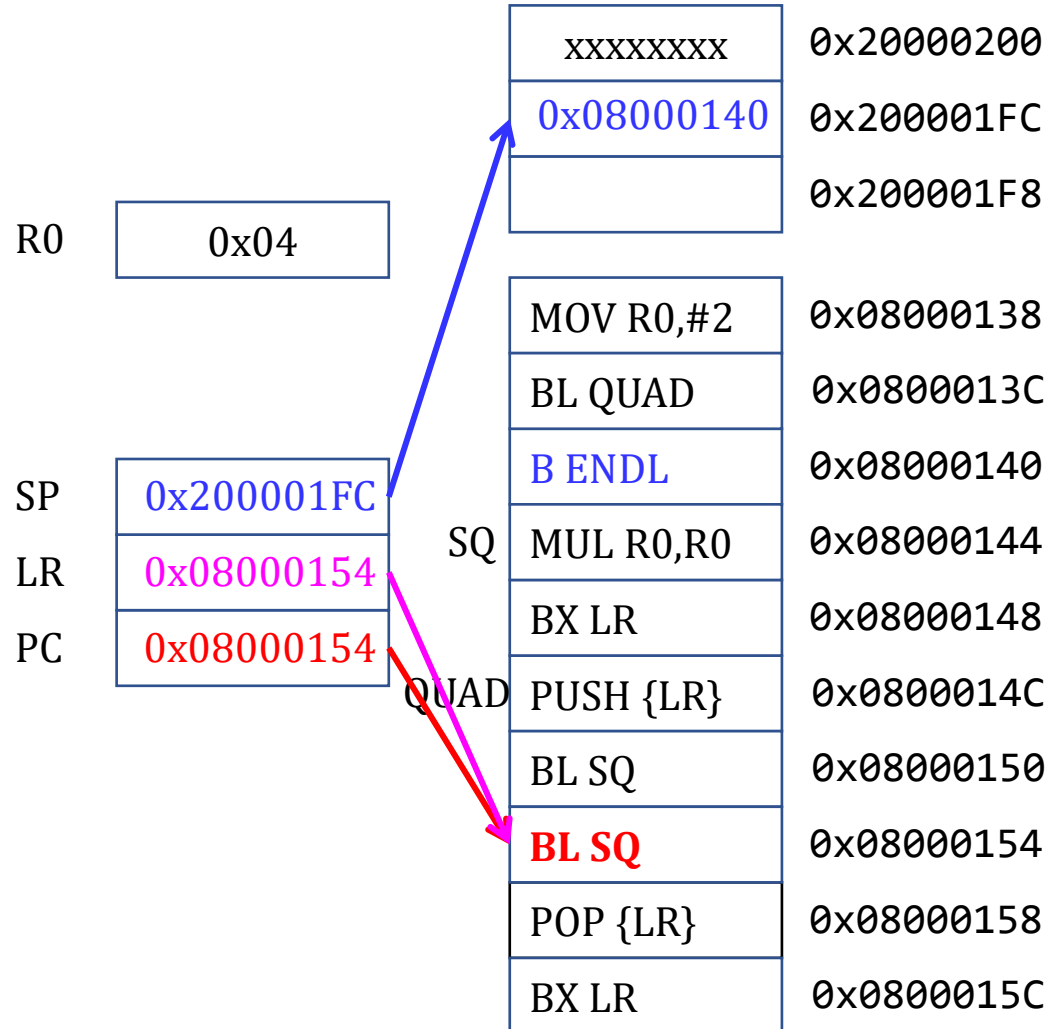
Example: R0 = R0⁴

```
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: ...
```



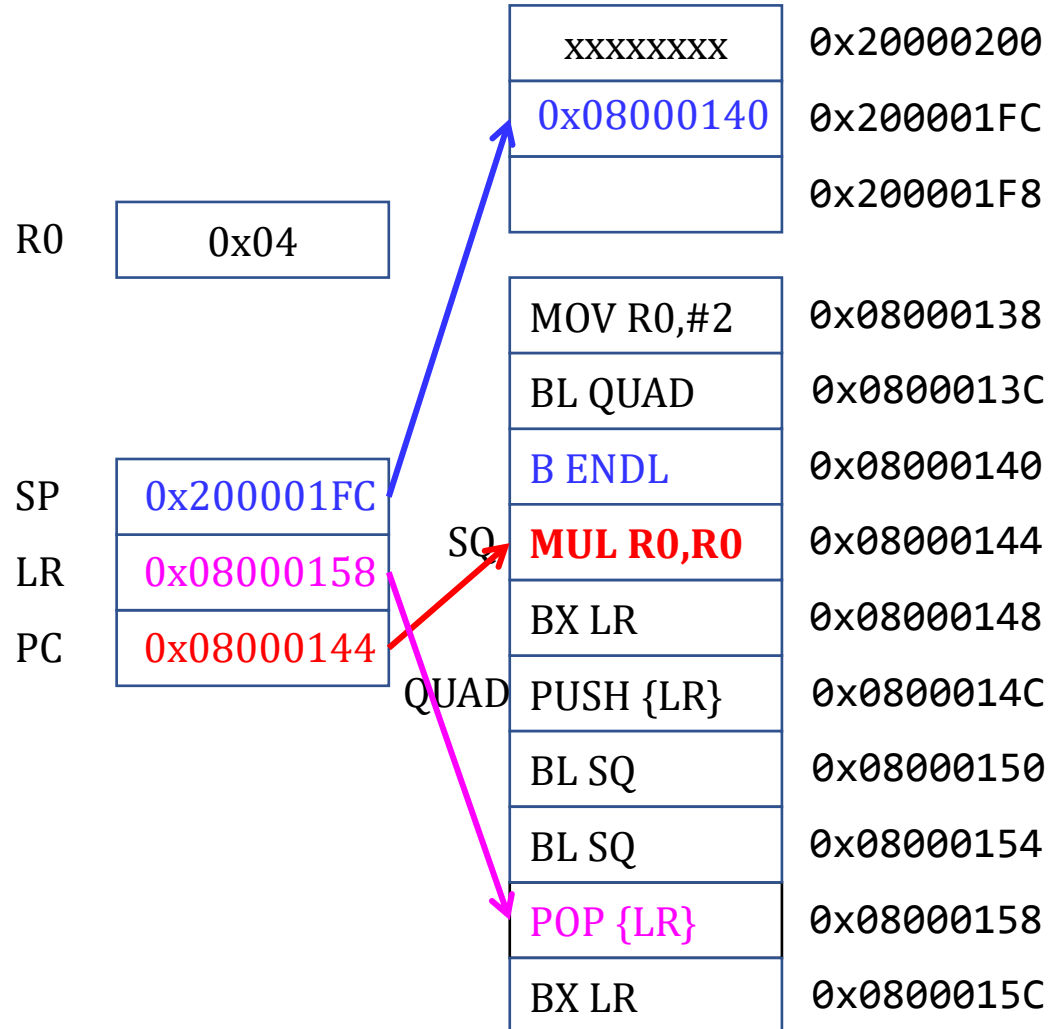
Example: R0 = R0⁴

```
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: ...
```



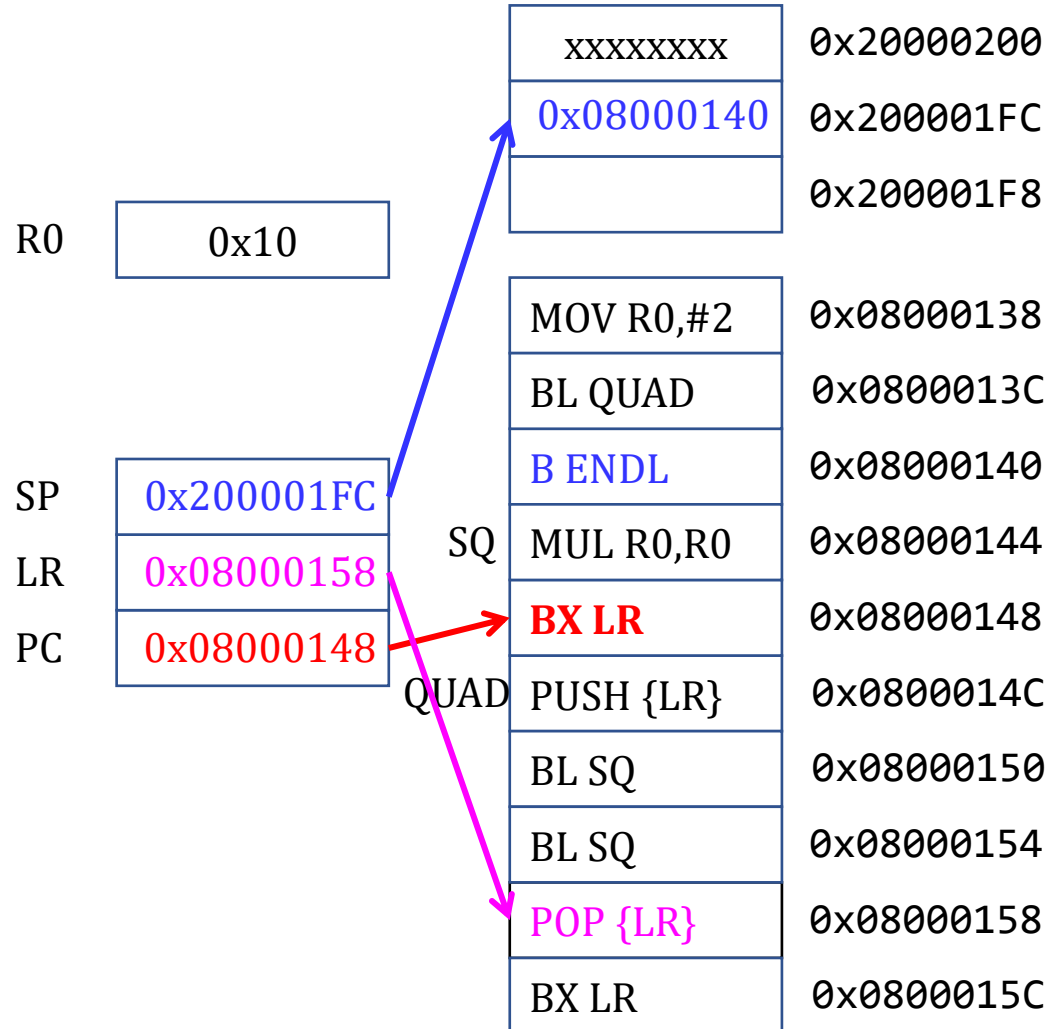
Example: R0 = R0⁴

```
        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:   ...
```



