ARM Subroutines

Lecture 8

Yeongpil Cho

Hanynag University

Topics

- Passing Parameters to Subroutines via Registers
- Preserve Environment via Stack
- Stack and Recursive Functions

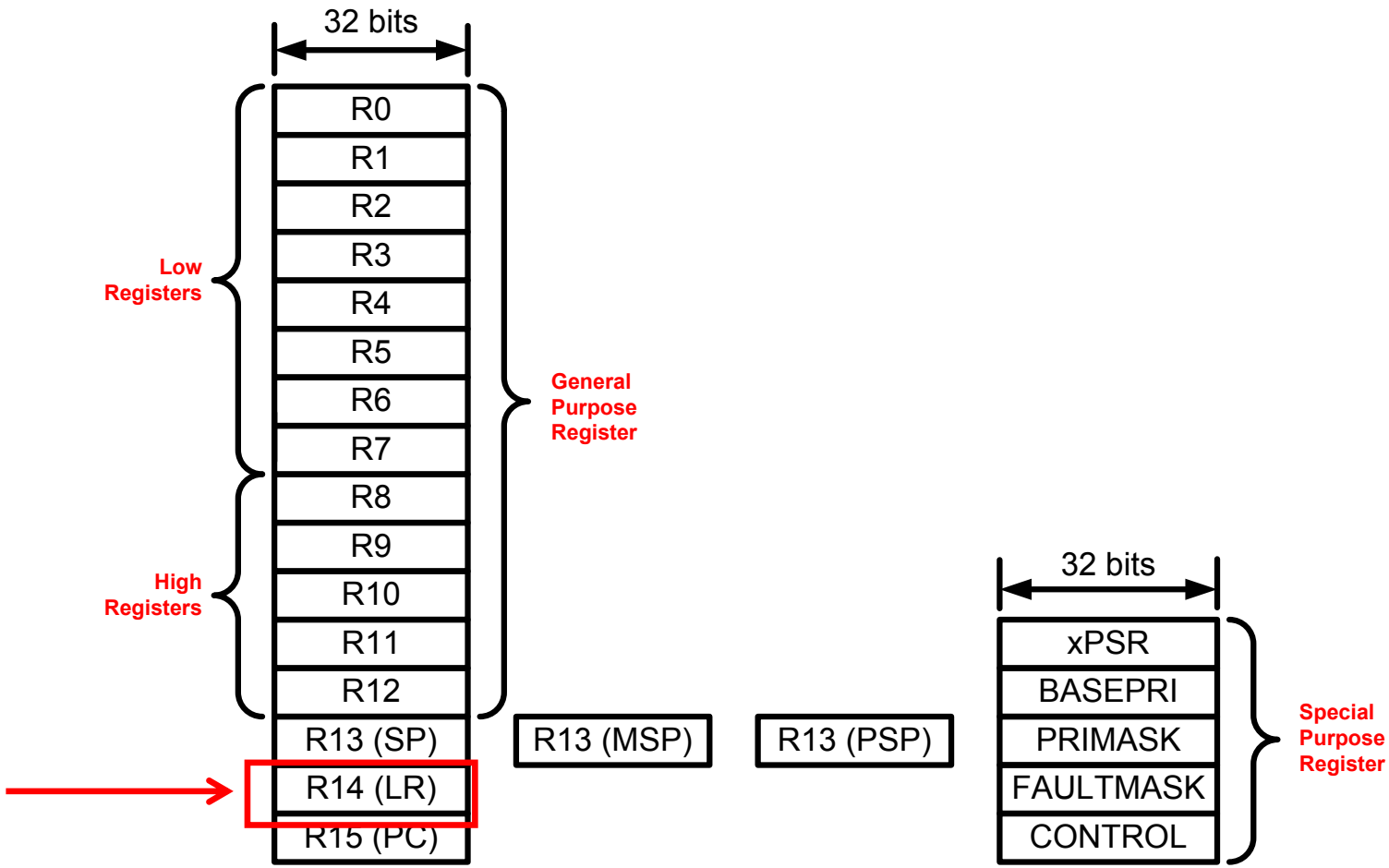
Passing Parameters to Subroutines via Registers

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



Call a Subroutine

Caller Program	Subroutine/Callee

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

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)

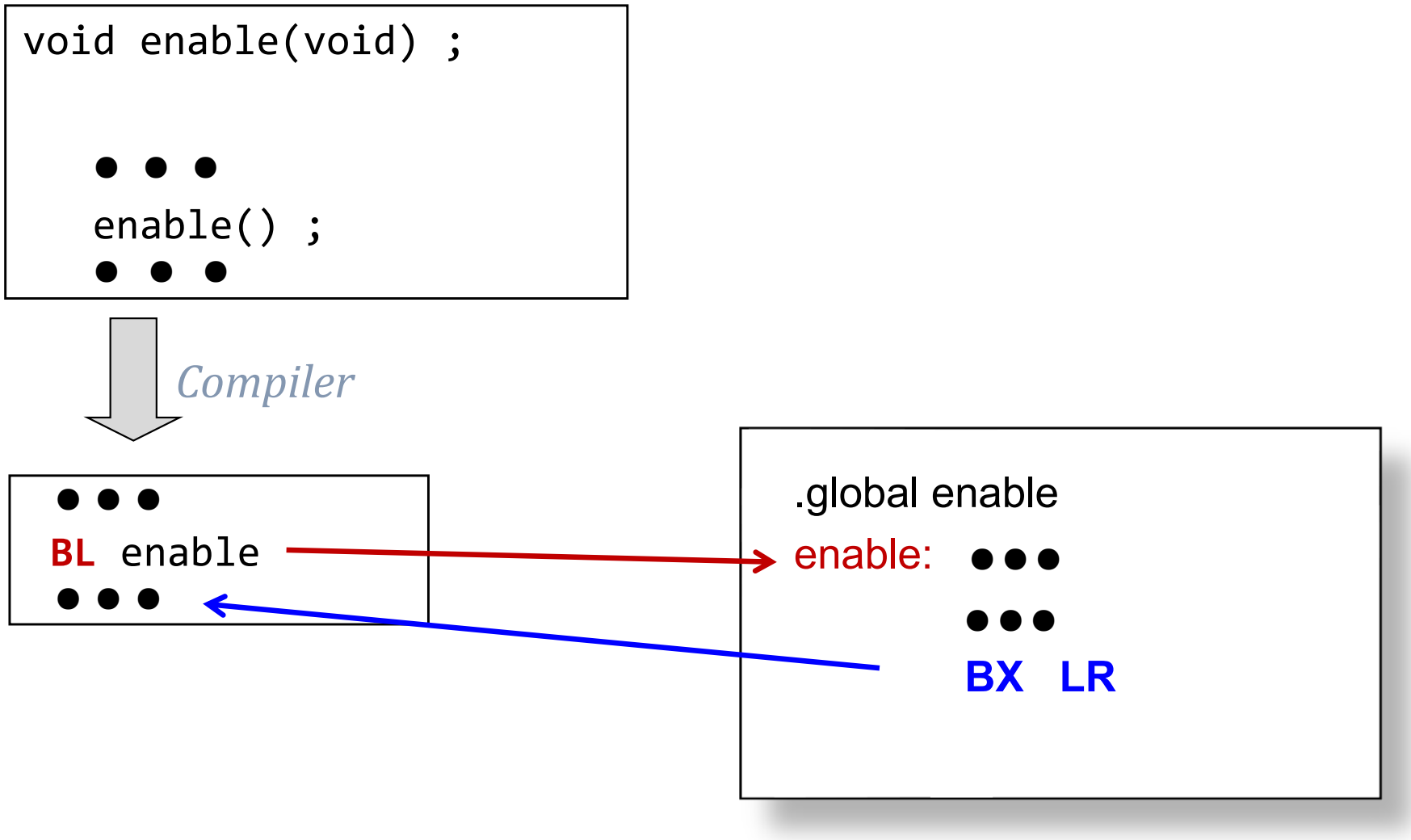
Caller Program

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

Subroutine/Callee

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

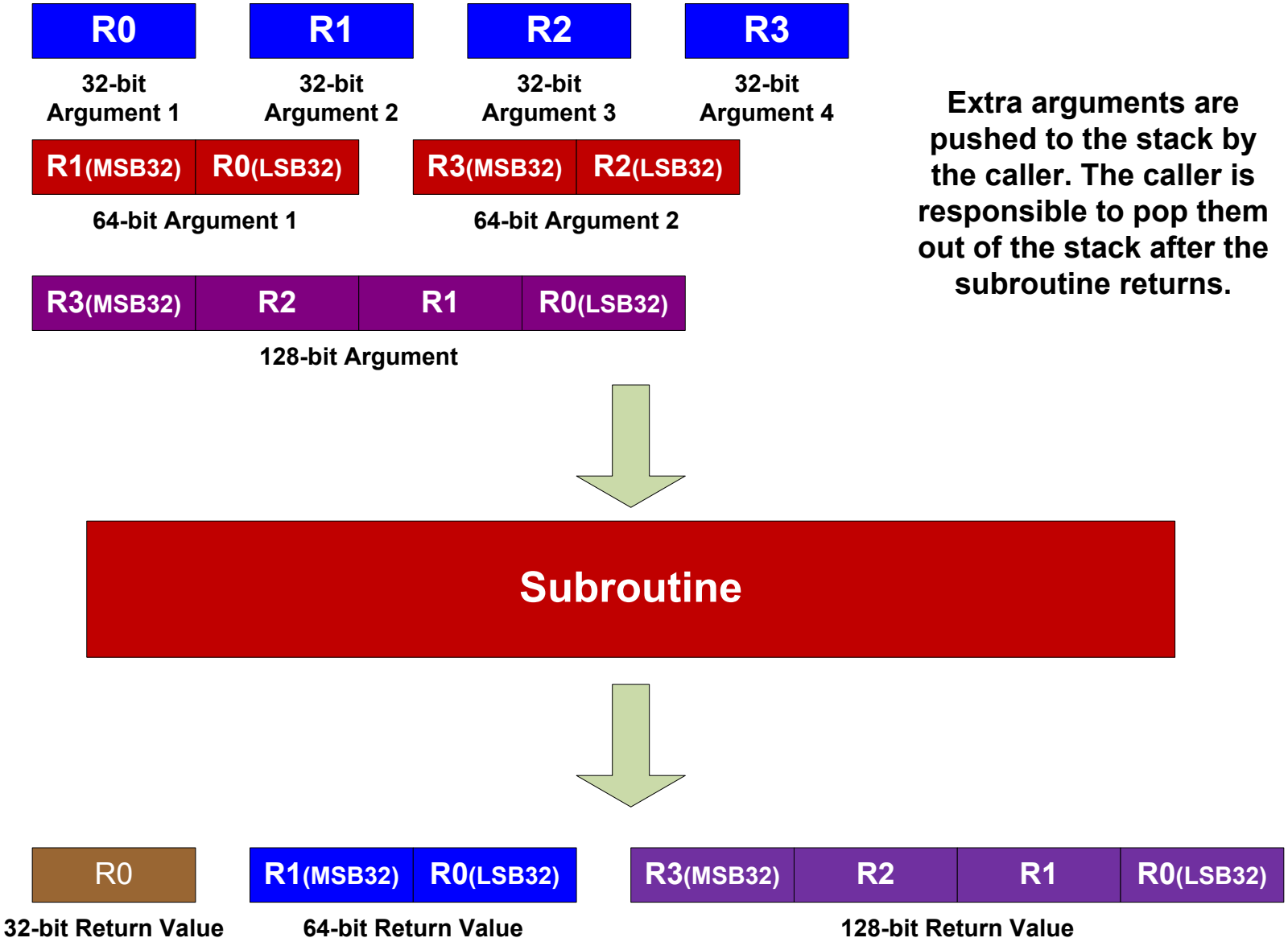
BL and BX


void enable(void) ;
enable() ;
Compiler
BL enable
.global enable
enable:
BX LR

ARM Calling Convention

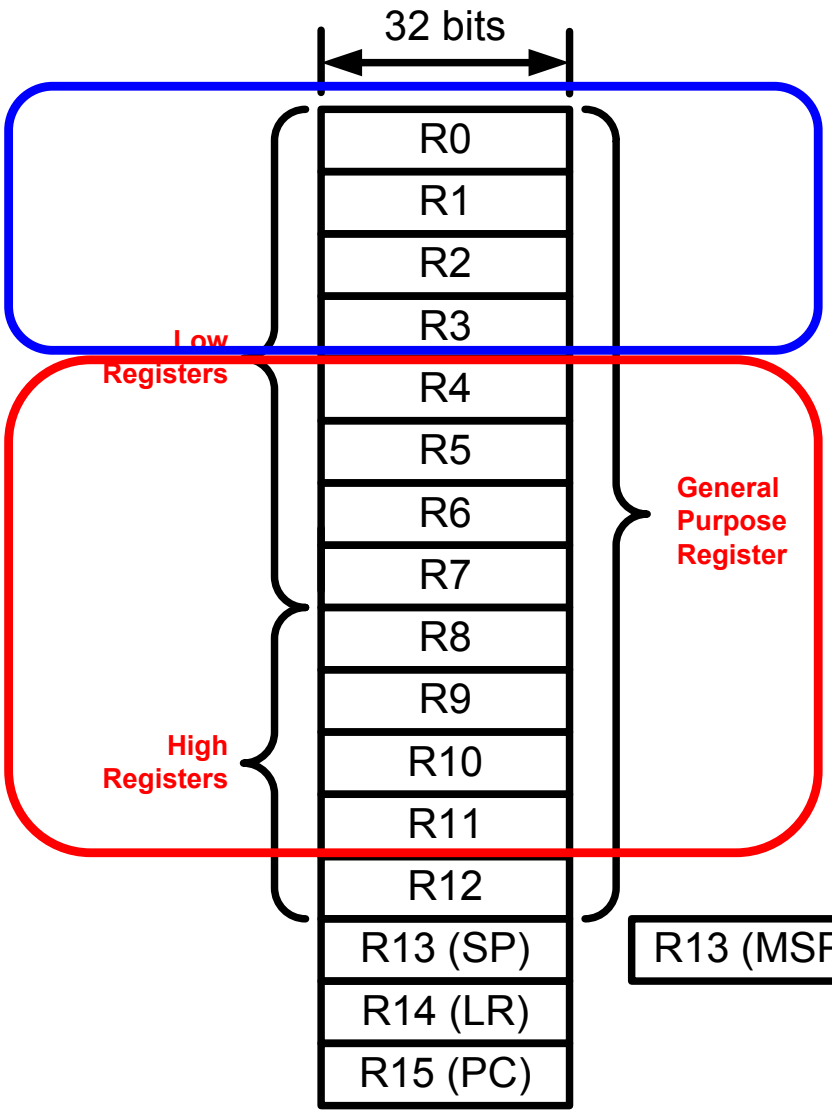
Register	Usage	Subroutine Preserved	Notes
r0	Argument 1 and return value	No	If return has 64 bits, then r0:r1 hold it. If argument 1 h as 64 bits, r0:r1 hold it.
r1	Argument 2	No	
r2	Argument 3	No	If the return has 128 bits, r0-r3 hold it.
r3	Argument 4	No	If more than 4 arguments, use the stack
r4	General-purpose 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 subr outine 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.

Caller Saved // Callee Saved registers

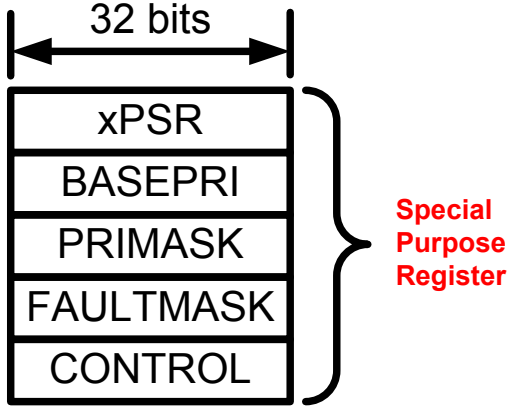


Not saved. Hold arguments, results, or temporary values. Caller doesn't expect them to be retained.

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

R13 (MSP)

R13 (PSP)



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
        ...
```

R1: second argument

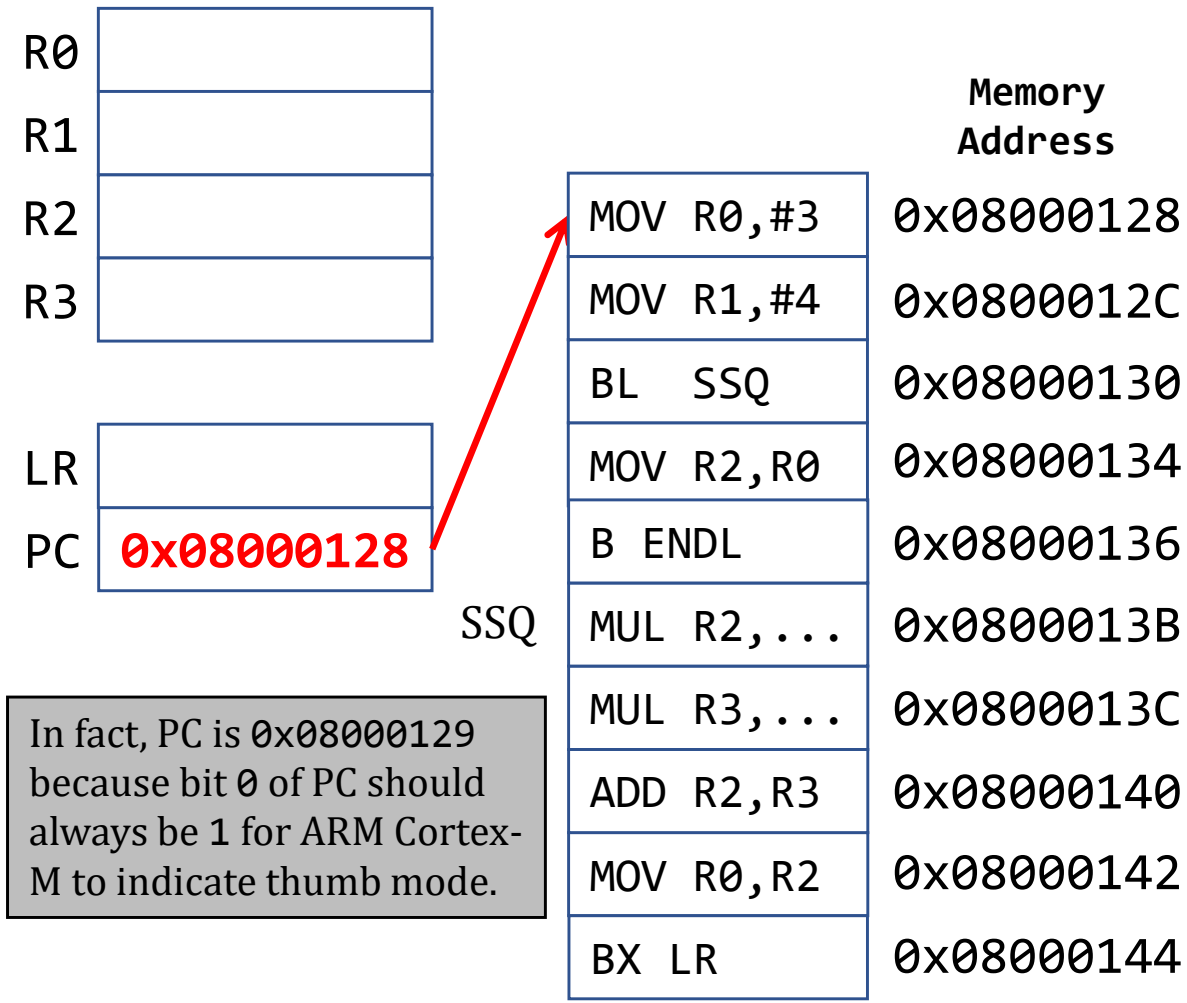
R0: first argument

```
int SSQ(int x, int y){
      int z;
      z = x*x + y * y;
      return z;
}
```

R0: Return Value

Example: R2 = R0*R0+R1*R1

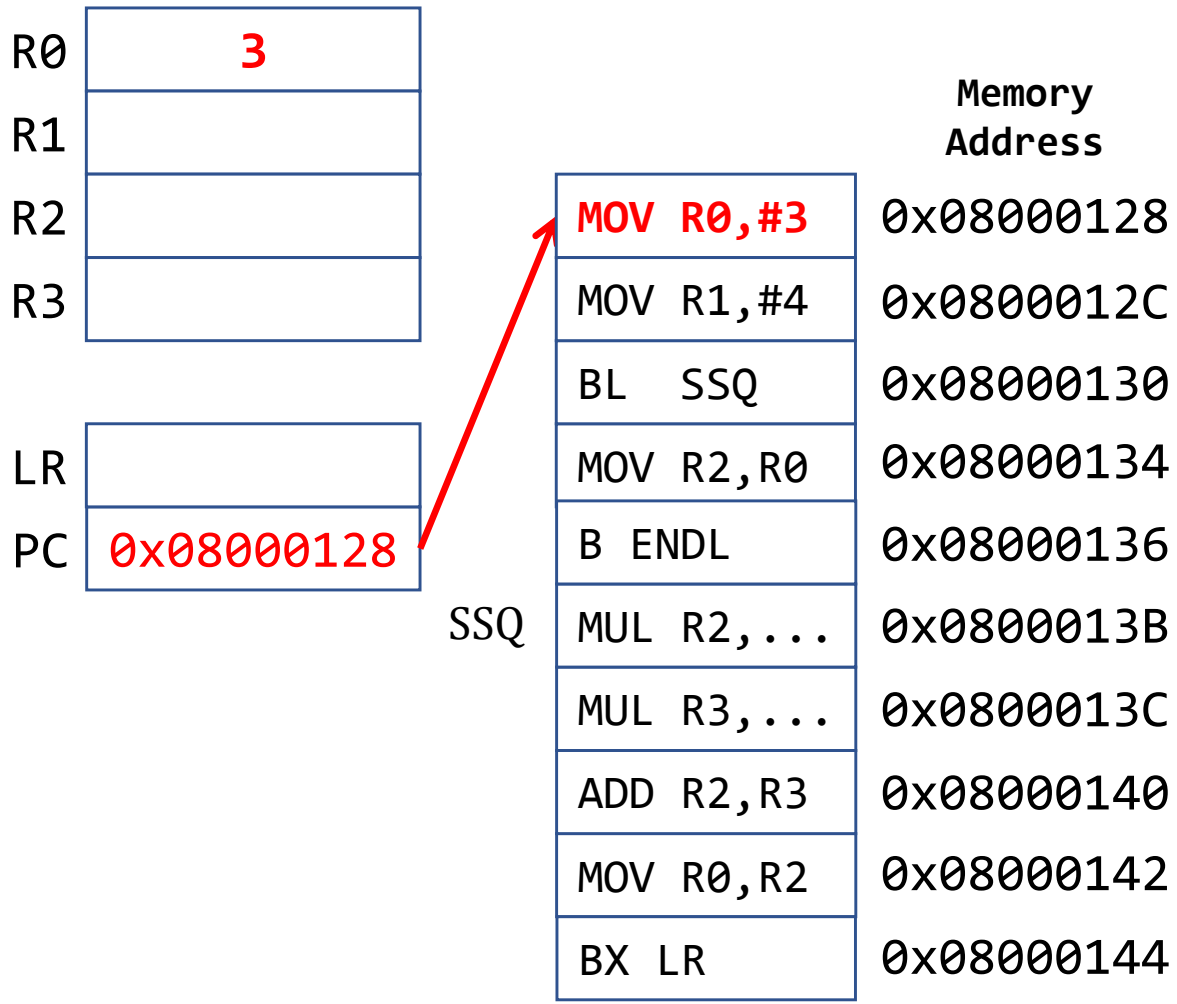
```
        MOV R0,#3
        MOV R1,#4
        BL  SSQ
        MOV R2,R0
        B ENDL
        ...
SSQ:
        MUL R2,R0,R0
        MUL R3,R1,R1
        ADD R2,R2,R3
        MOV R0,R2
        BX  LR
ENDL:   ...
```



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

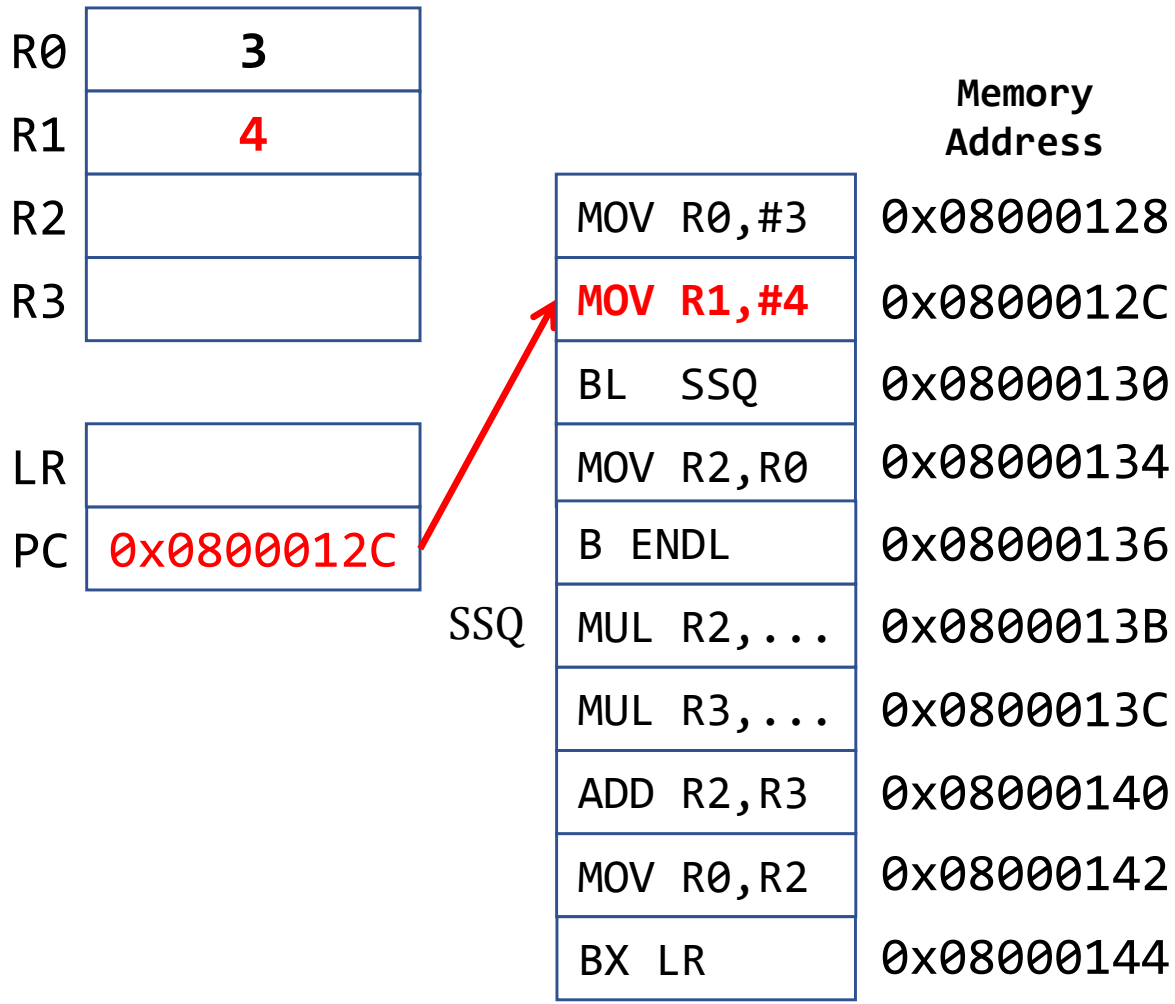
Example: R2 = R0*R0+R1*R1