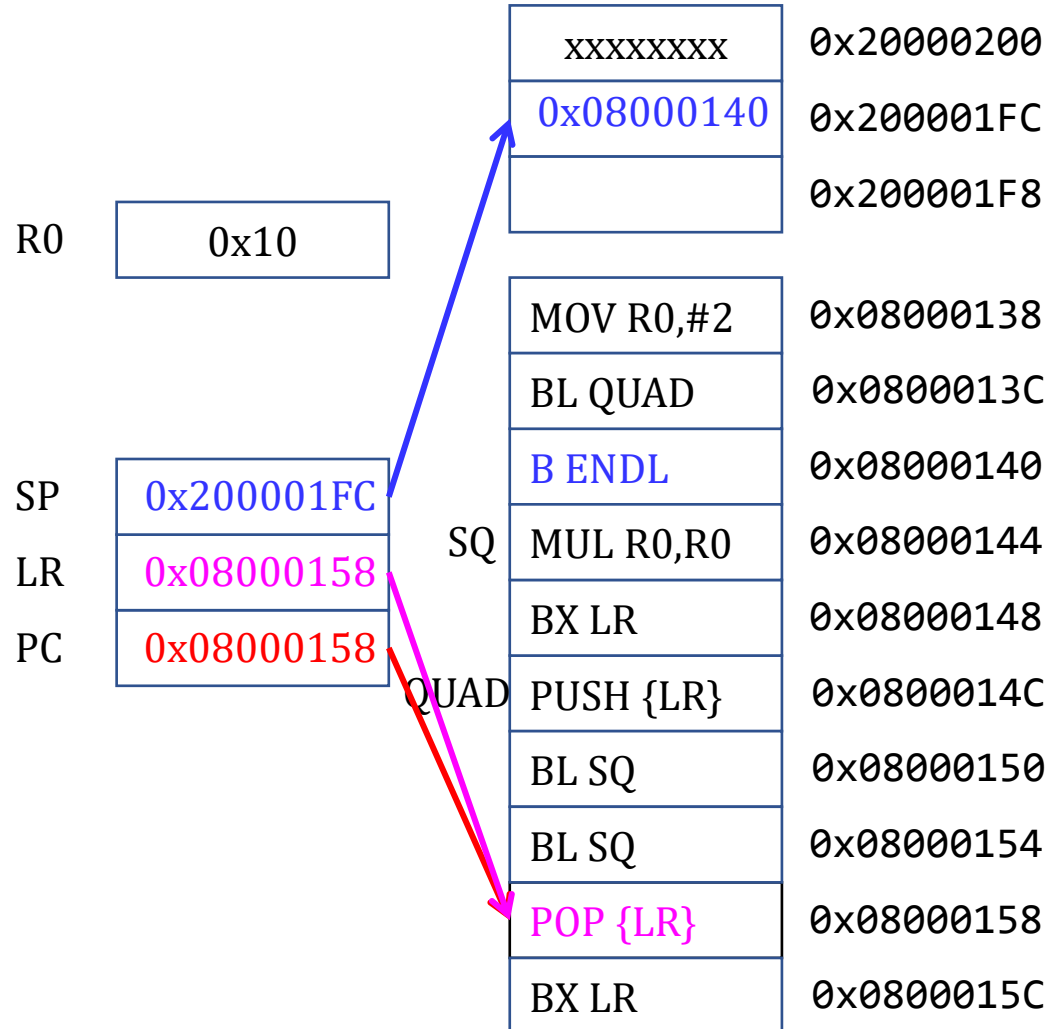
Example: R0 = R0⁴

```
        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:    ...
```



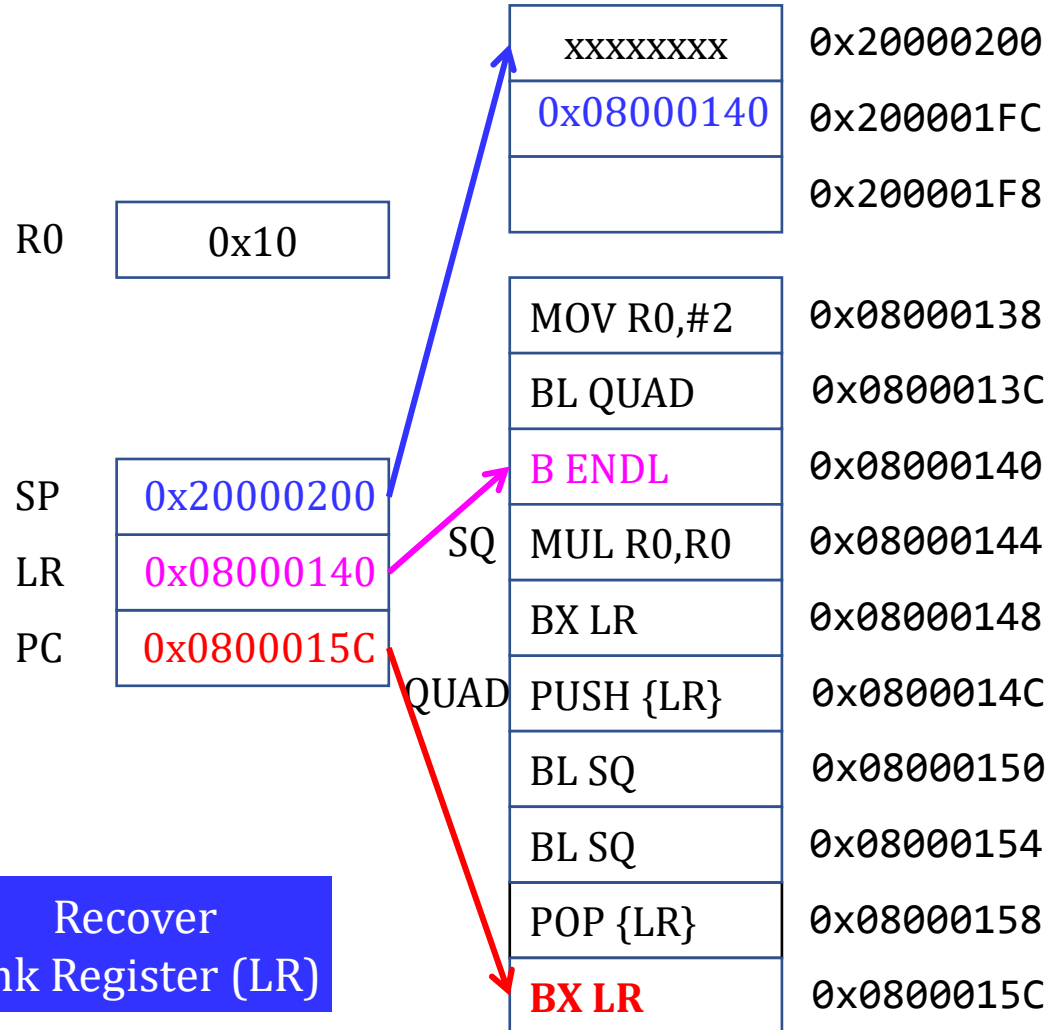
Example: R0 = R0⁴

```
        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:    ...
```



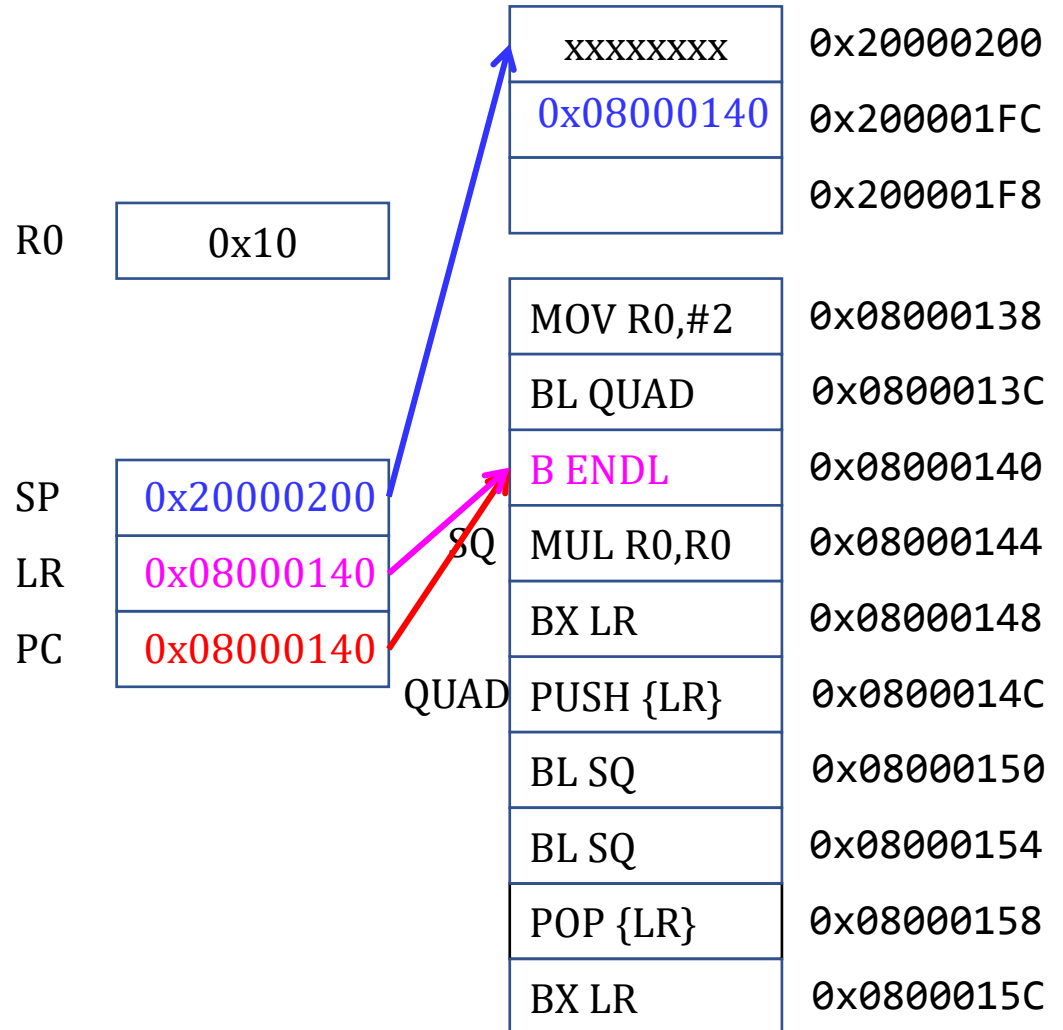
Example: R0 = R0⁴

```
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:  ...
```



Initializing the stack pointer (SP)

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

LDR sp, =__stack_top__

- Cortex-M provides an automatic mechanism that initializes SP to the value at the first four-byte of the vector table.

Stack and Recursive Functions

Recursive Functions

- A **recursive function** is one that solves its task by **calling itself** on smaller pieces of data.
- An effective tactic is to
 - divide a problem into sub-problems of the same type as the original,
 - solve those sub-problems, and
 - combine the results

Defining Factorial(n)

- Product of the first n numbers

$$1 \times 2 \times 3 \times \dots \times n$$

$$\text{factorial}(0) = 1$$

$$\text{factorial}(1) = 1 = 1 \times \text{factorial}(0)$$

$$\text{factorial}(2) = 2 \times 1 = 2 \times \text{factorial}(1)$$

$$\text{factorial}(3) = 3 \times 2 \times 1 = 3 \times \text{factorial}(2)$$

$$\text{factorial}(4) = 4 \times 3 \times 2 \times 1 = 4 \times \text{factorial}(3)$$

$$\text{factorial}(n) = n \times (n-1) \times \dots \times 1 = n \times \text{factorial}(n-1)$$

Classic Example: Factorial

- Factorial is the classic example:
 - $6! = 6 \times 5!$
 - $6! = 6 \times 5 \times 4!$
 - ...
 - $6! = 6 \times 5 \times 4 \times 3 \times 2 \times 1$
- The factorial function can be easily written as a recursive function:

```
int Factorial(int n) {  
  
    if (n < 2)  
        return 1; /* base case */  
  
    return (n * Factorial(n - 1));  
  
}
```

Recursive Factorial in Assembly

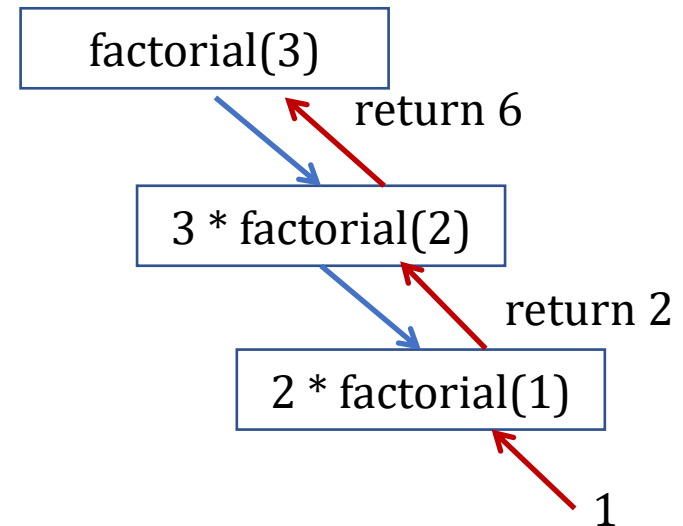
- **push LR** (& working registers) onto stack before nested call
- **pop LR** (& working registers) off stack after nested return