```
        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:   ...
```



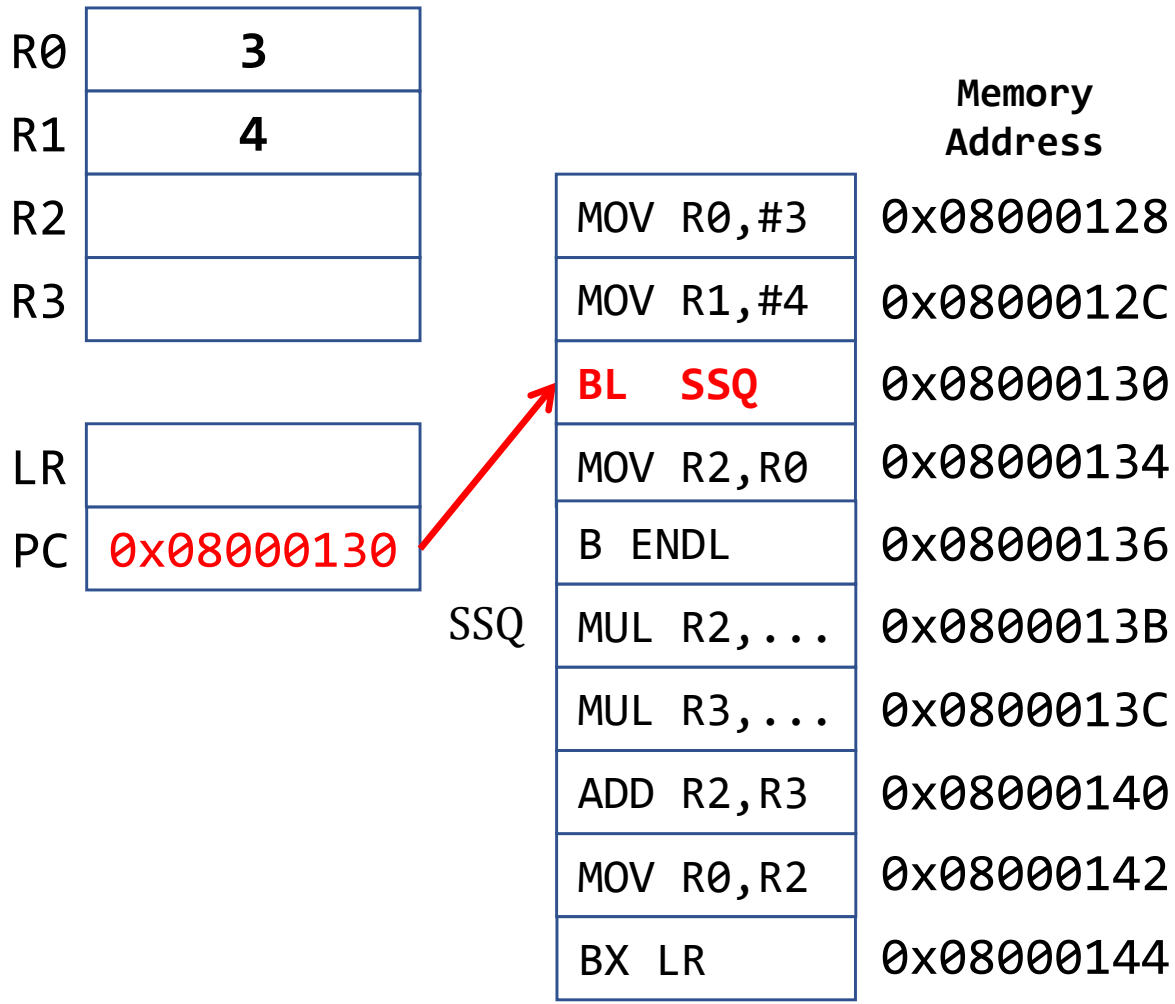
Example: R2 = R0*R0+R1*R1

```
        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:   ...
```



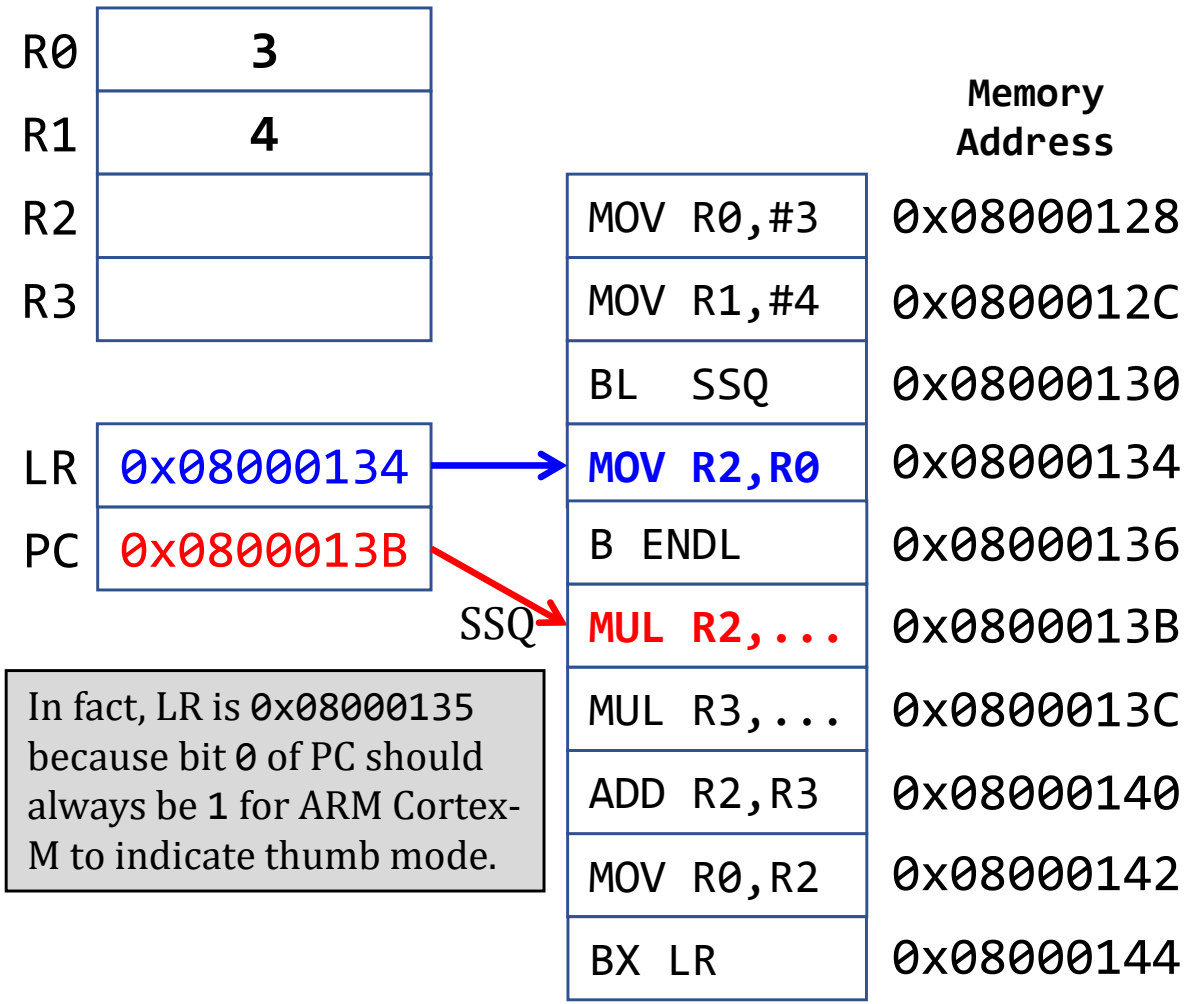
Example: R2 = R0*R0+R1*R1

```
        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:   ...
```



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:   ...
```

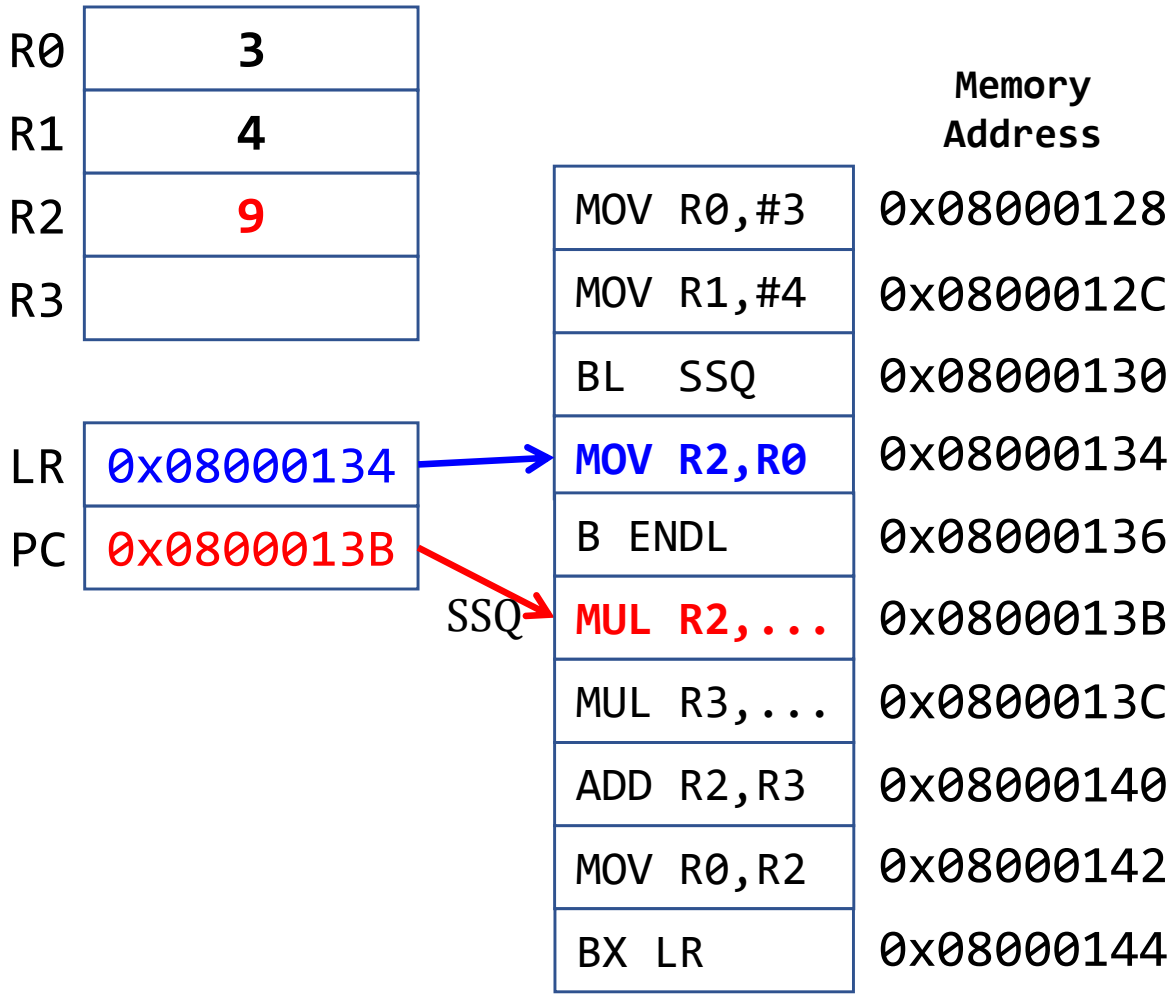


In fact, LR is 0x08000135 because bit 0 of PC should always be 1 for ARM Cortex-M to indicate thumb mode.

Address of the next instruction after the branch is saved into

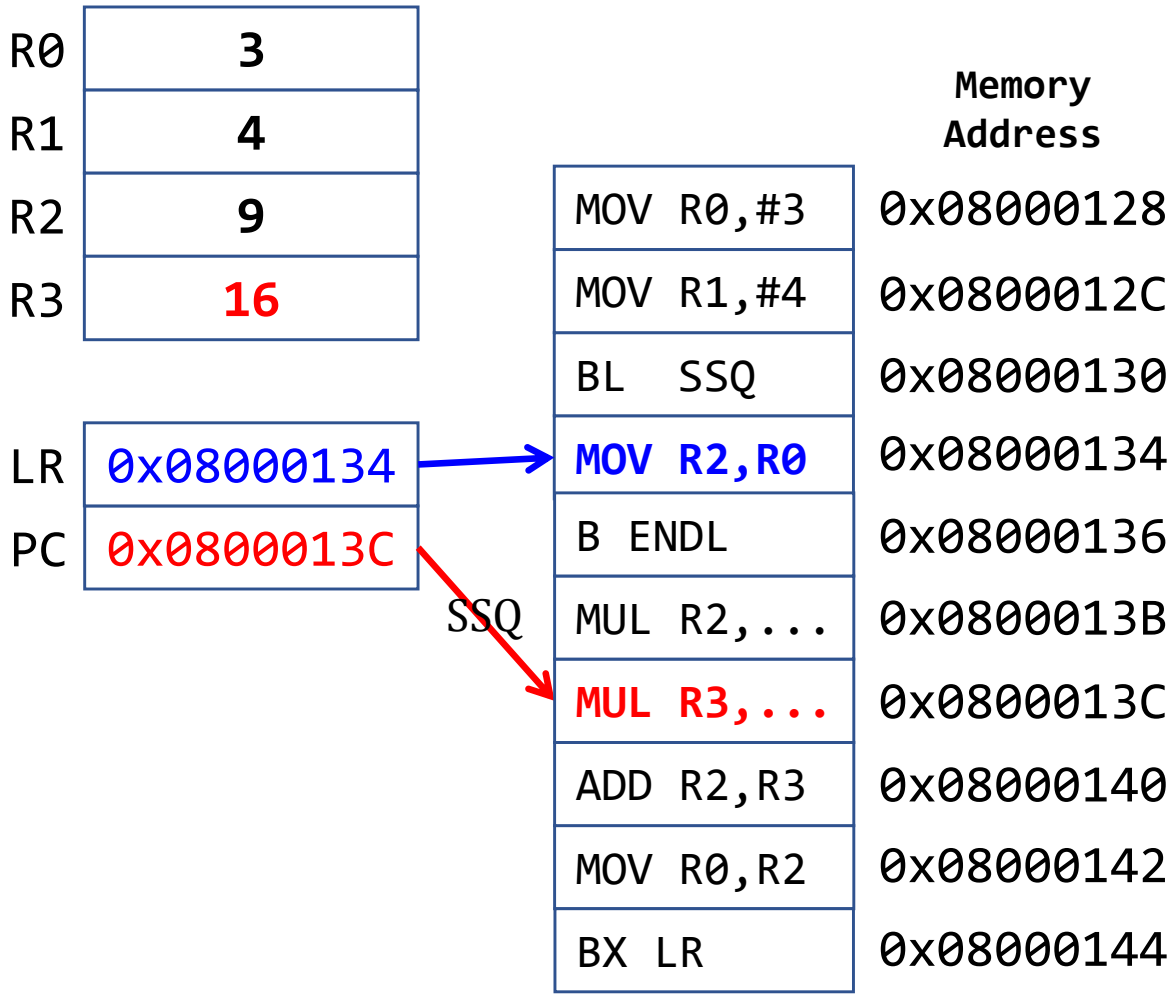
Example: R2 = R0*R0+R1*R1

```
        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:   ...
```



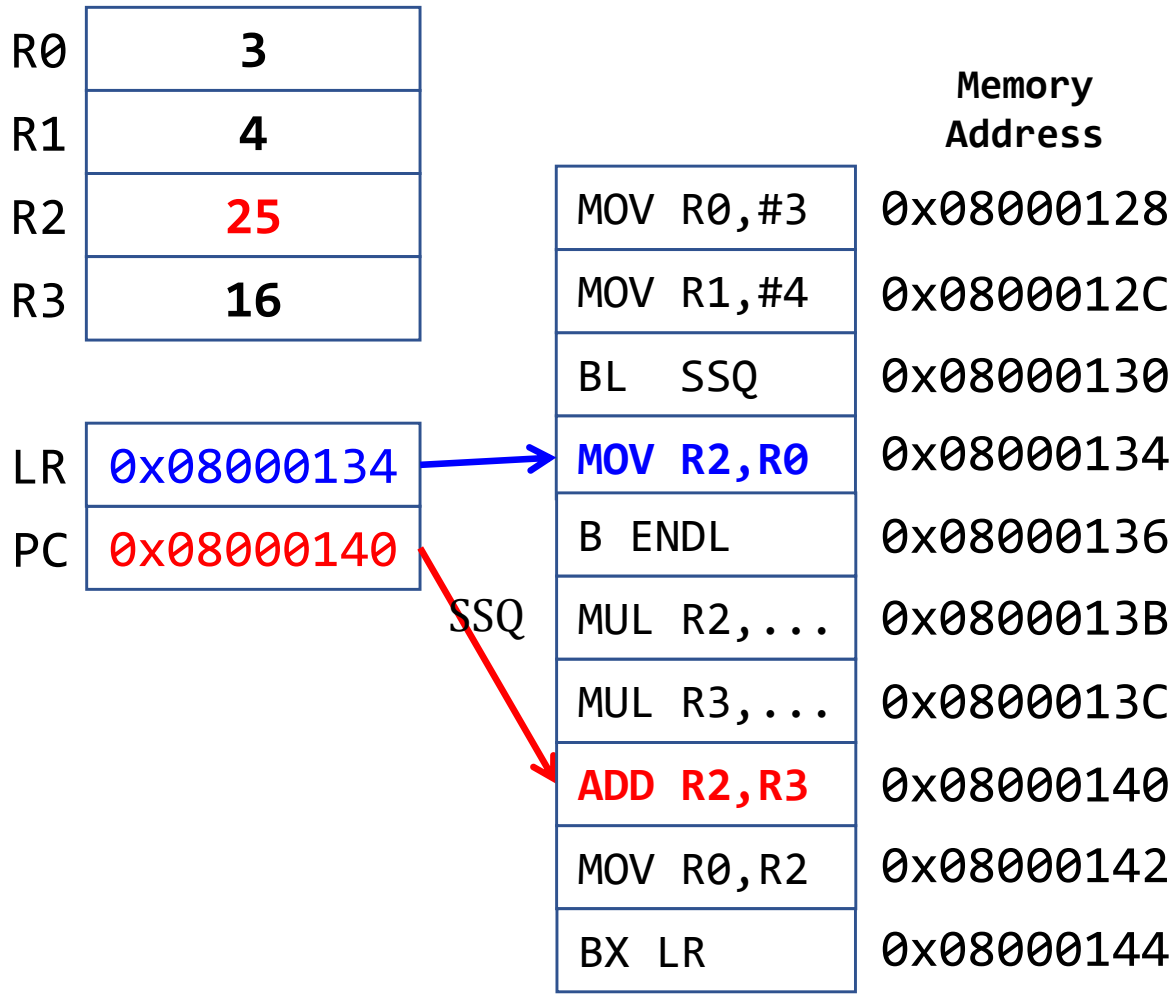
Example: R2 = R0*R0+R1*R1

```
        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:   ...
```



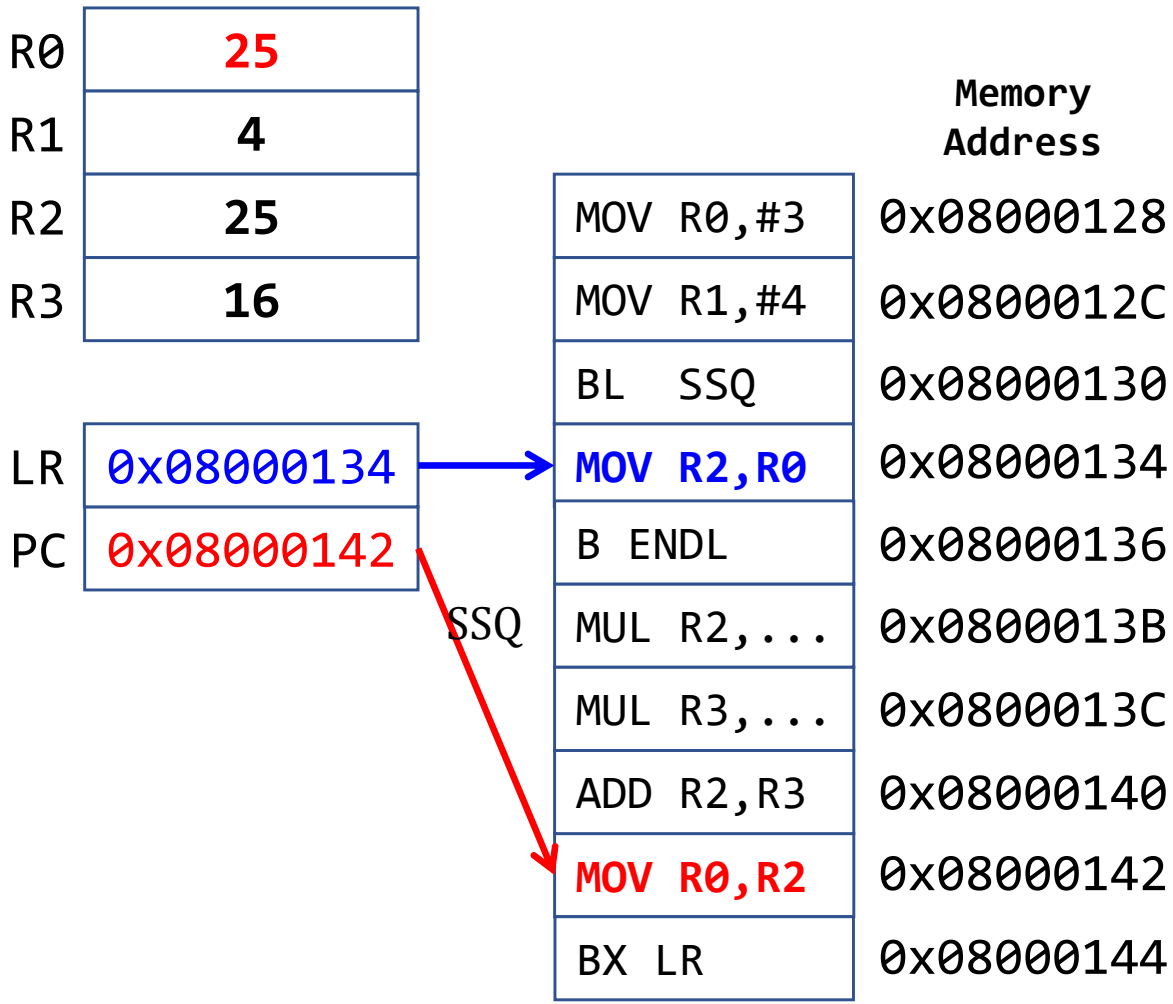
Example: R2 = R0*R0+R1*R1

```
        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:   ...
```



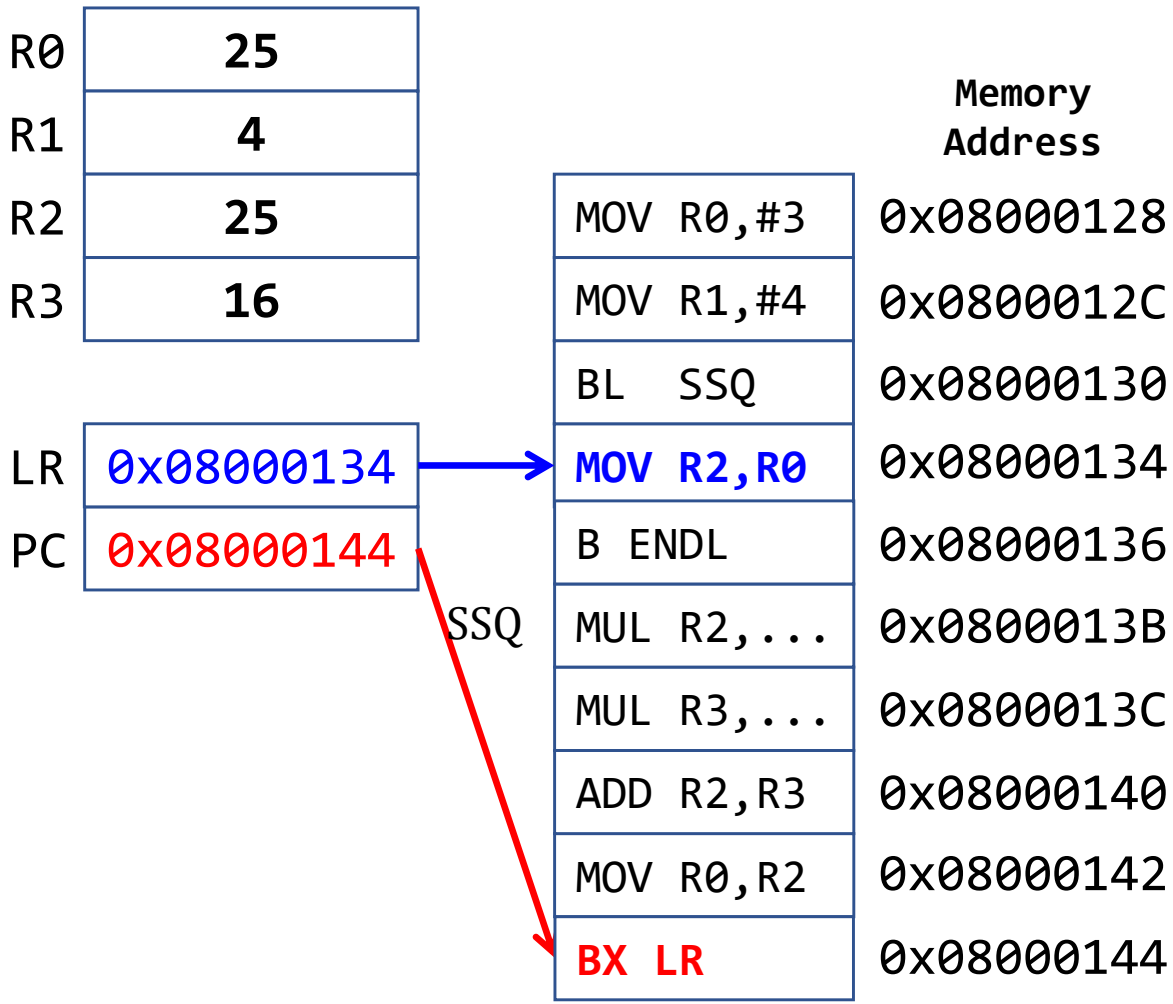
Example: R2 = R0*R0+R1*R1

```
        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:   ...
```



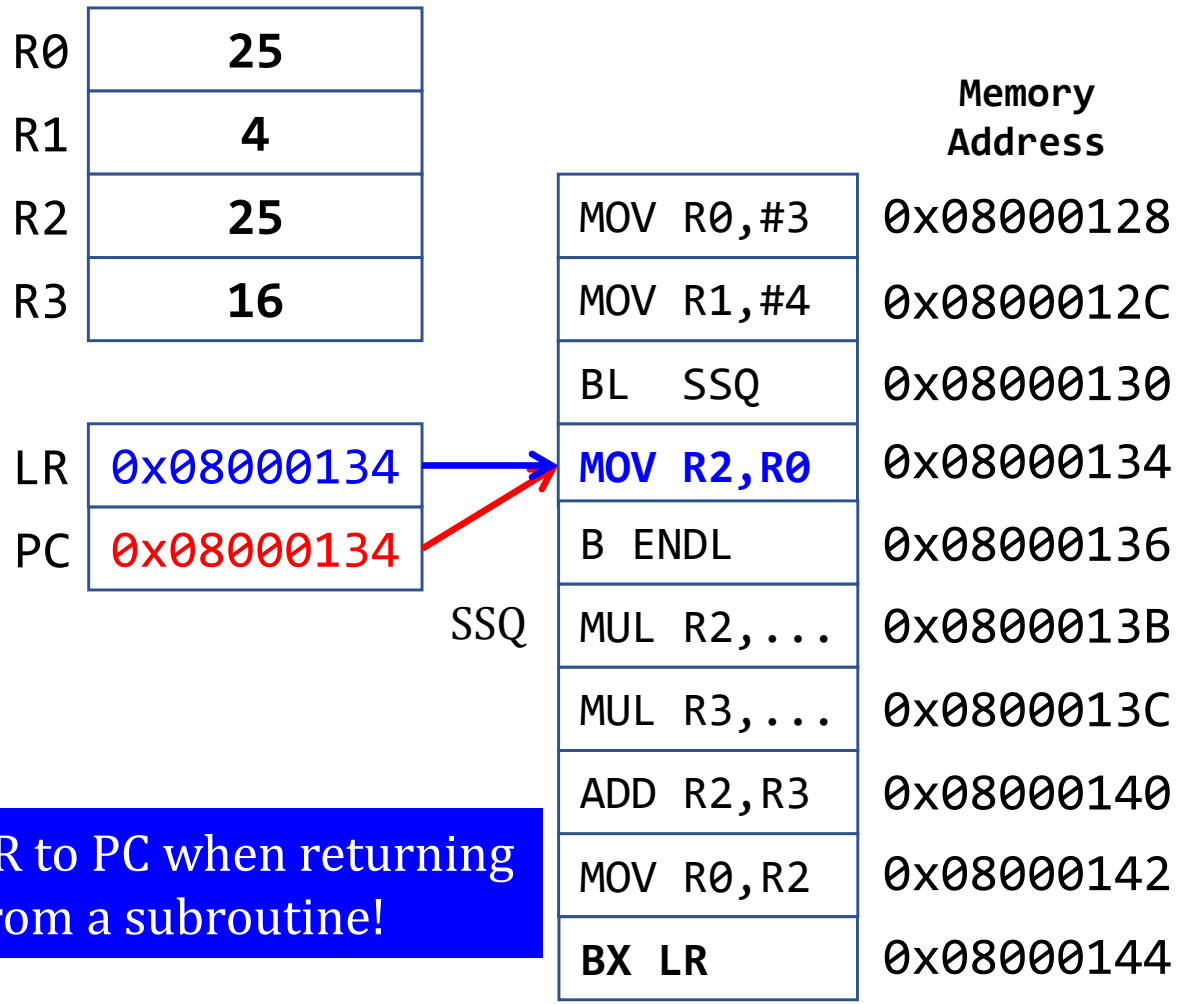
Example: R2 = R0*R0+R1*R1

```
        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:   ...
```