Recursive Factorial in Assembly

```
.global __main

__main:
    MOV r0, #0x03
0x08000130    BL    factorial
0x08000134 stop: B    stop

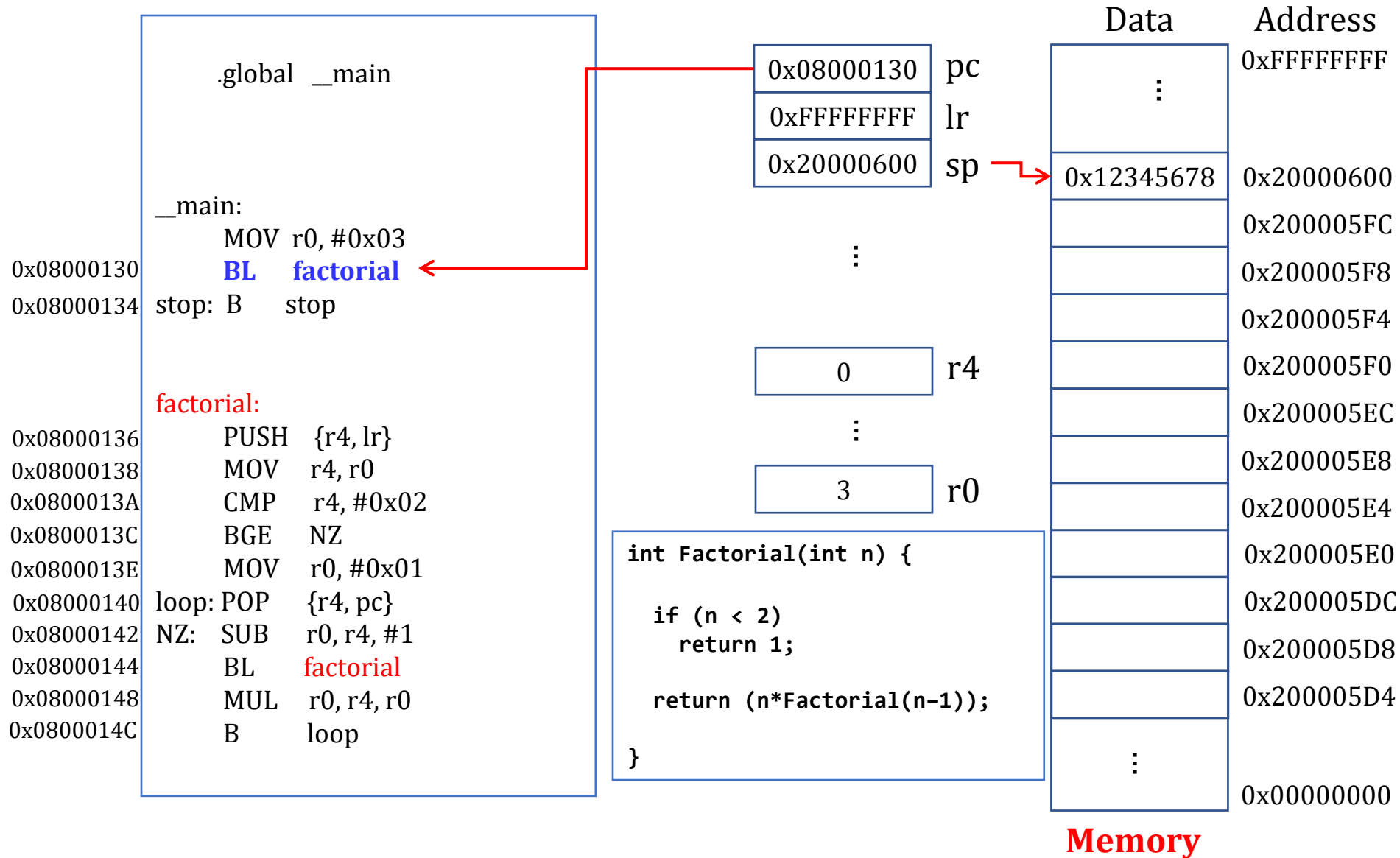
factorial:
0x08000136    PUSH   {r4, lr}
0x08000138    MOV    r4, r0
0x0800013A    CMP    r4, #0x02
0x0800013C    BGE    NZ
0x0800013E    MOV    r0, #0x01
0x08000140 loop: POP    {r4, pc}
0x08000142 NZ:  SUB    r0, r4, #1
0x08000144    BL     factorial
0x08000148    MUL    r0, r4, r0
0x0800014C    B      loop
```



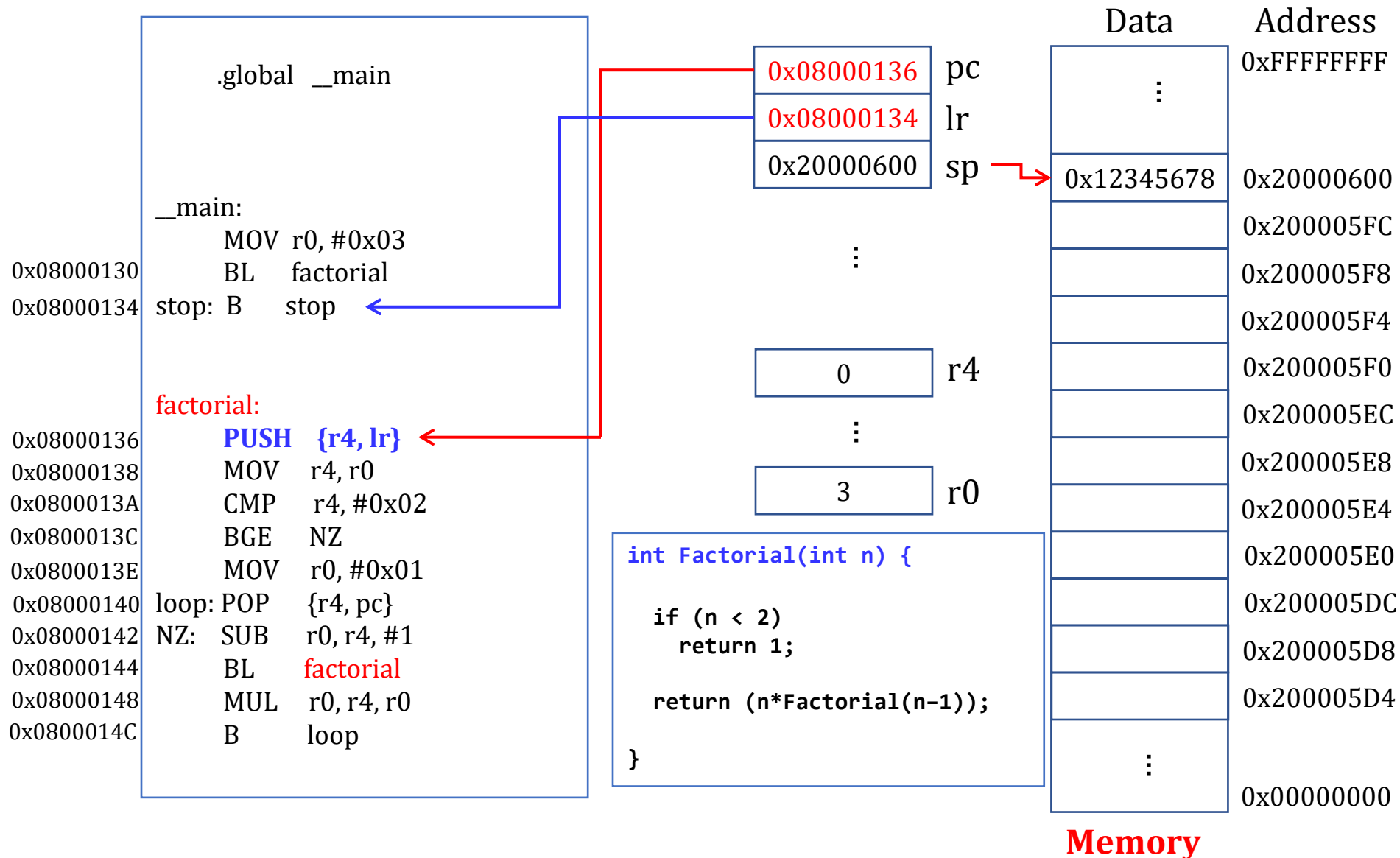
```
int Factorial(int n) {
    if (n < 2)
        return 1;

    return (n*Factorial(n-1));
}
```