Example: R2 = R0*R0+R1*R1

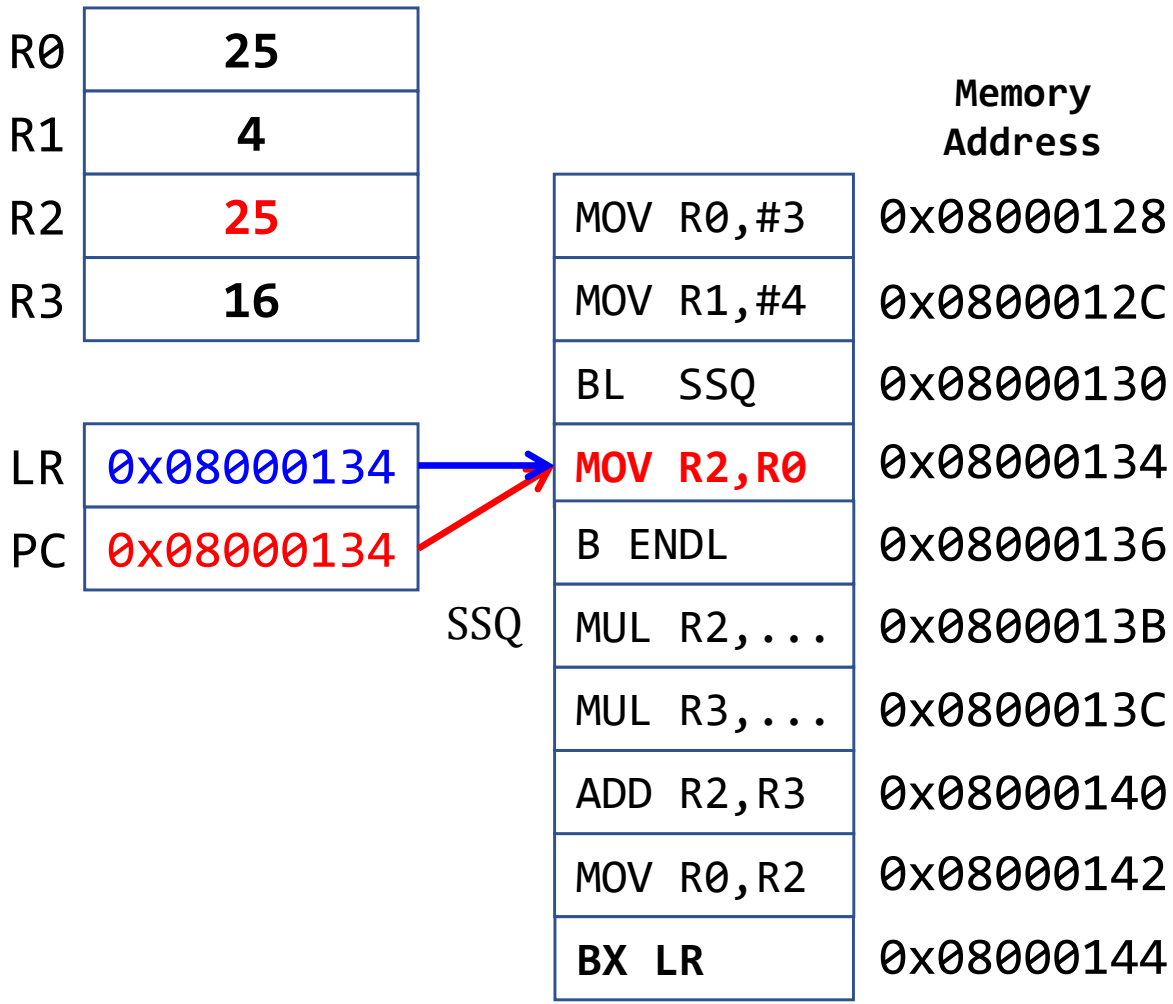
```
        MOV R0,#3
        MOV R1,#4
        BL  SSQ
        MOV R2,R0
        B ENDL
        ...
SSQ:
        MUL R2,R0,R0
        MUL R3,R1,R1
        ADD R2,R2,R3
        MOV R0,R2
        BX  LR
ENDL:   ...
```



Copy LR to PC when returning from a subroutine!

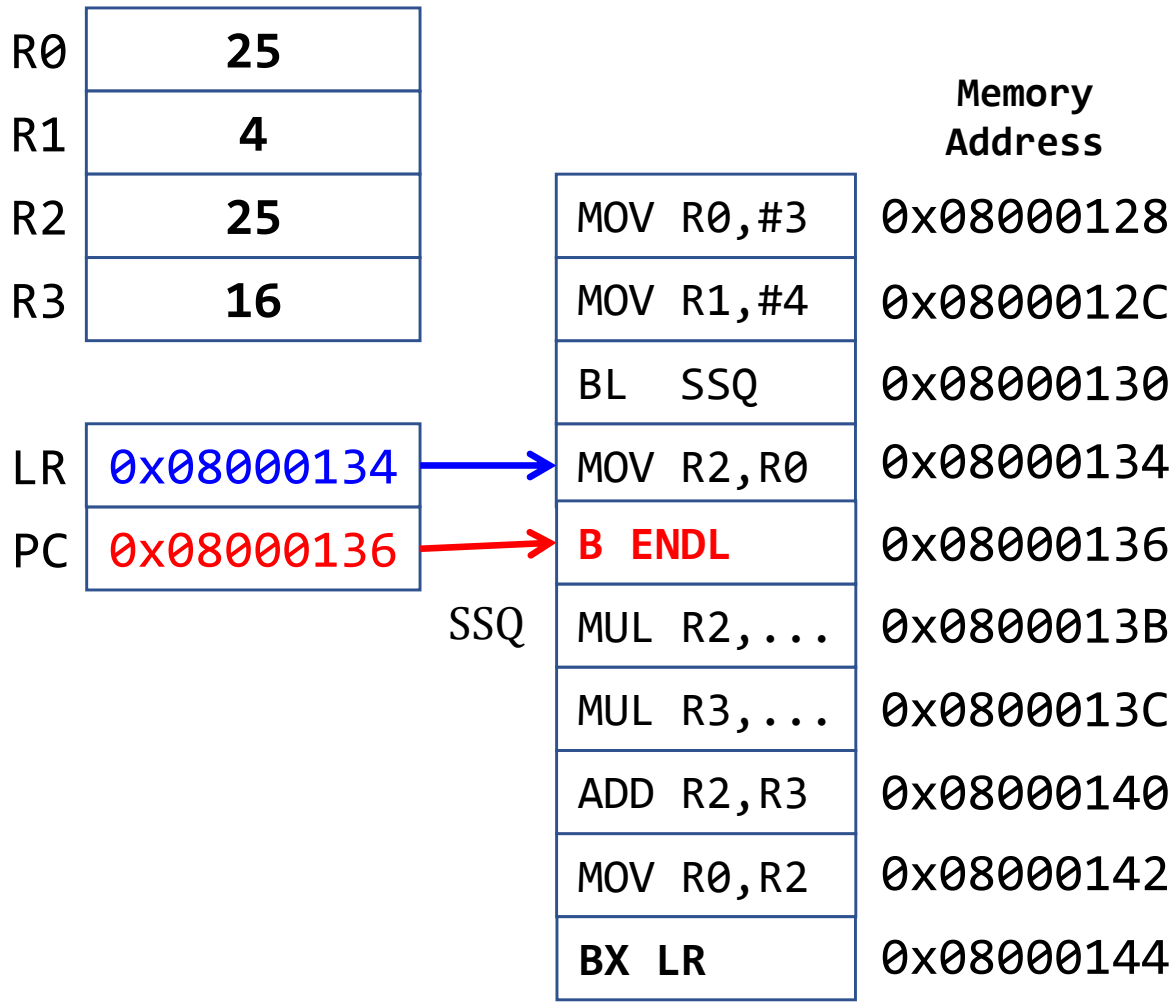
Example: R2 = R0*R0+R1*R1

```
        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:   ...
```



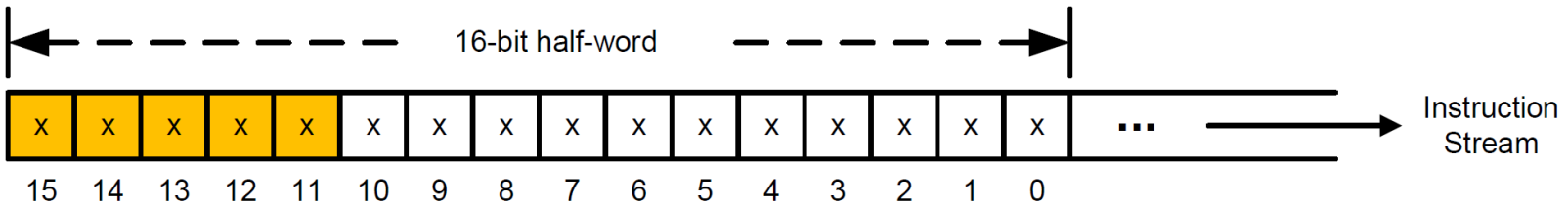
Example: R2 = R0*R0+R1*R1

```
        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:   ...
```



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

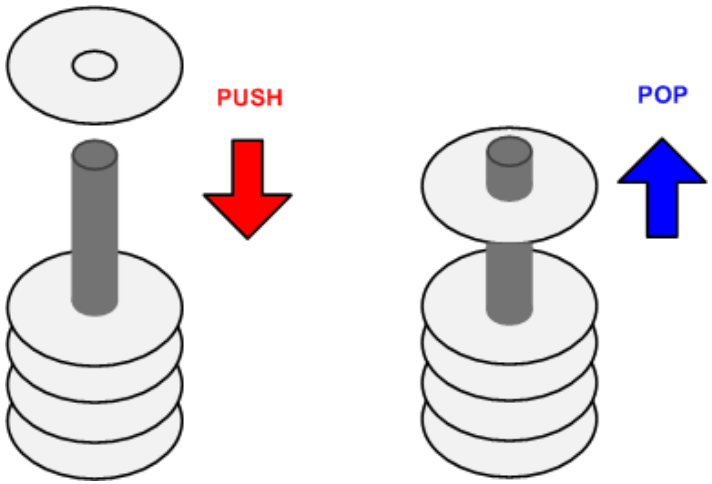


If bit [15-11] = `11101`, `11110`, or `11111`, then, it is the first half-word of a 32-bit instruction. Otherwise, it is a 16-bit instruction.

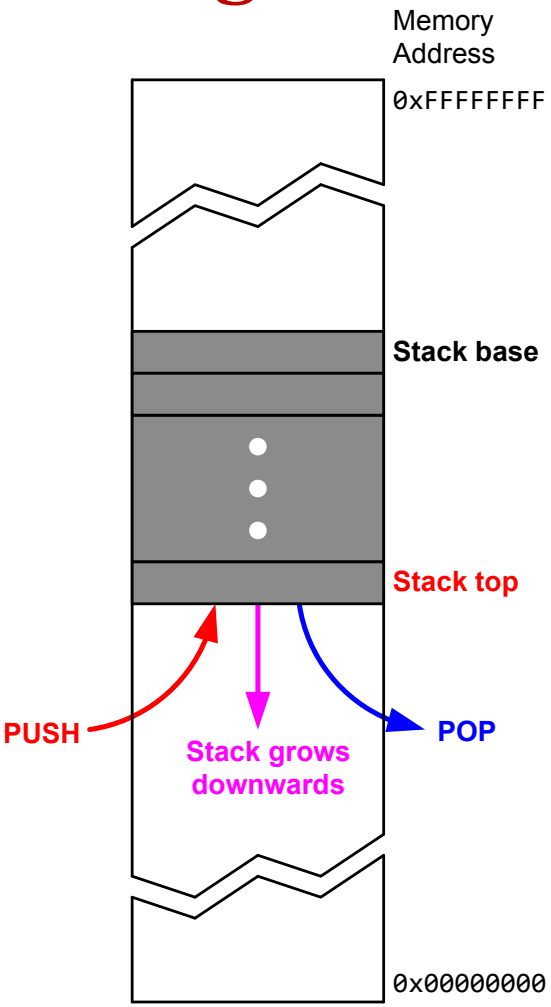
Preserve Environment via Stack

Stack

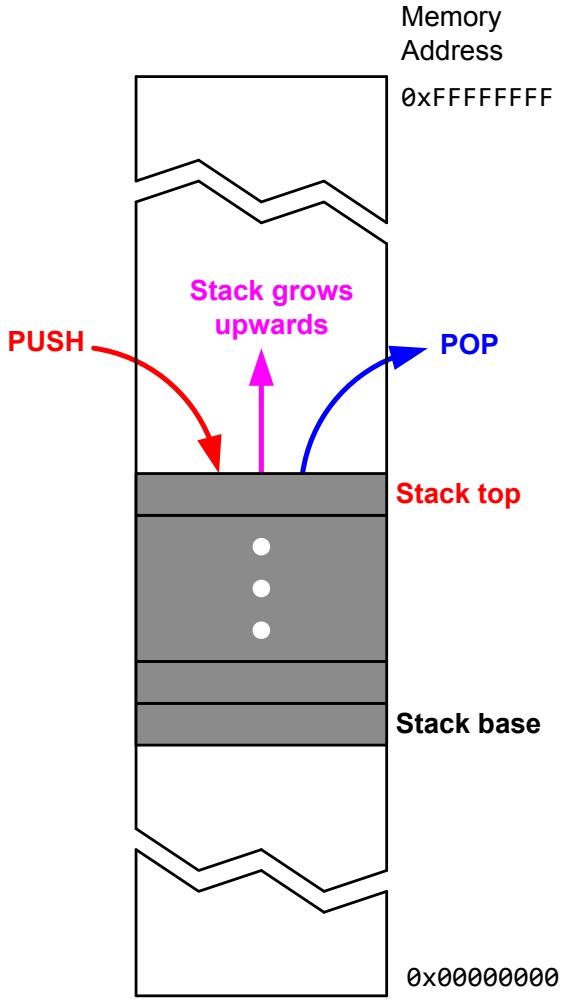
- A **Last-In-First-Out** data structure
- Only allow to access the most recently added item
  - Also called the top of the stack
- Key operations:
  - push (add item to stack)
  - pop (remove top item from stack)



Stack Growth Convention:
Ascending *vs* Descending

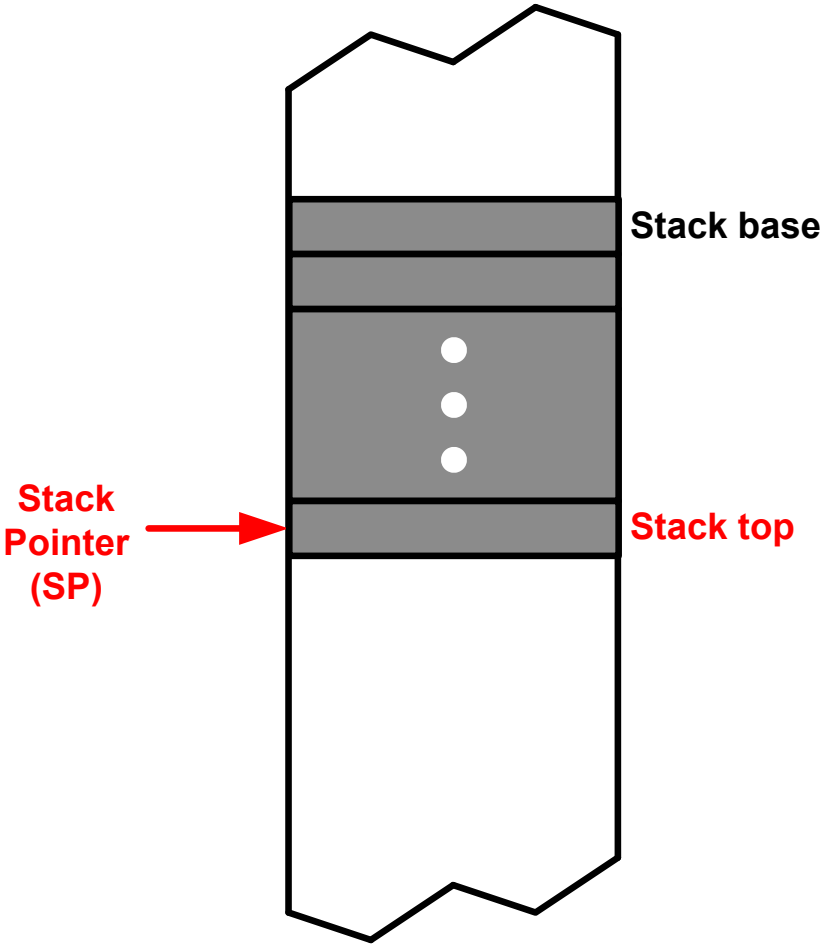


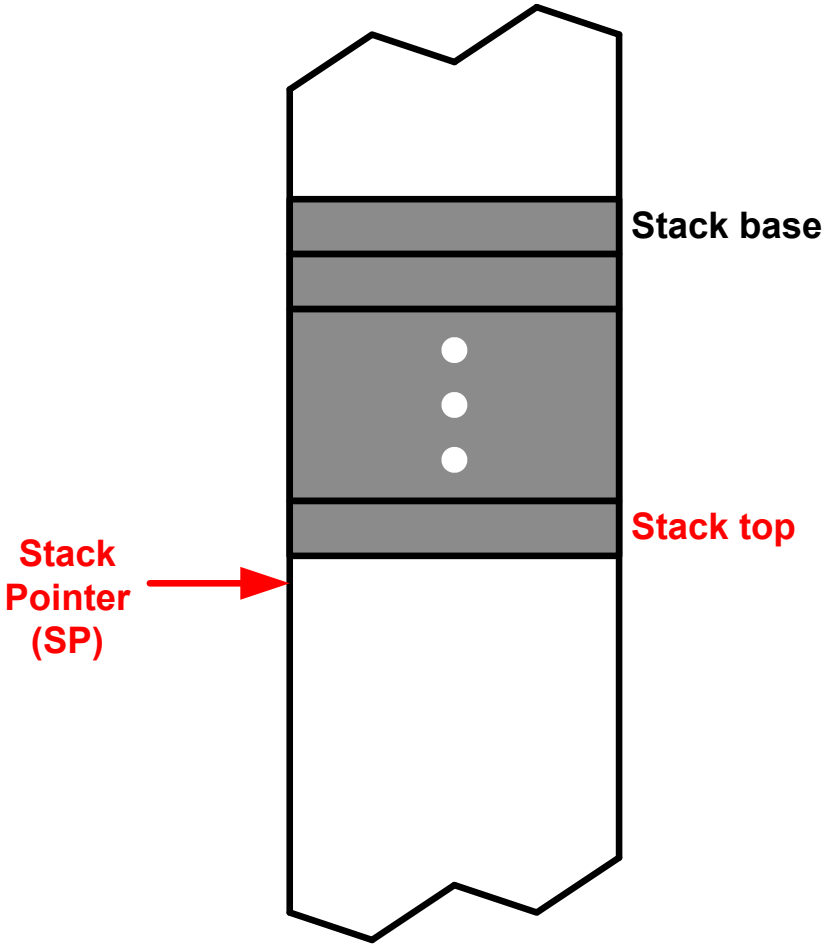
Descending stack: Stack grows towards low memory address



Ascending stack: Stack grows towards high memory address

Stack Growth Convention:
Full *vs* Empty



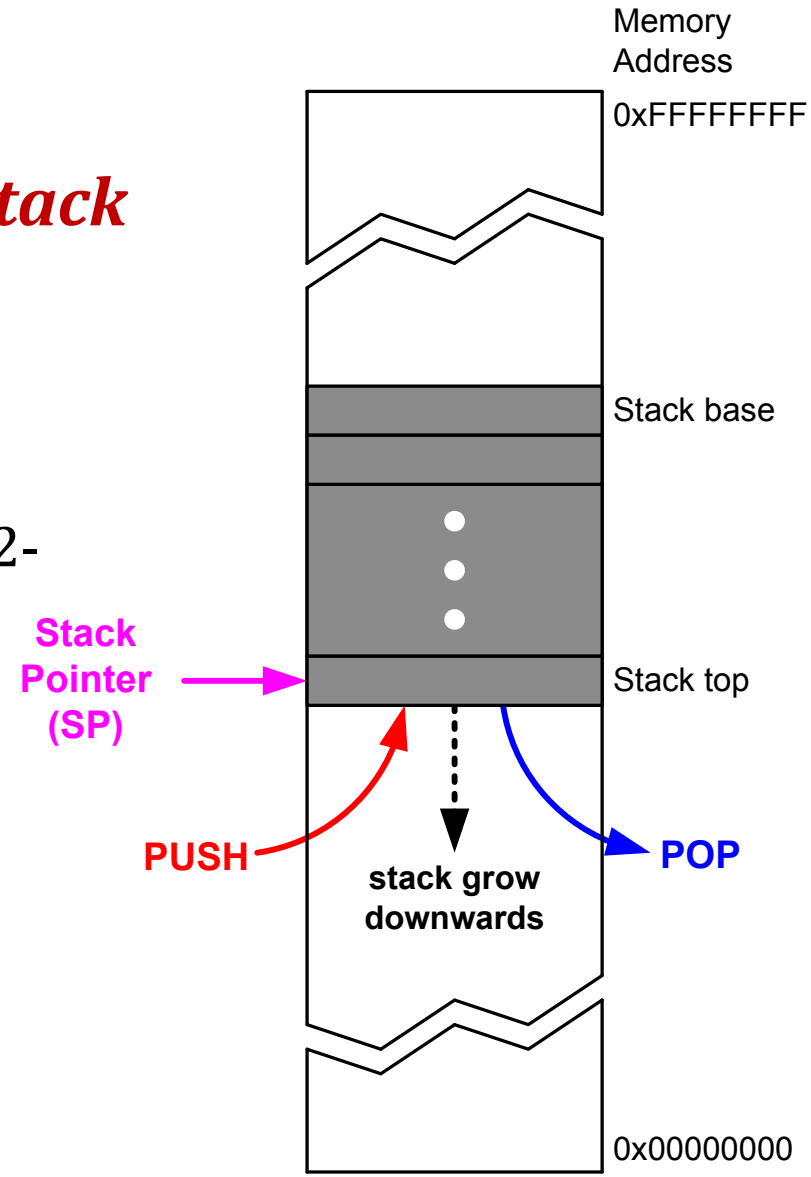


Full stack: SP points to the last item pushed onto the stack

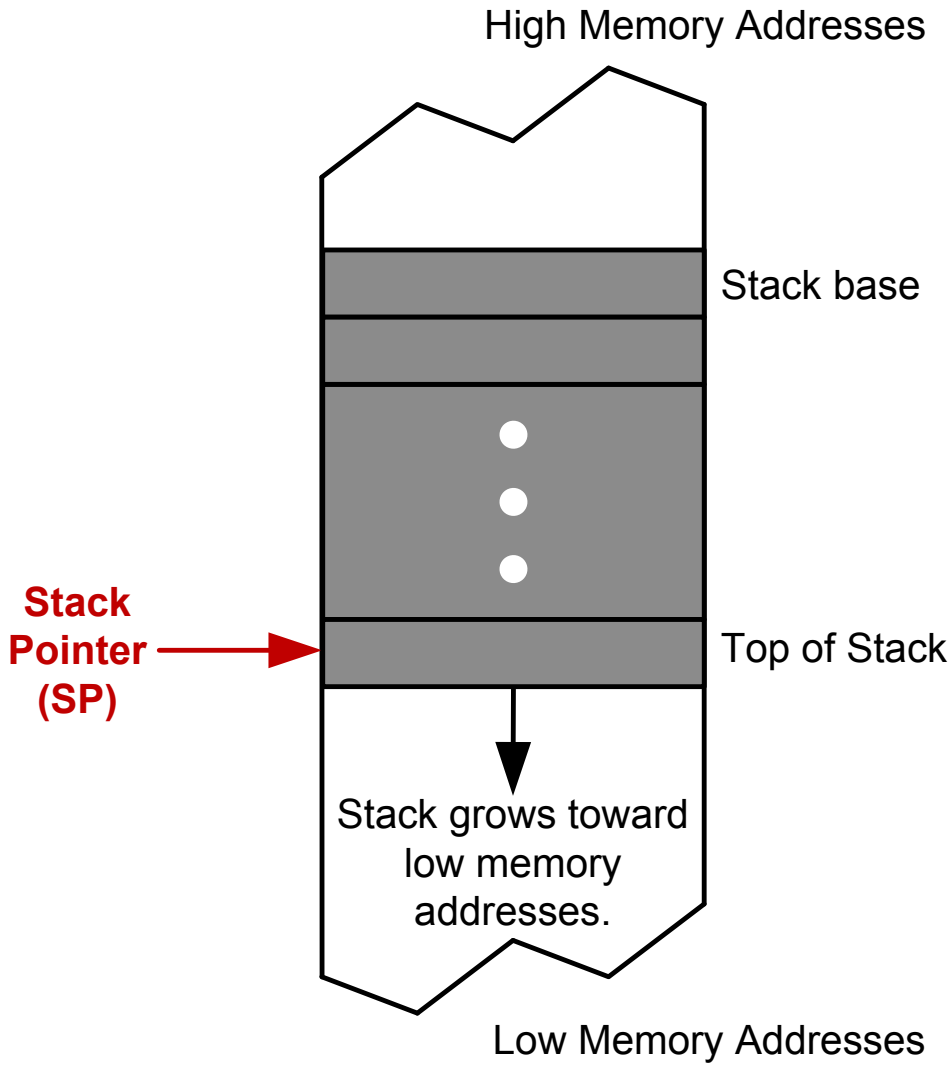
Empty stack: SP points to the next free space on the stack

Cortex-M Stack

- stack pointer (SP) = R13
- Cortex-M uses ***full descending stack***
- stack pointer
  - decremented on **PUSH**
  - incremented on **POP**
  - SP starts at **`0x20000200`** for STM32-Discovery



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 ⟶ 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

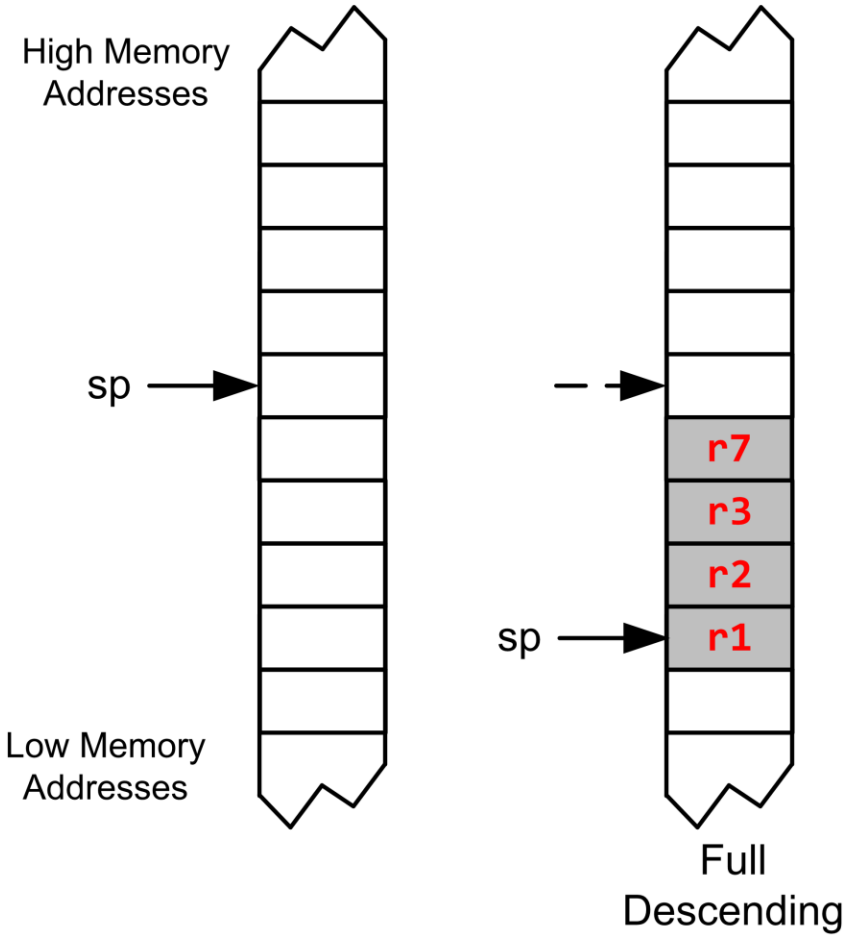
They are equivalent.

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

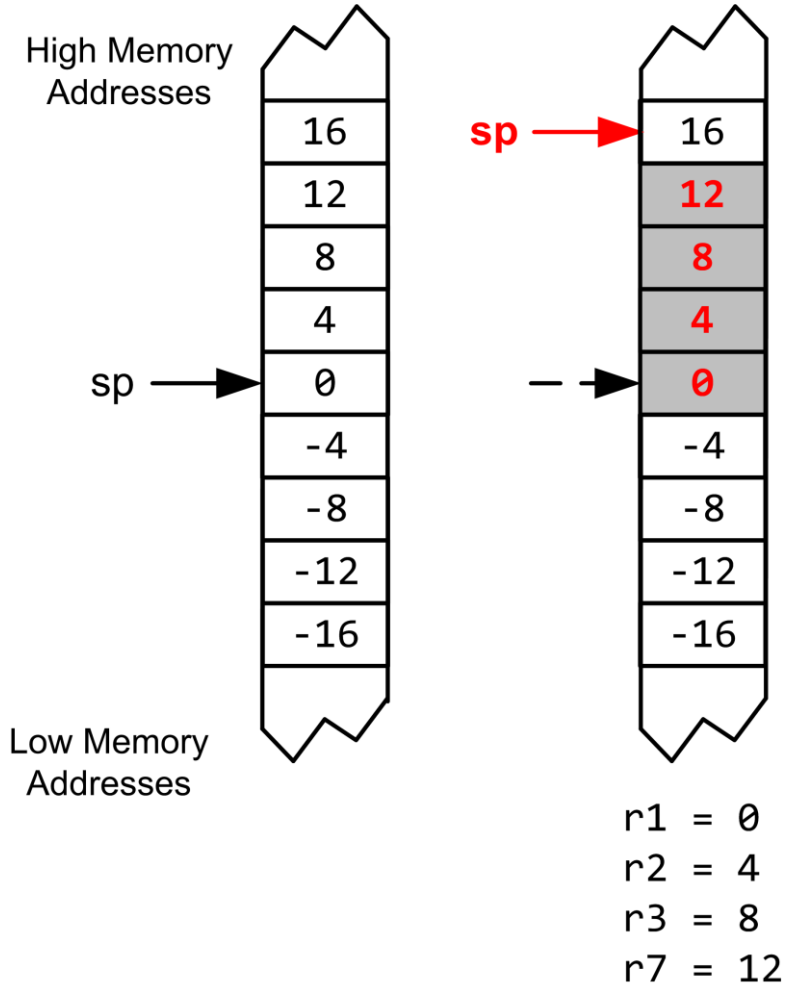
- The order in which registers listed in the register list does not matter.
- When popping multiple registers, these registers are automatically sorted by name and the lowest-numbered register is loaded from the lowest memory address, *i.e.* is loaded first.