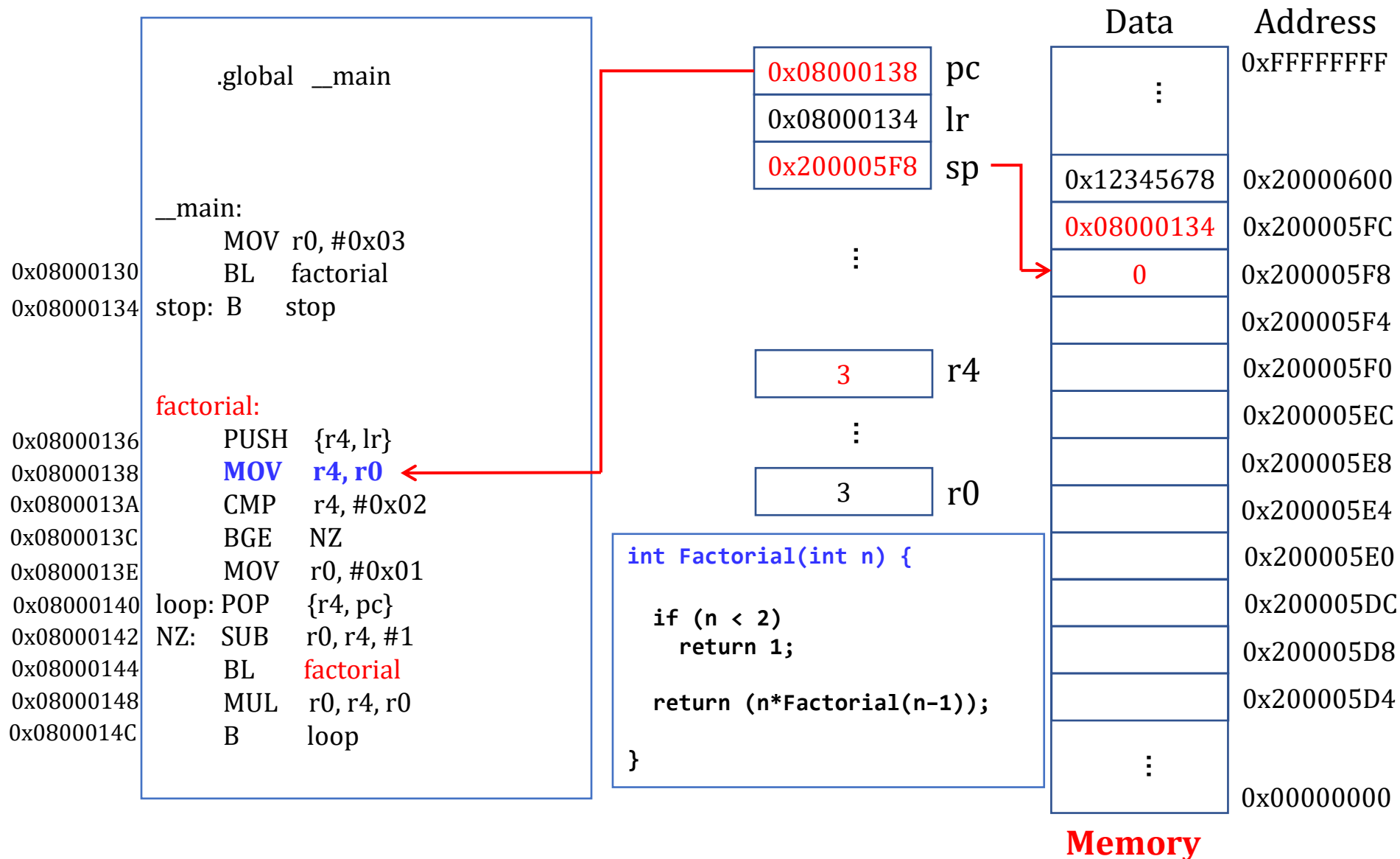
Recursive Factorial in Assembly



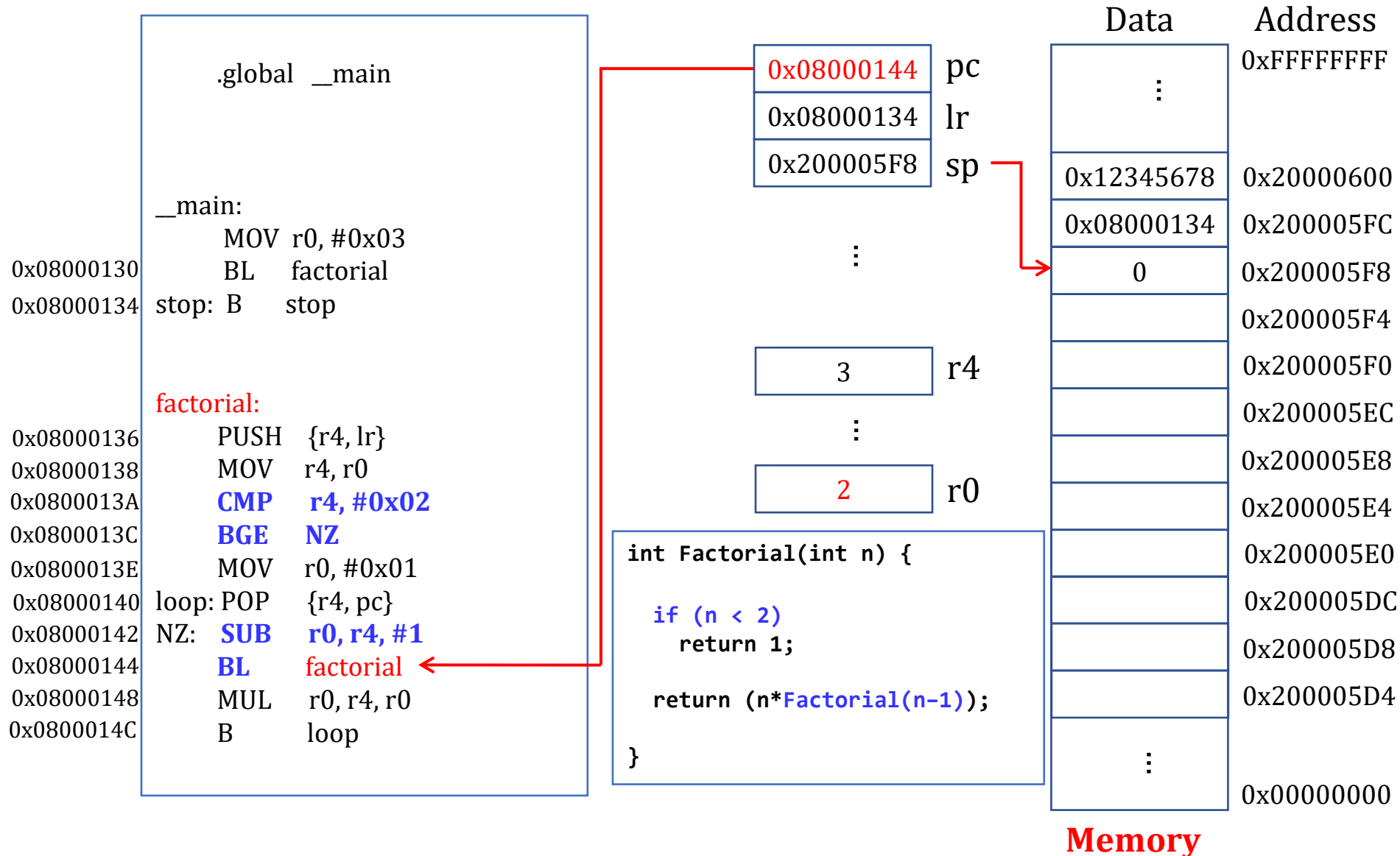
Recursive Factorial in Assembly



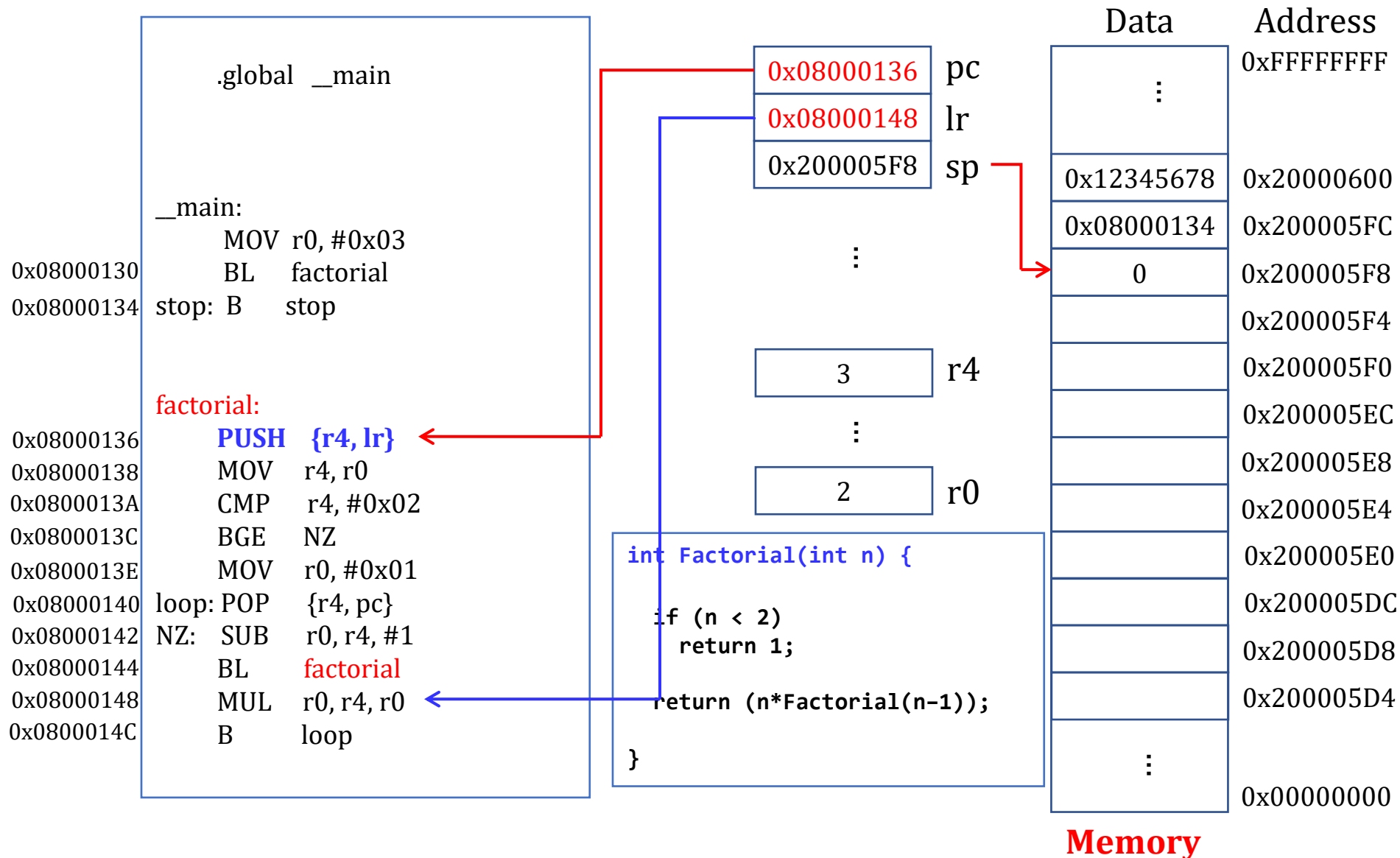
Recursive Factorial in Assembly



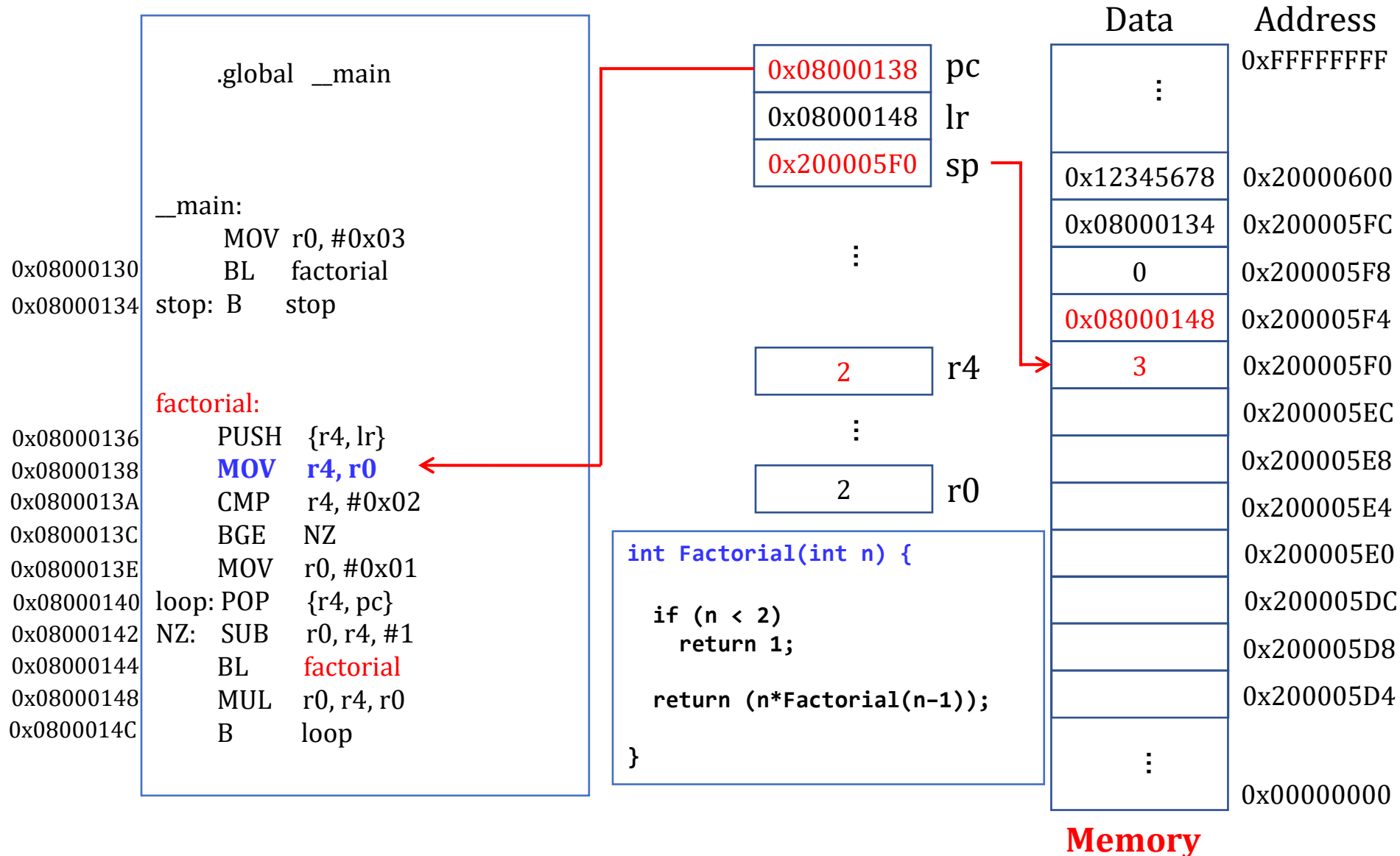
Recursive Factorial in Assembly



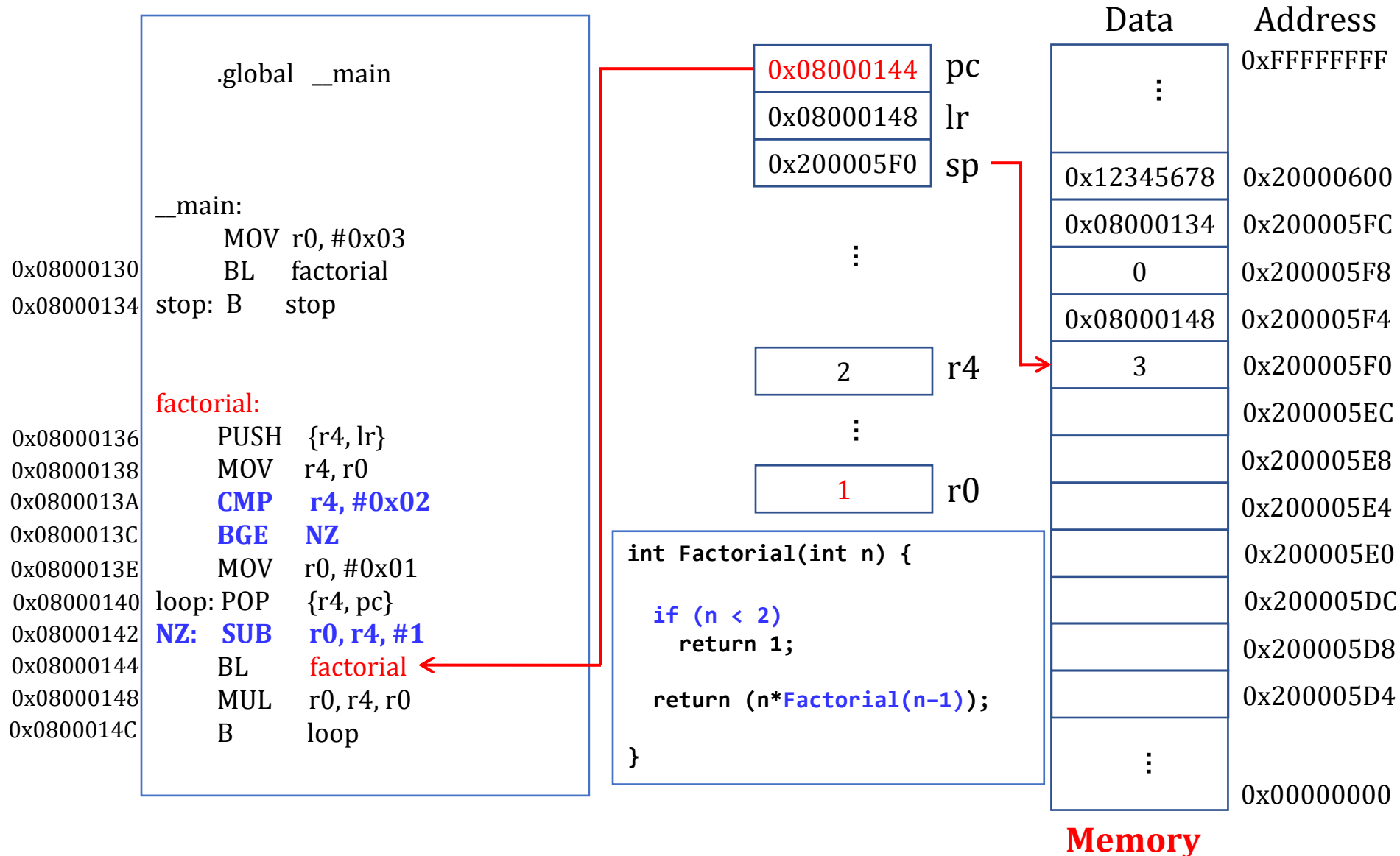
Recursive Factorial in Assembly



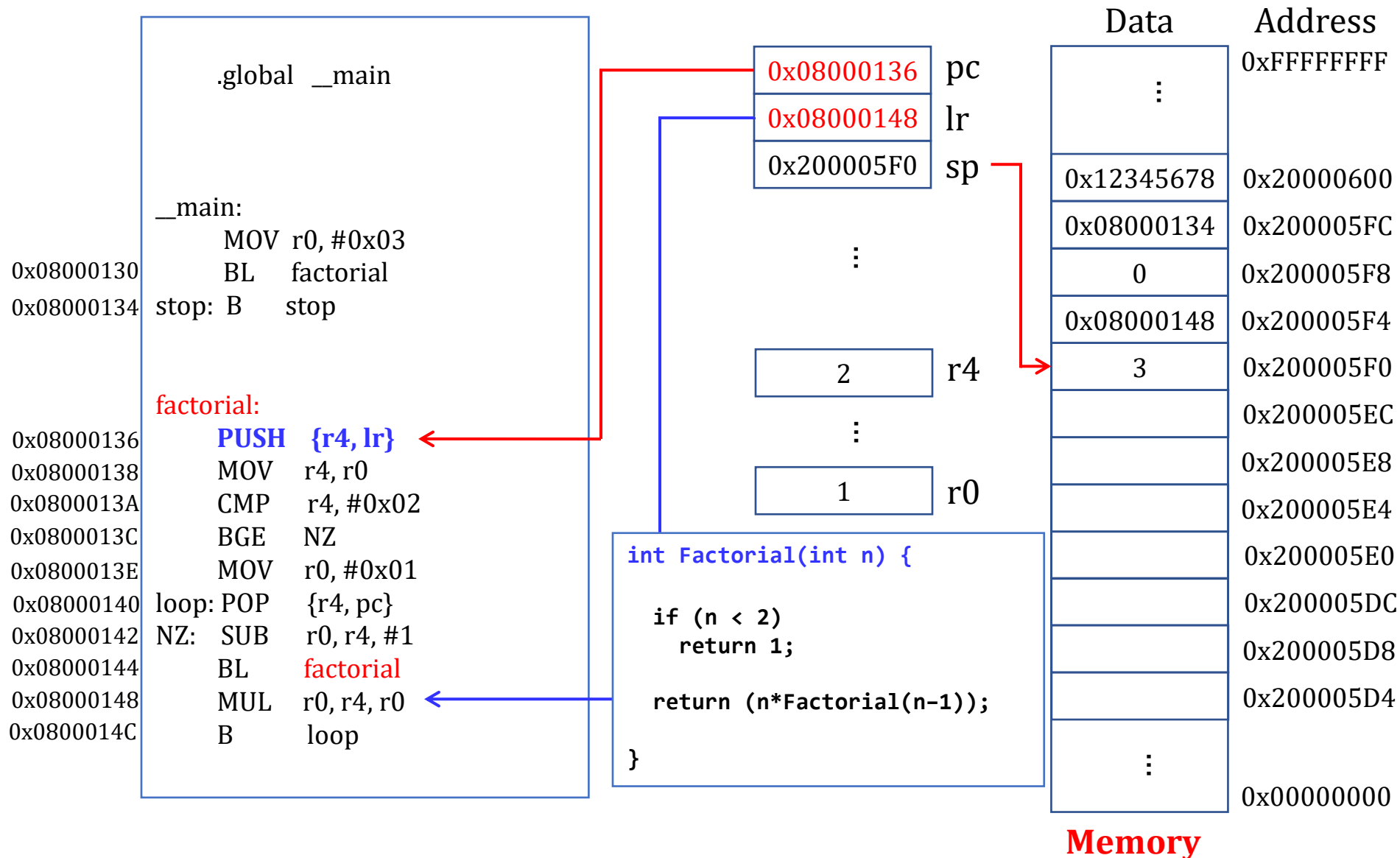
Recursive Factorial in Assembly



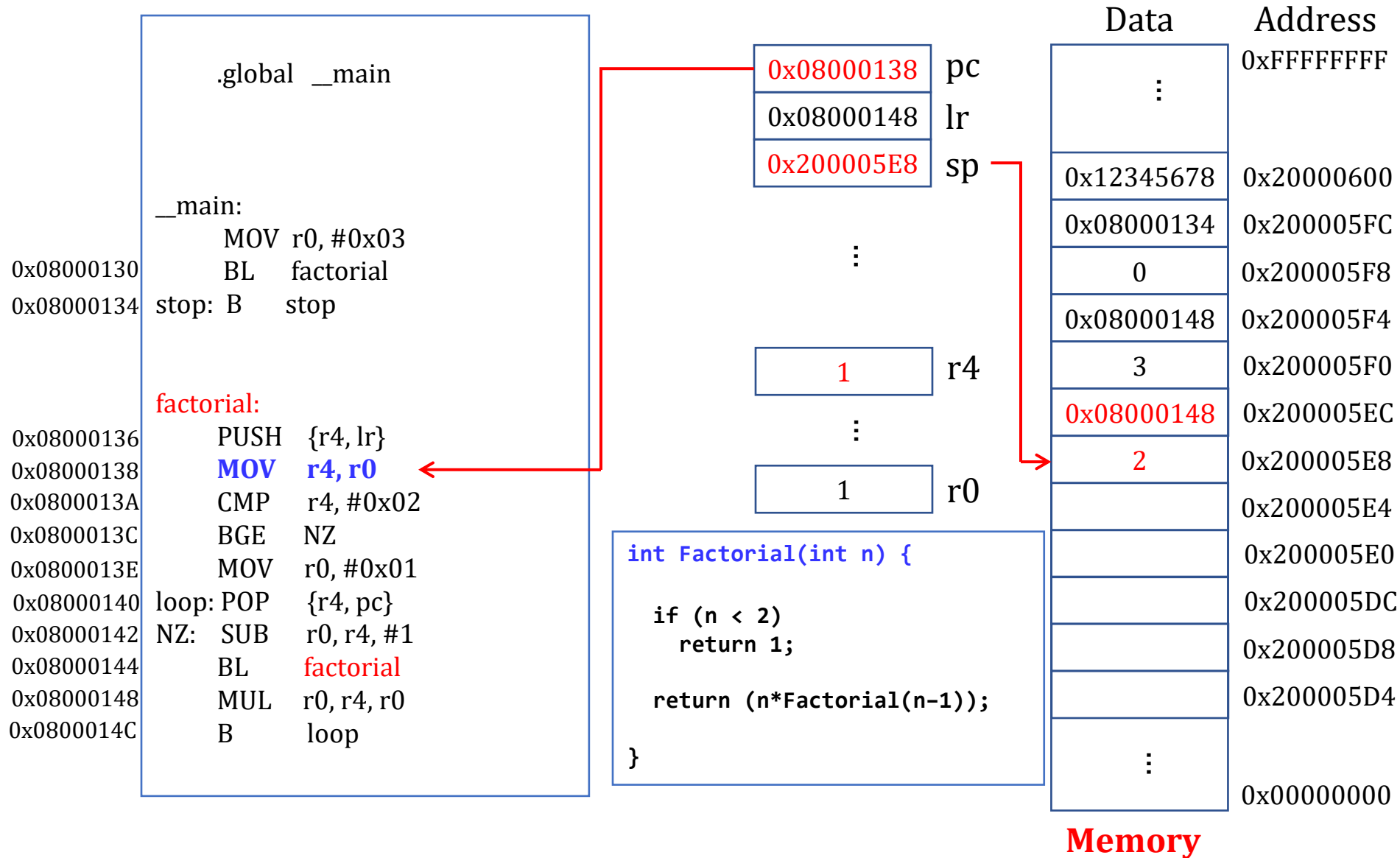
Recursive Factorial in Assembly