Full Descending Stack

PUSH {r3, r1, r7, r2}



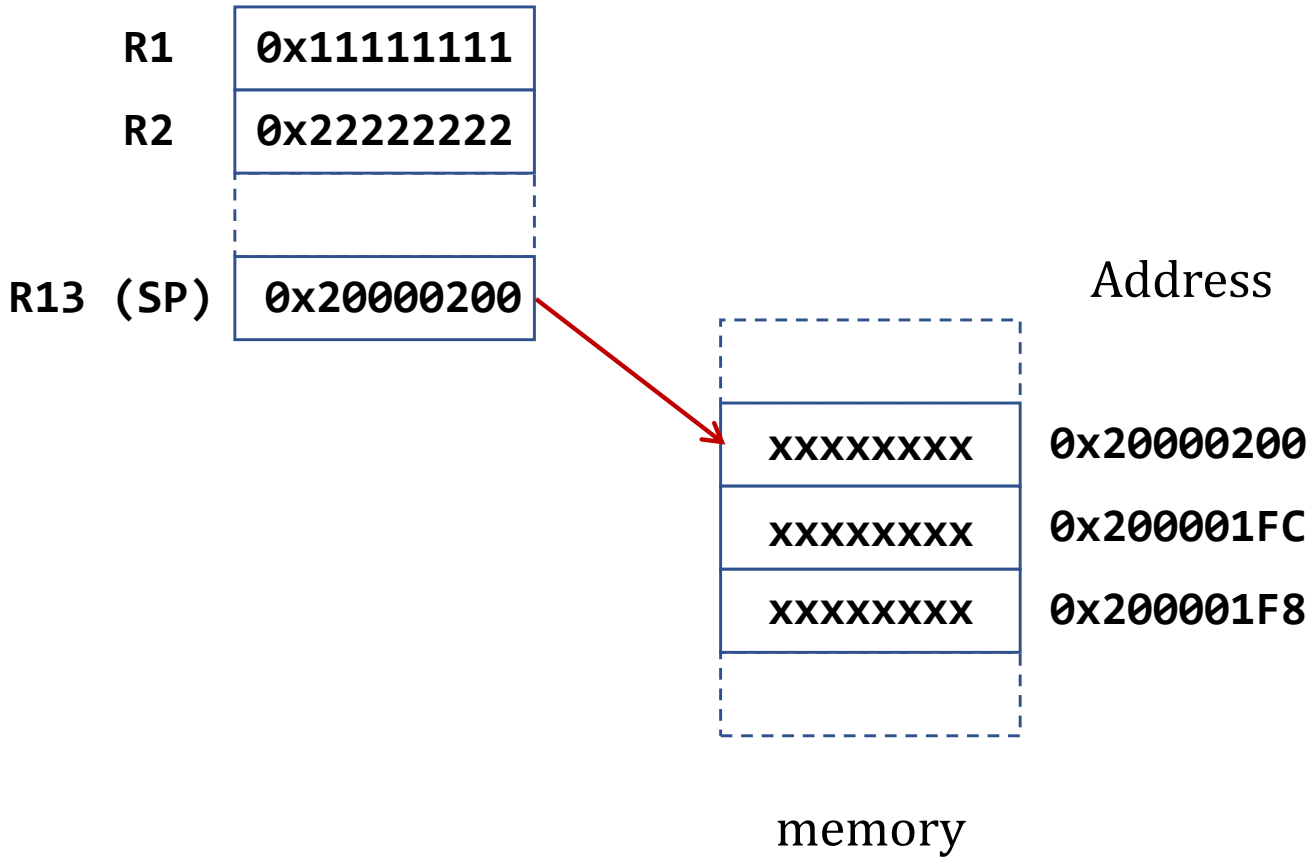
POP {r3, r1, r7, r2}



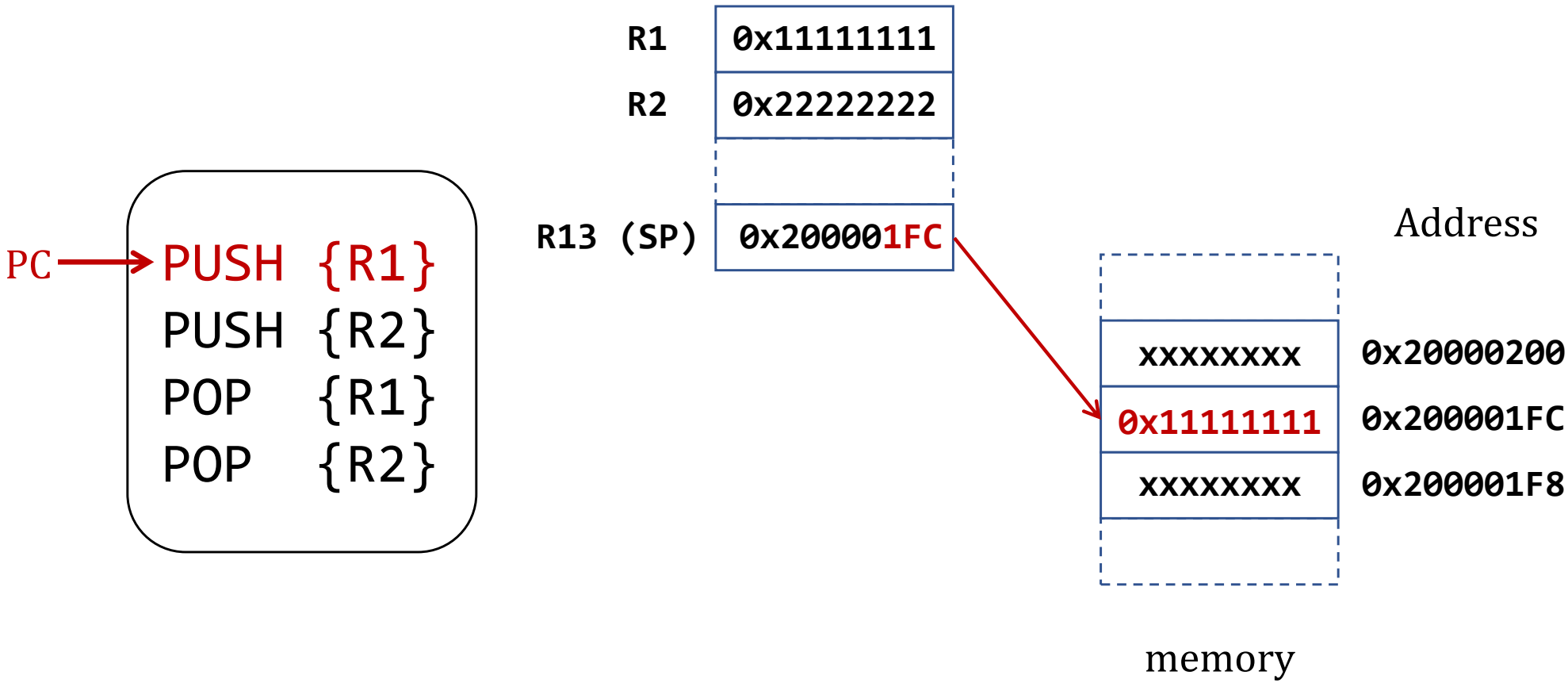
r1 = 0
r2 = 4
r3 = 8
r7 = 12

Example: swap R1 & R2

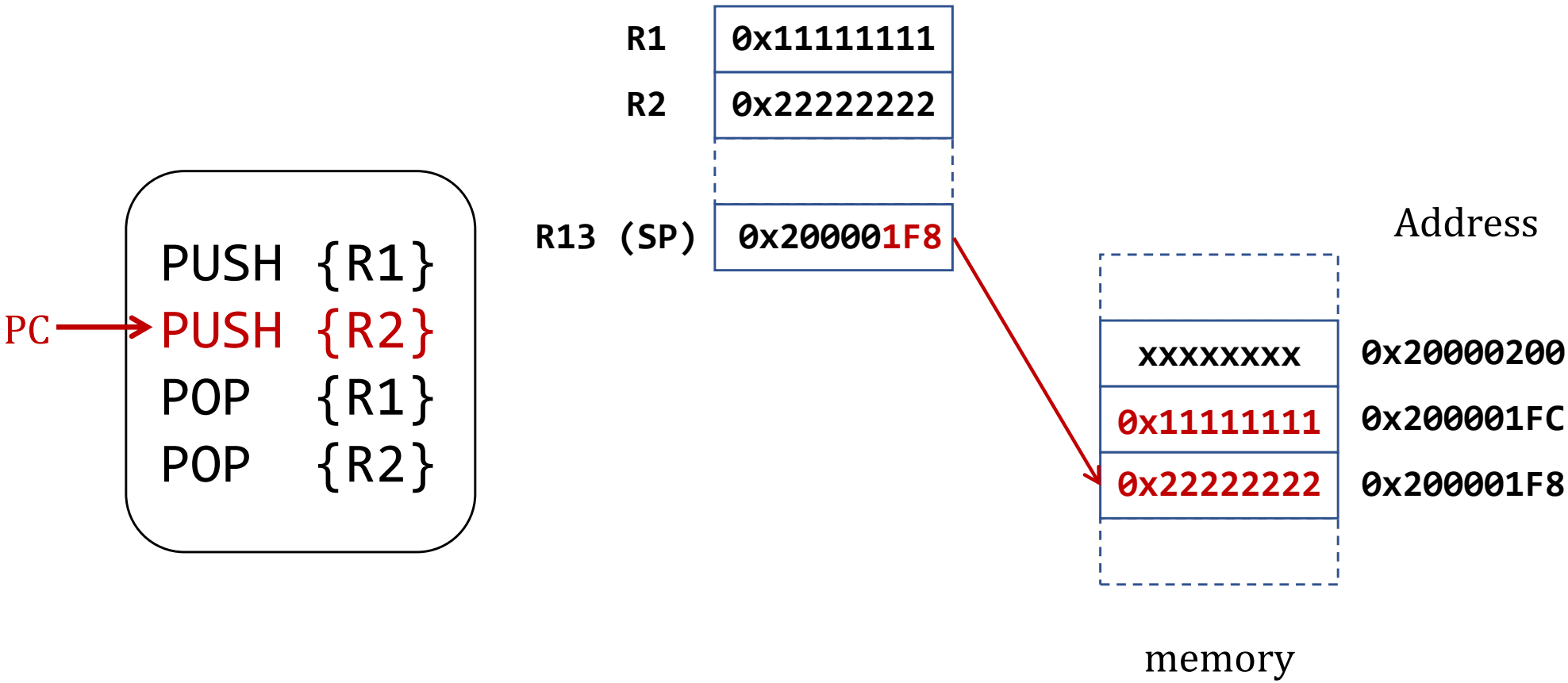
```
PUSH {R1}
PUSH {R2}
POP  {R1}
POP  {R2}
```