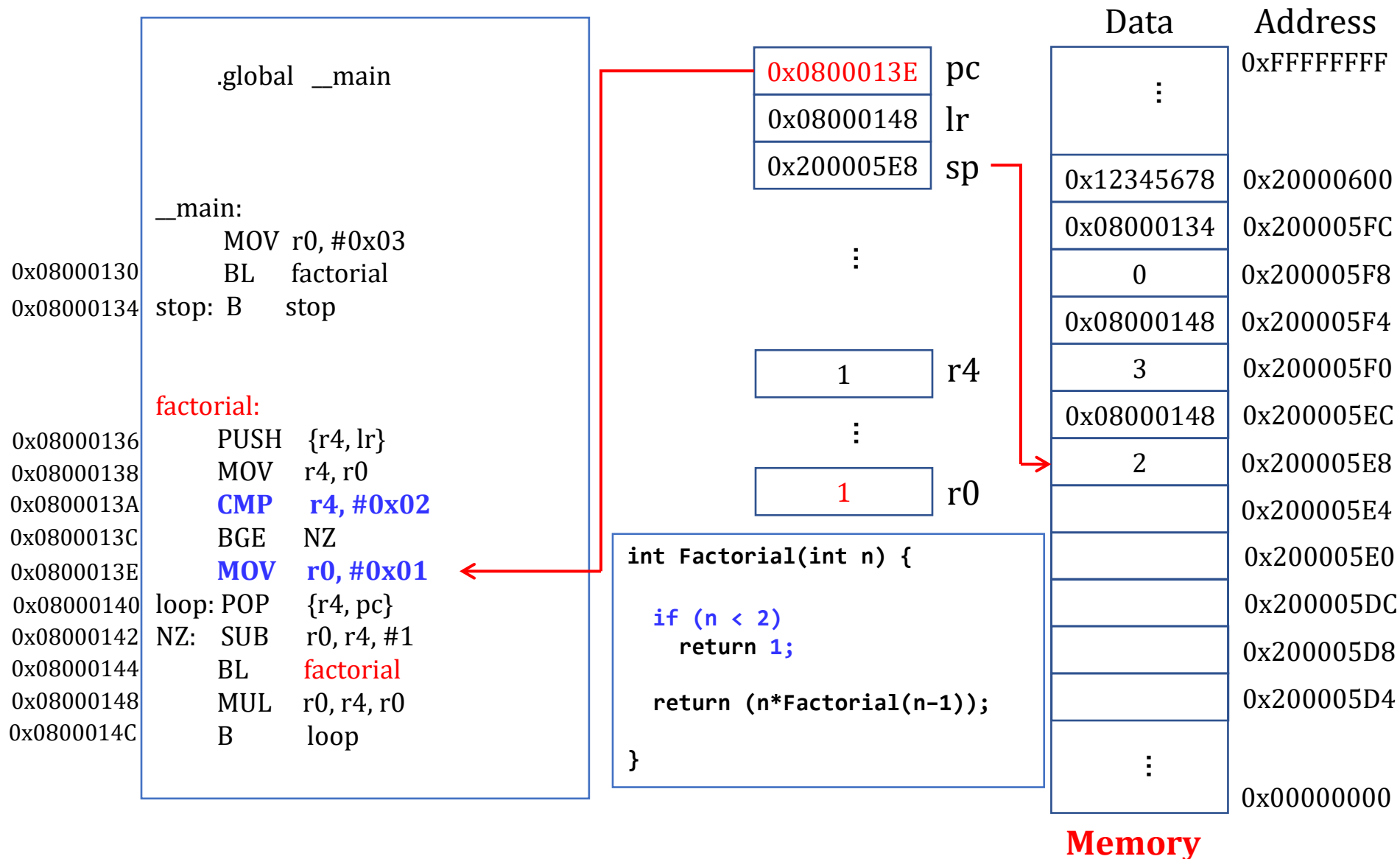
Recursive Factorial in Assembly



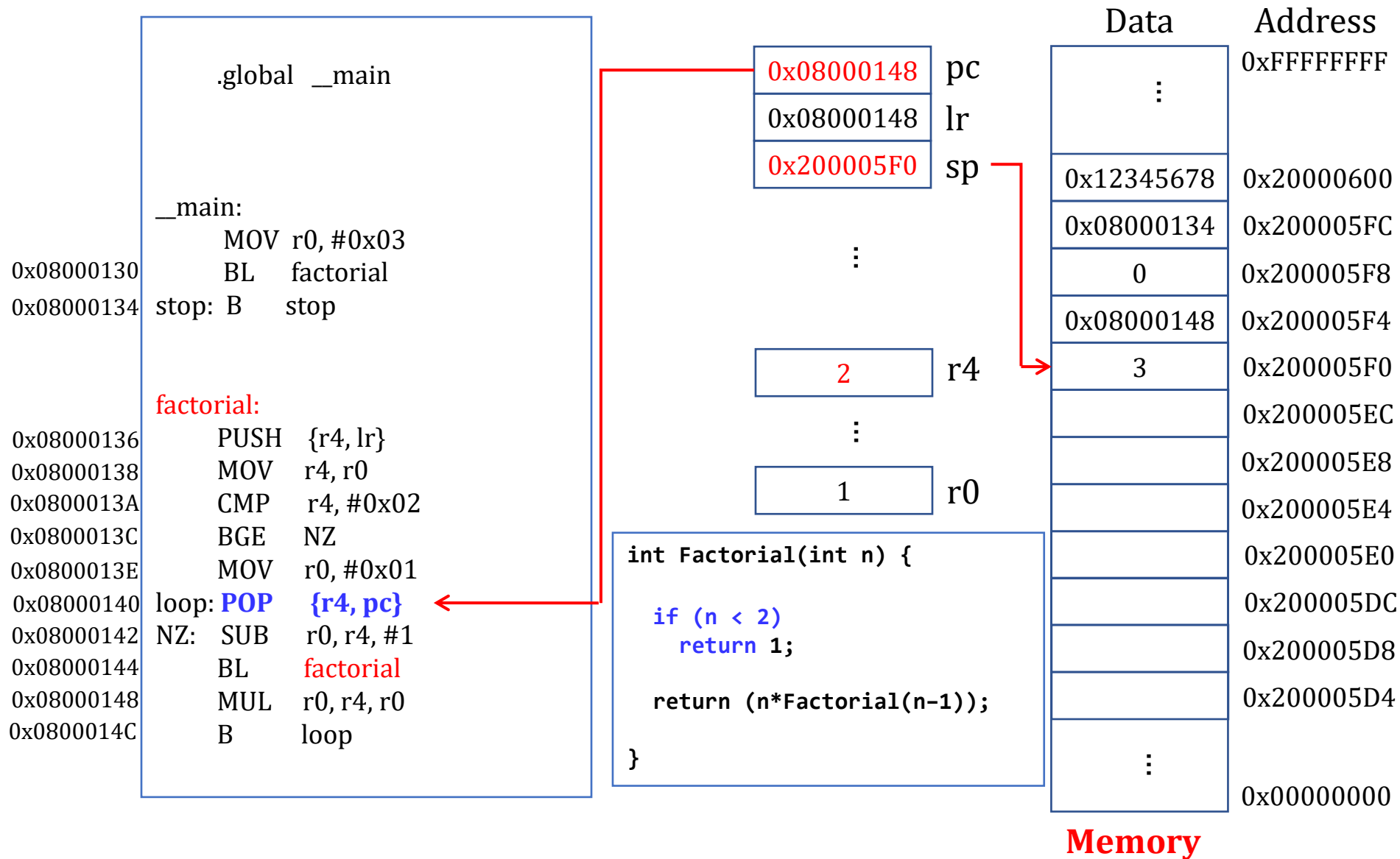
Recursive Factorial in Assembly



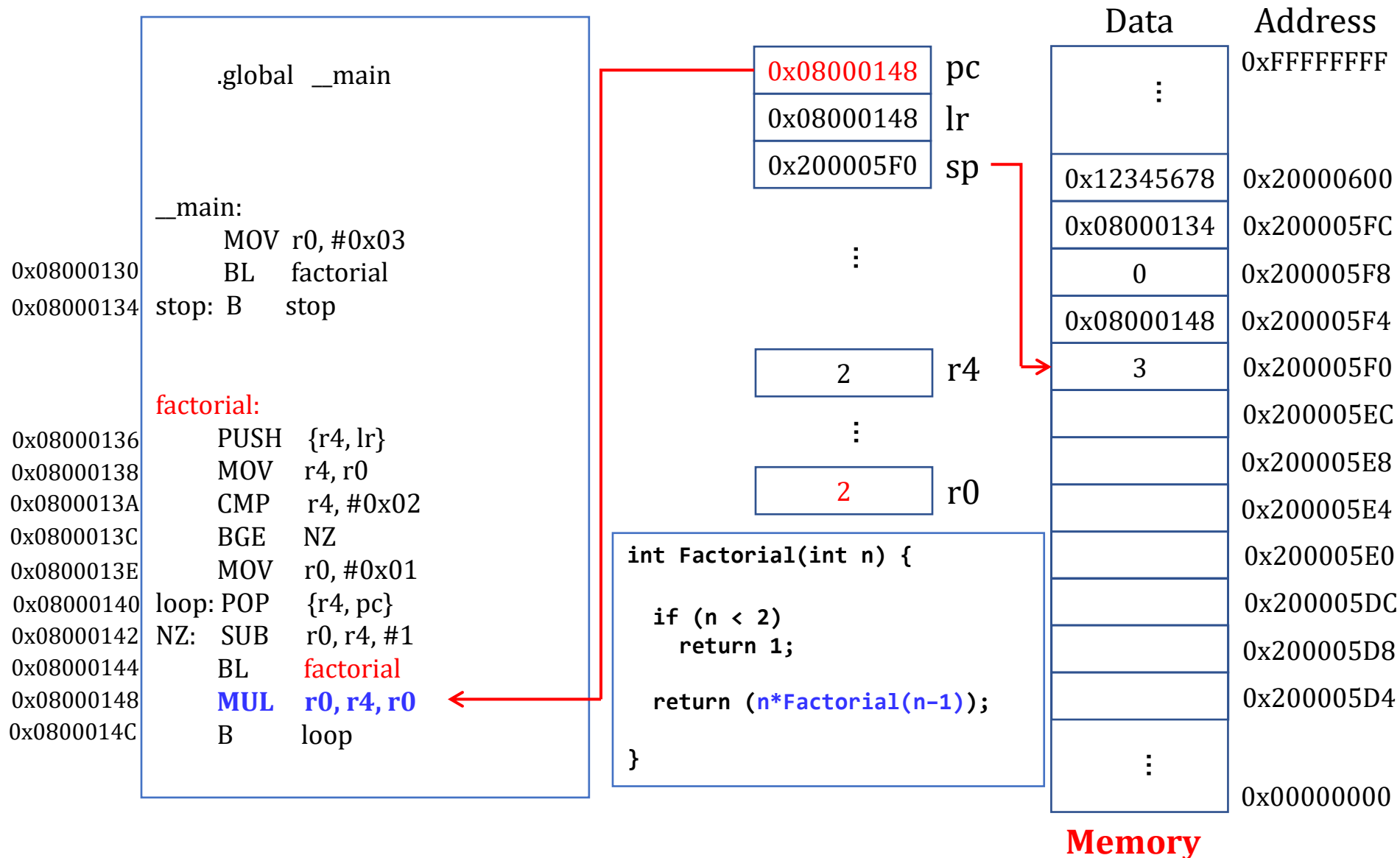
Recursive Factorial in Assembly



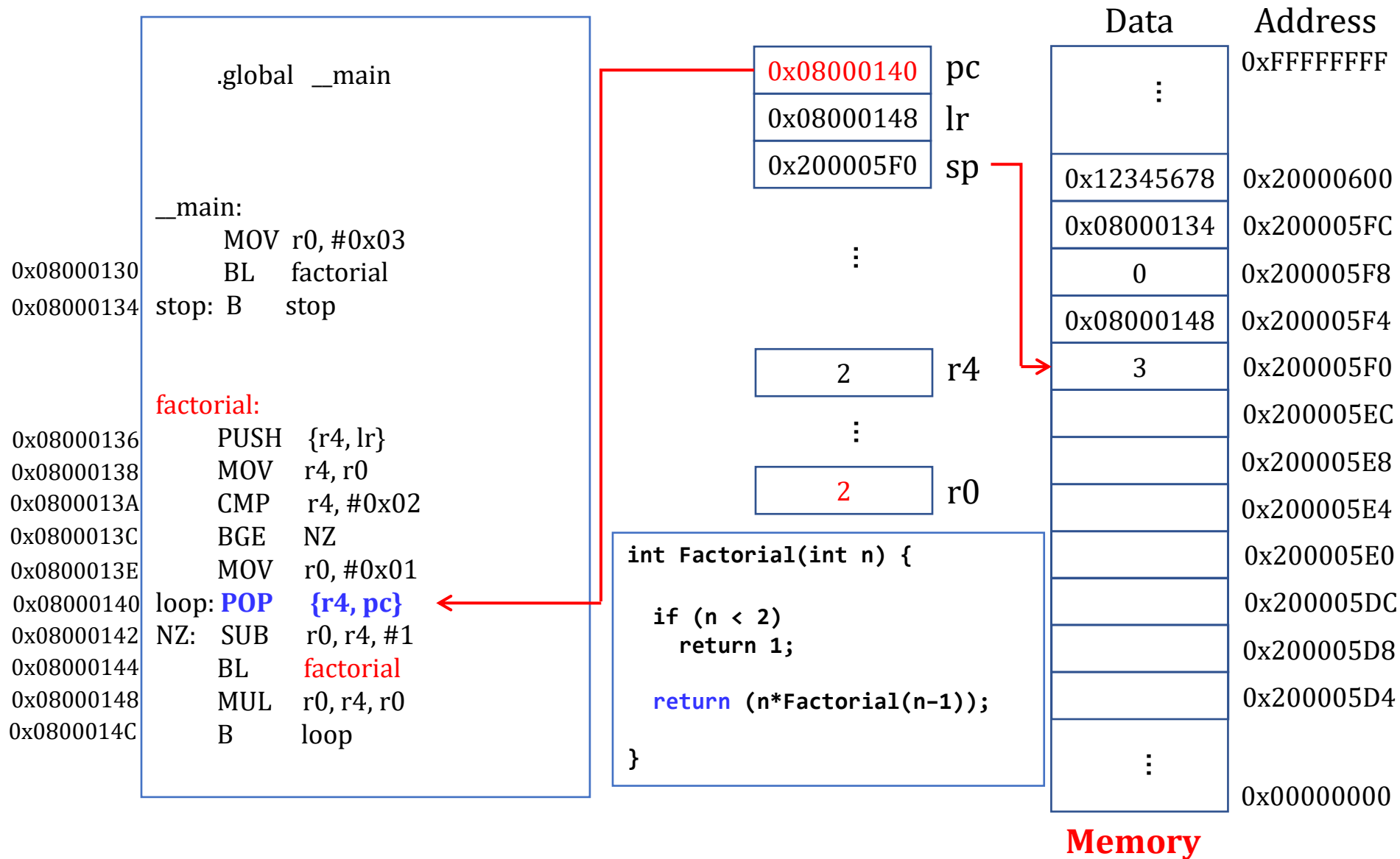
Recursive Factorial in Assembly



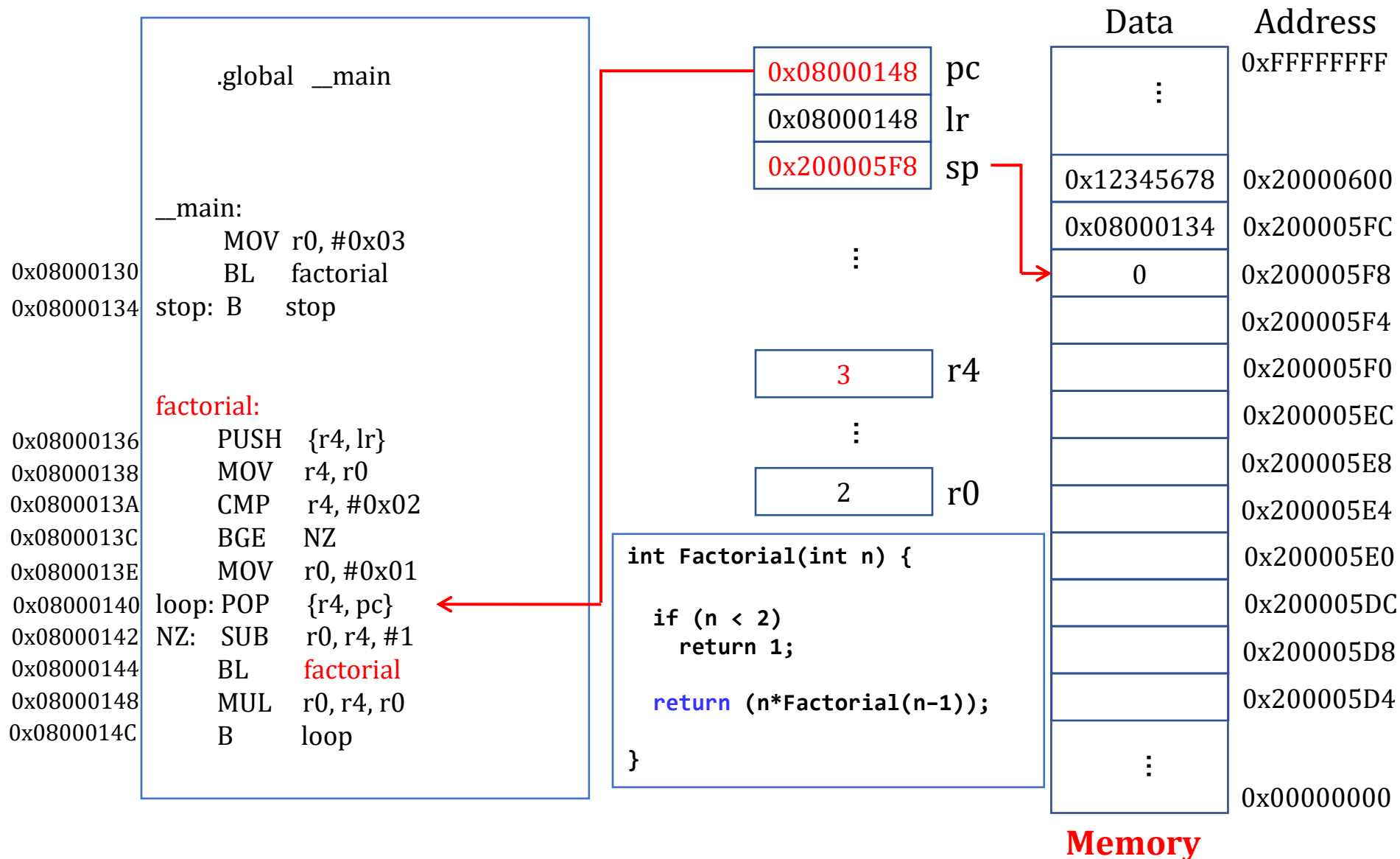
Recursive Factorial in Assembly



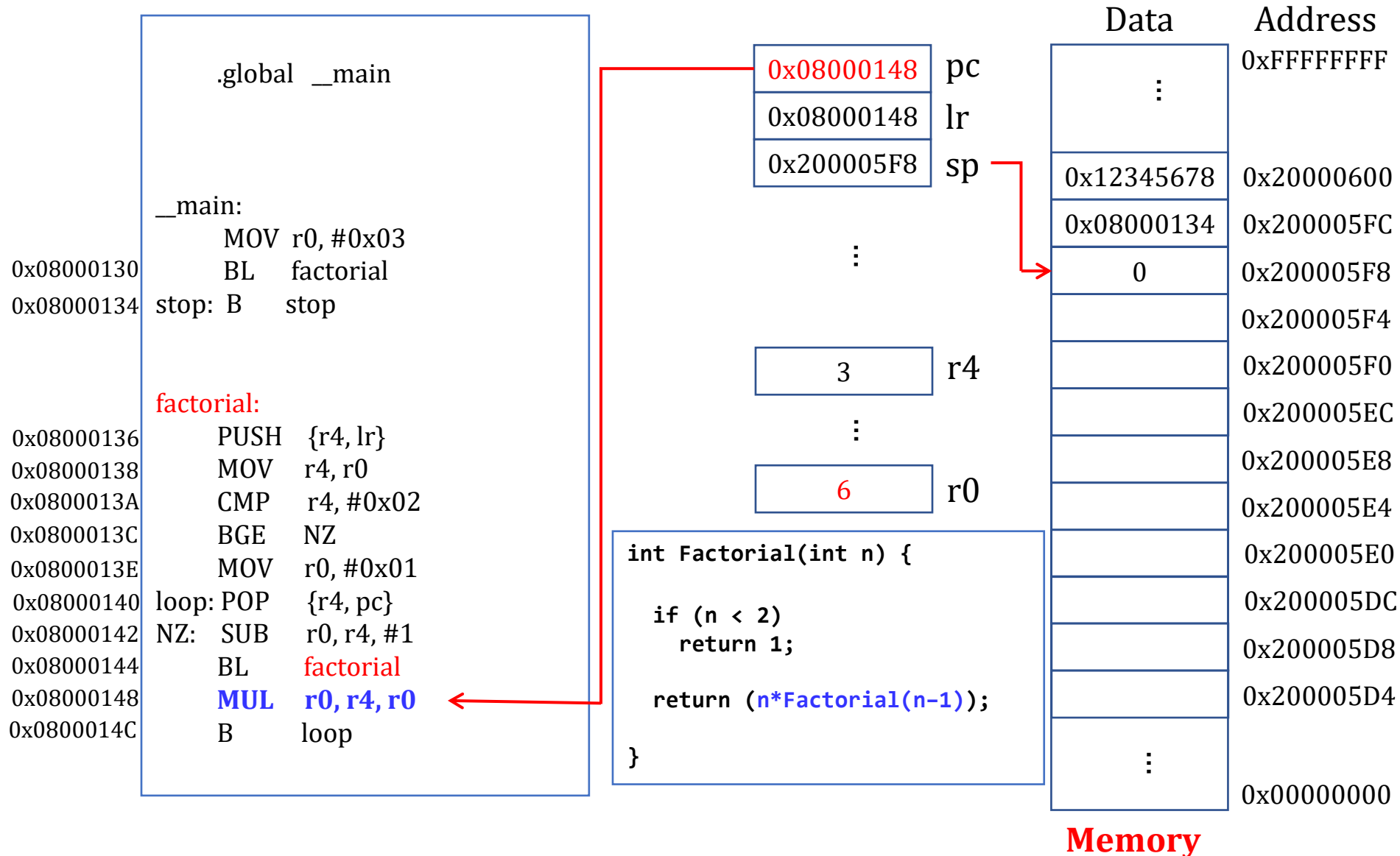
Recursive Factorial in Assembly



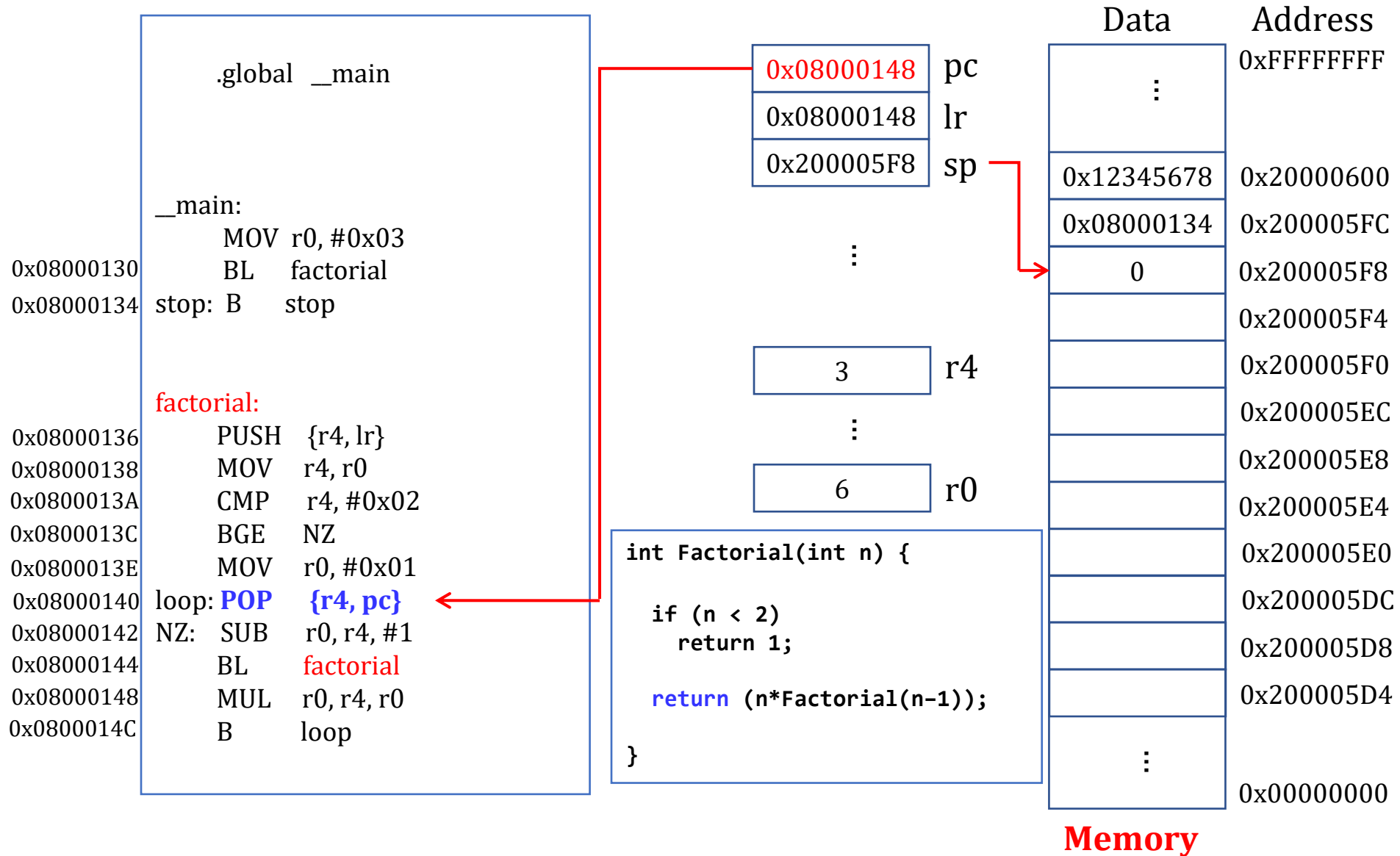
Recursive Factorial in Assembly



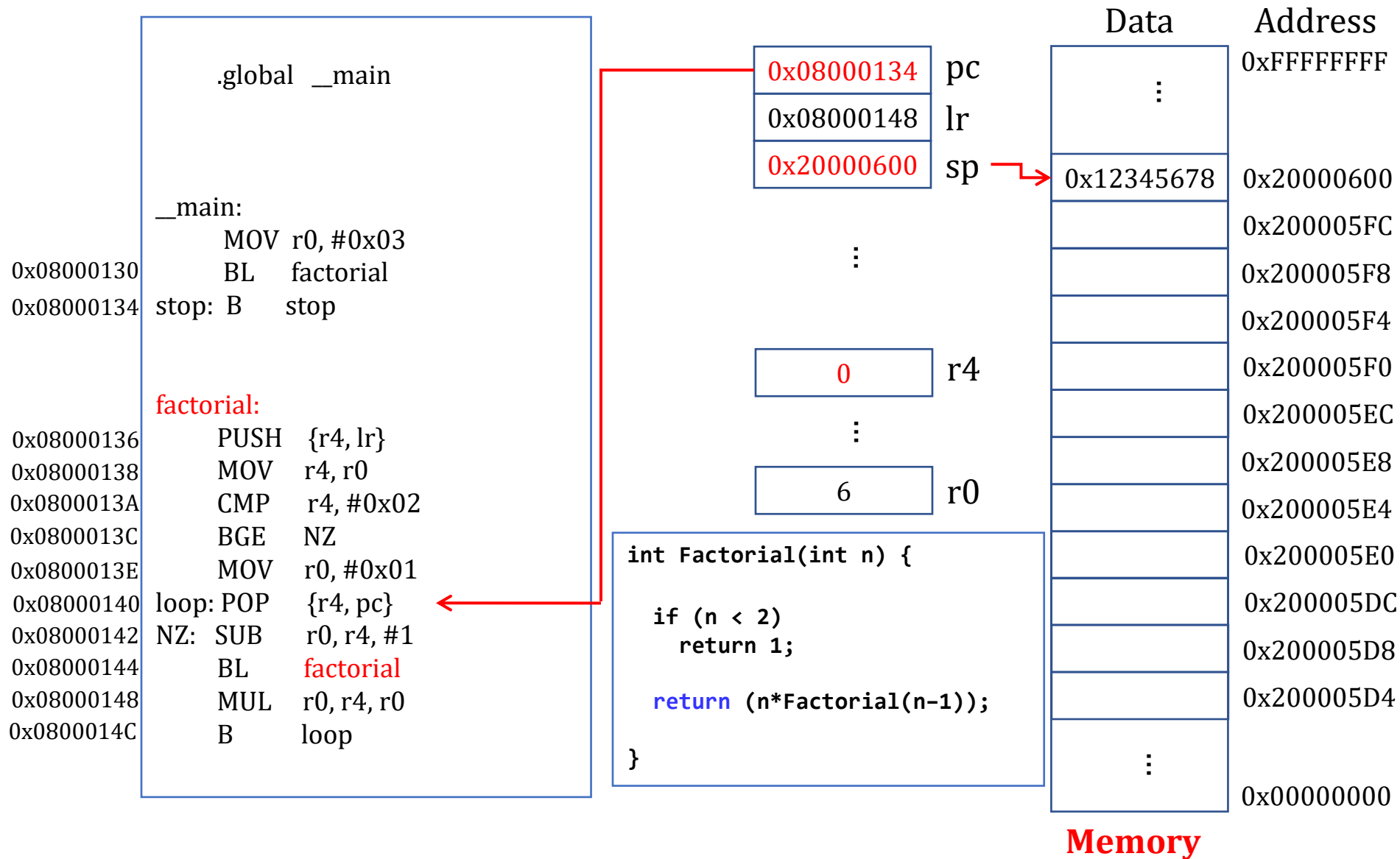
Recursive Factorial in Assembly



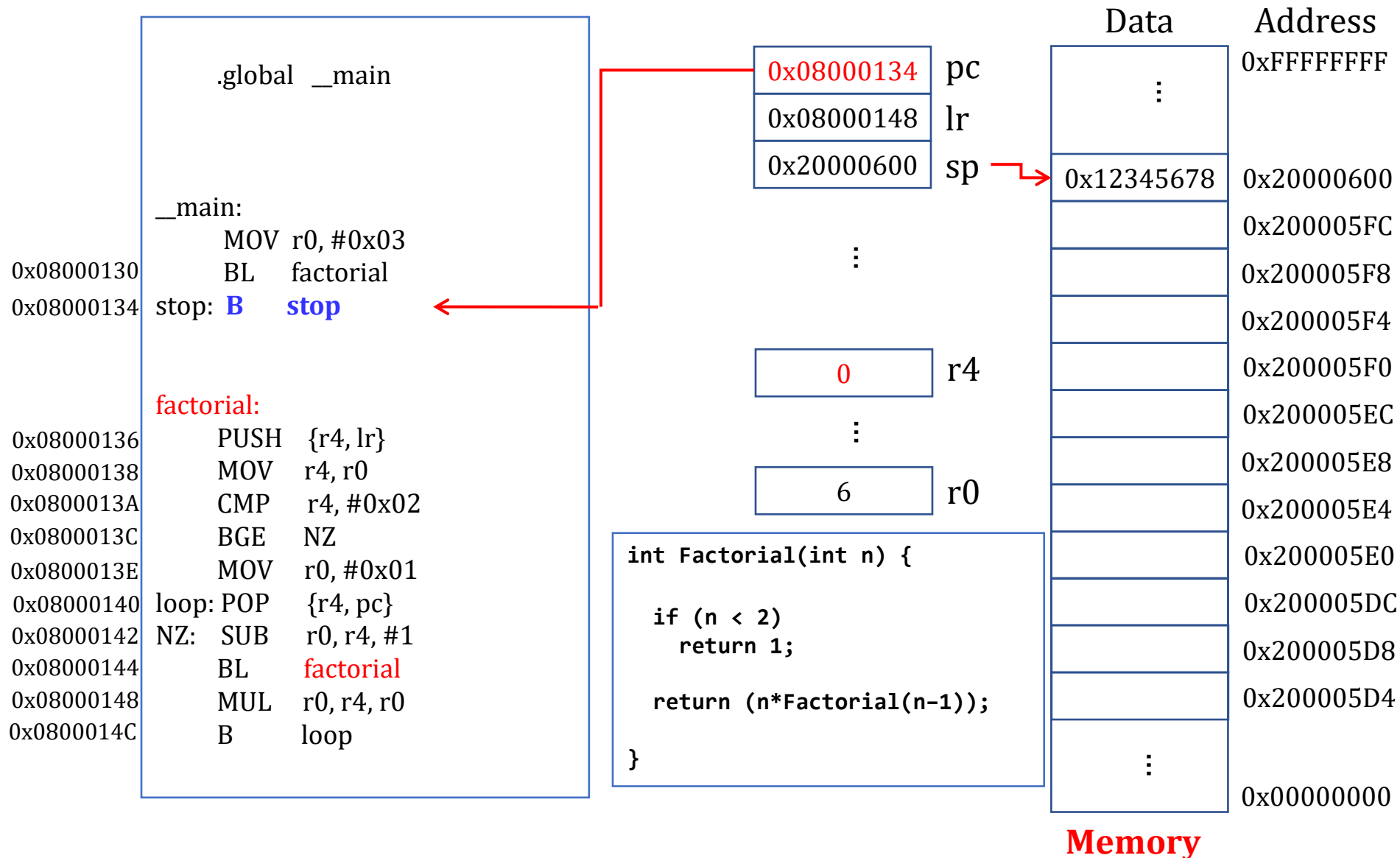
Recursive Factorial in Assembly



Recursive Factorial in Assembly



Recursive Factorial in Assembly



Summary

- Subroutine and calling convention
- Nested Subroutine calls via Stack