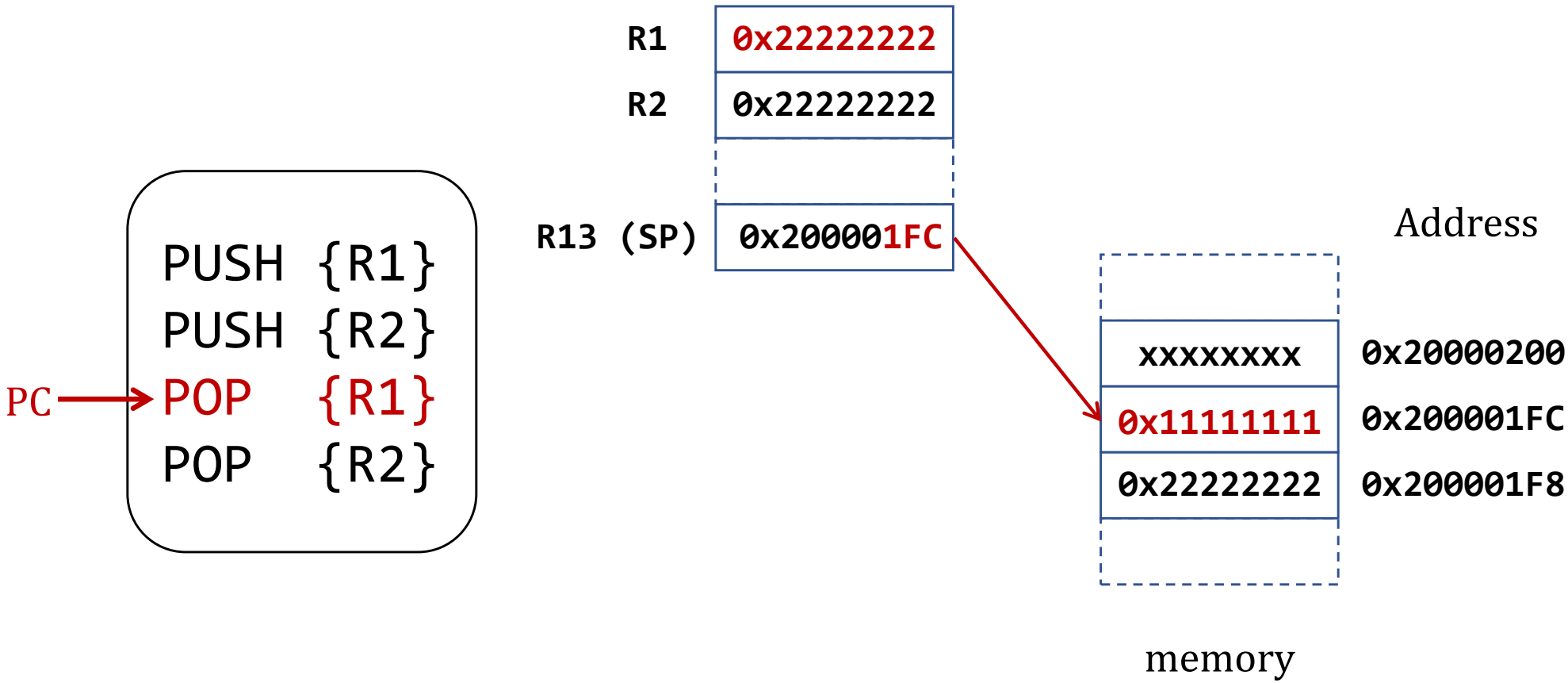
Example: swap R1 & R2



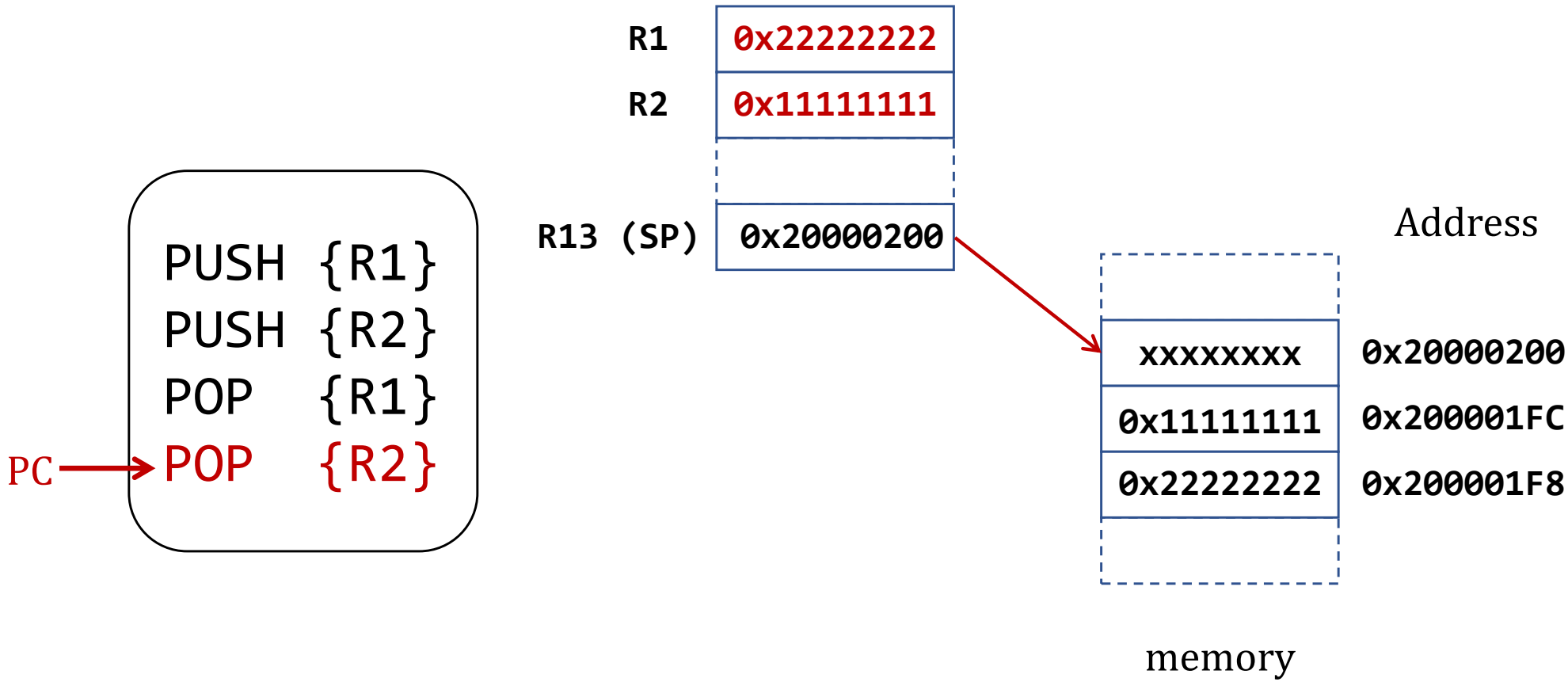
Example: swap R1 & R2



Example: swap R1 & R2



Example: swap R1 & R2

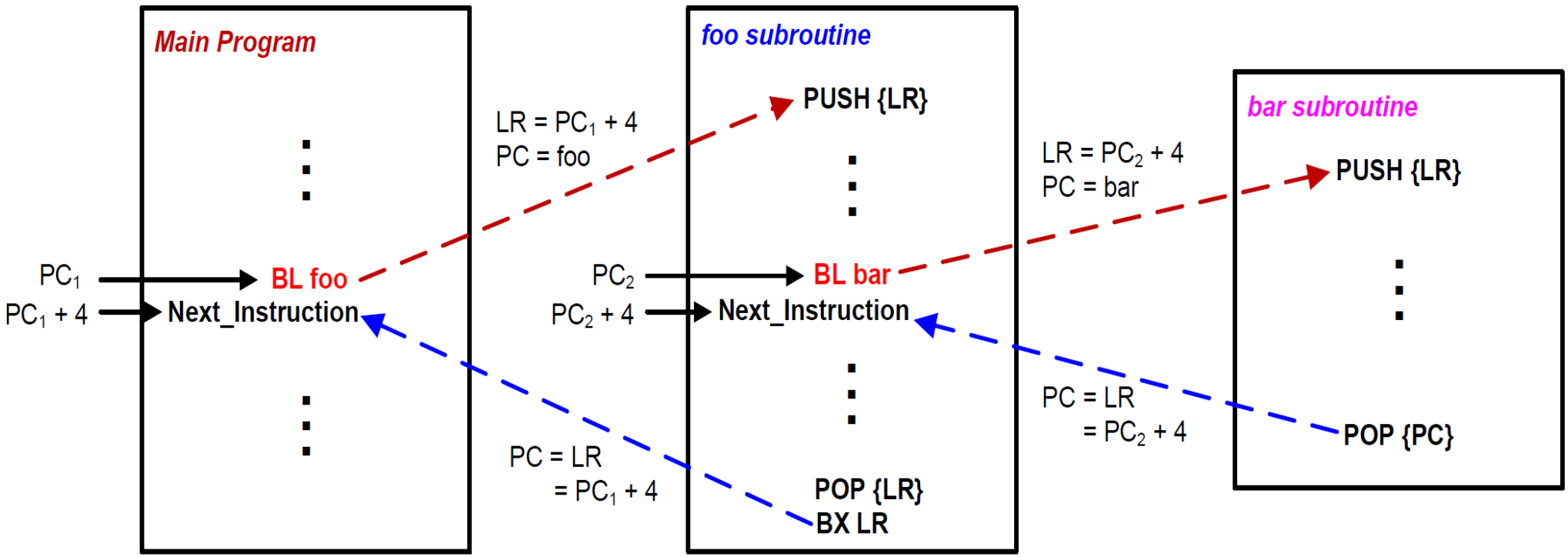


Preserve Runtime Environment via Stack

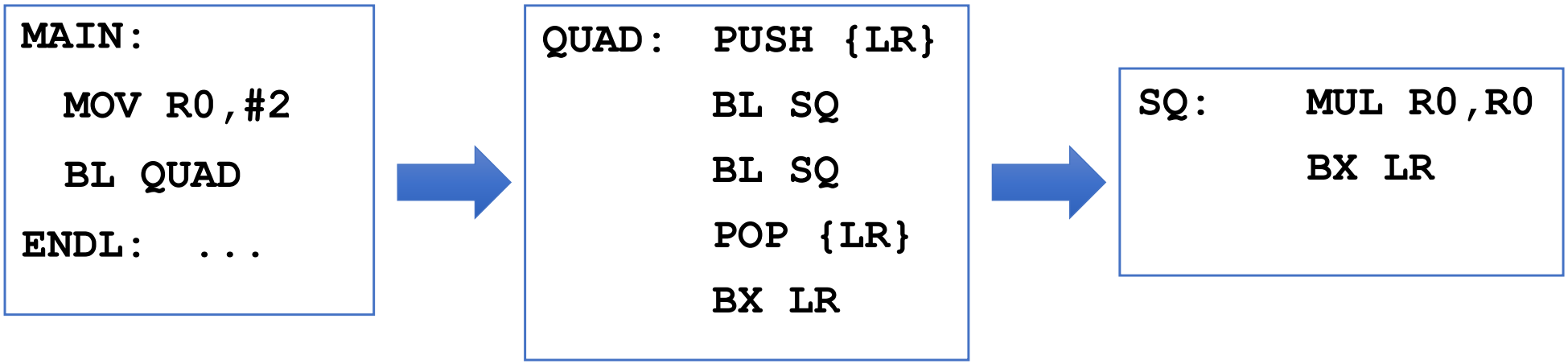
Caller Program	Subroutine/Callee

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

Stacks and Subroutines



Subroutine Calling Another Subroutine

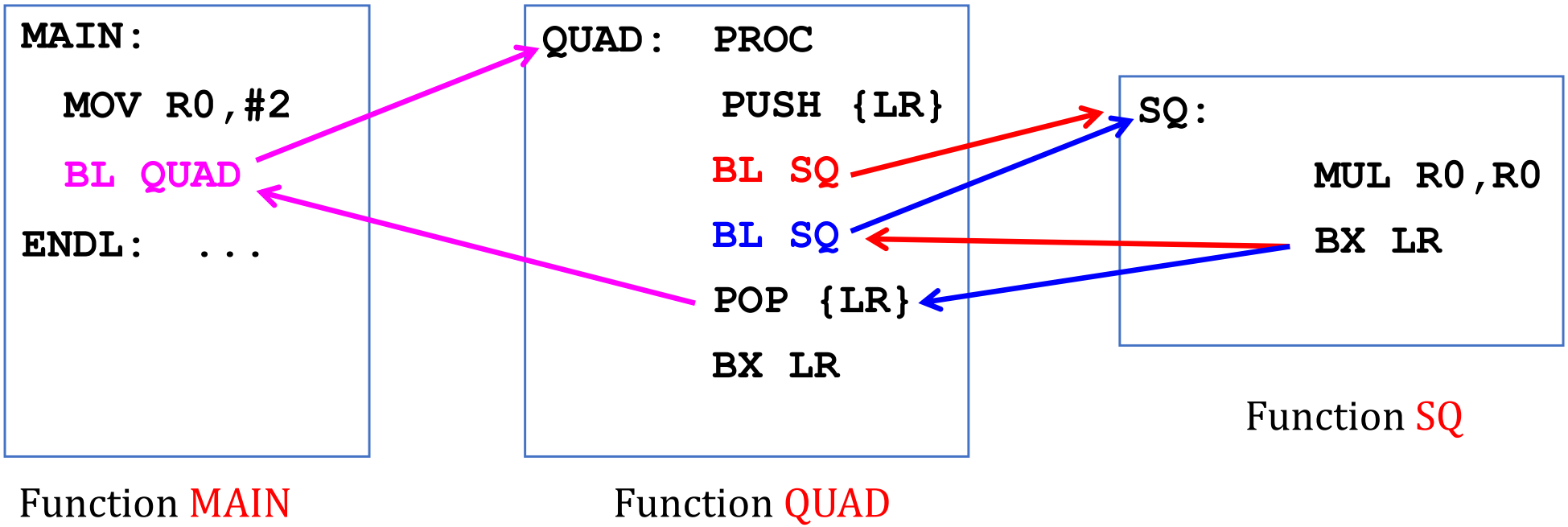


Function MAIN

Function QUAD

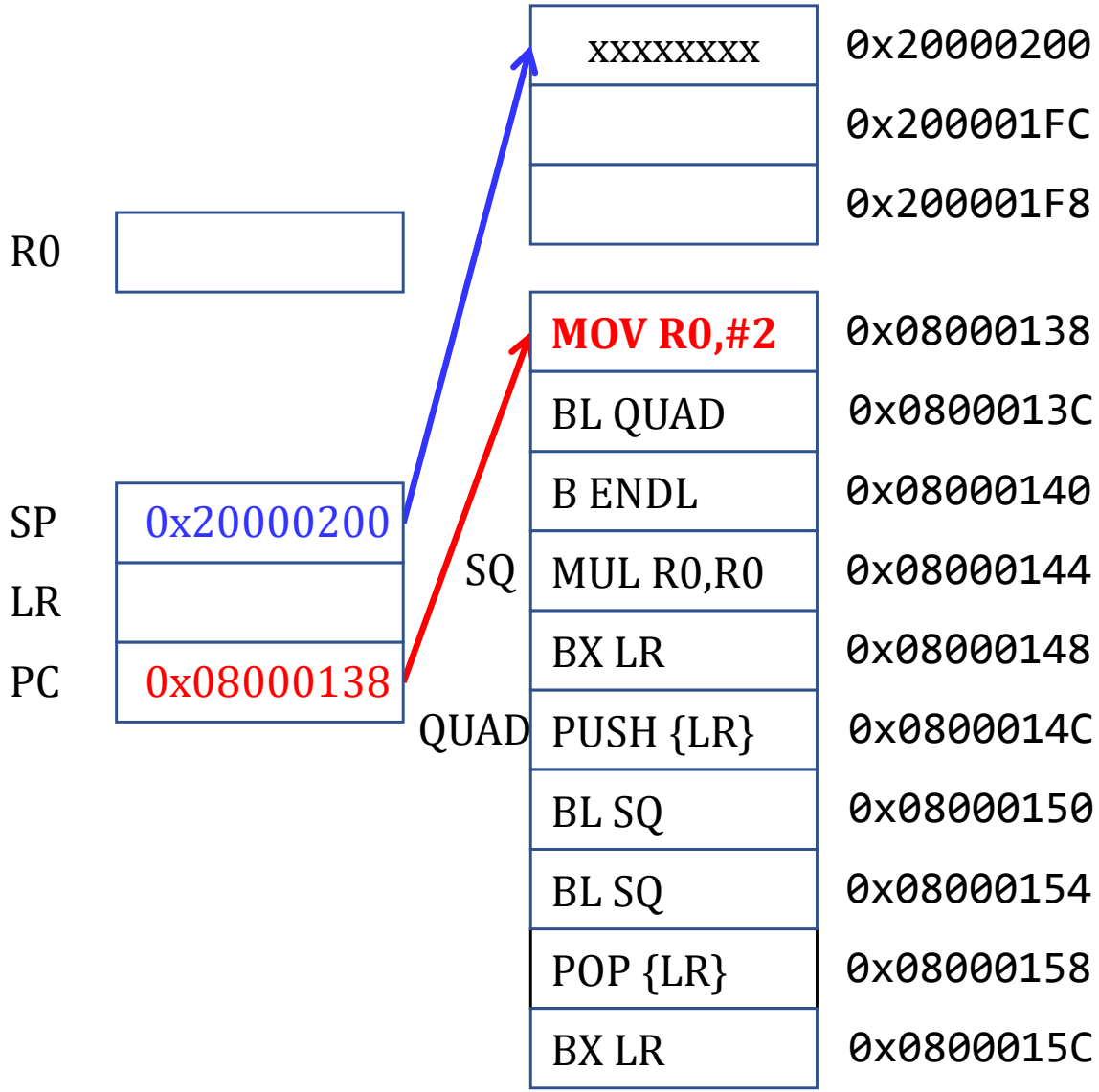
Function SQ

Subroutine Calling Another Subroutine



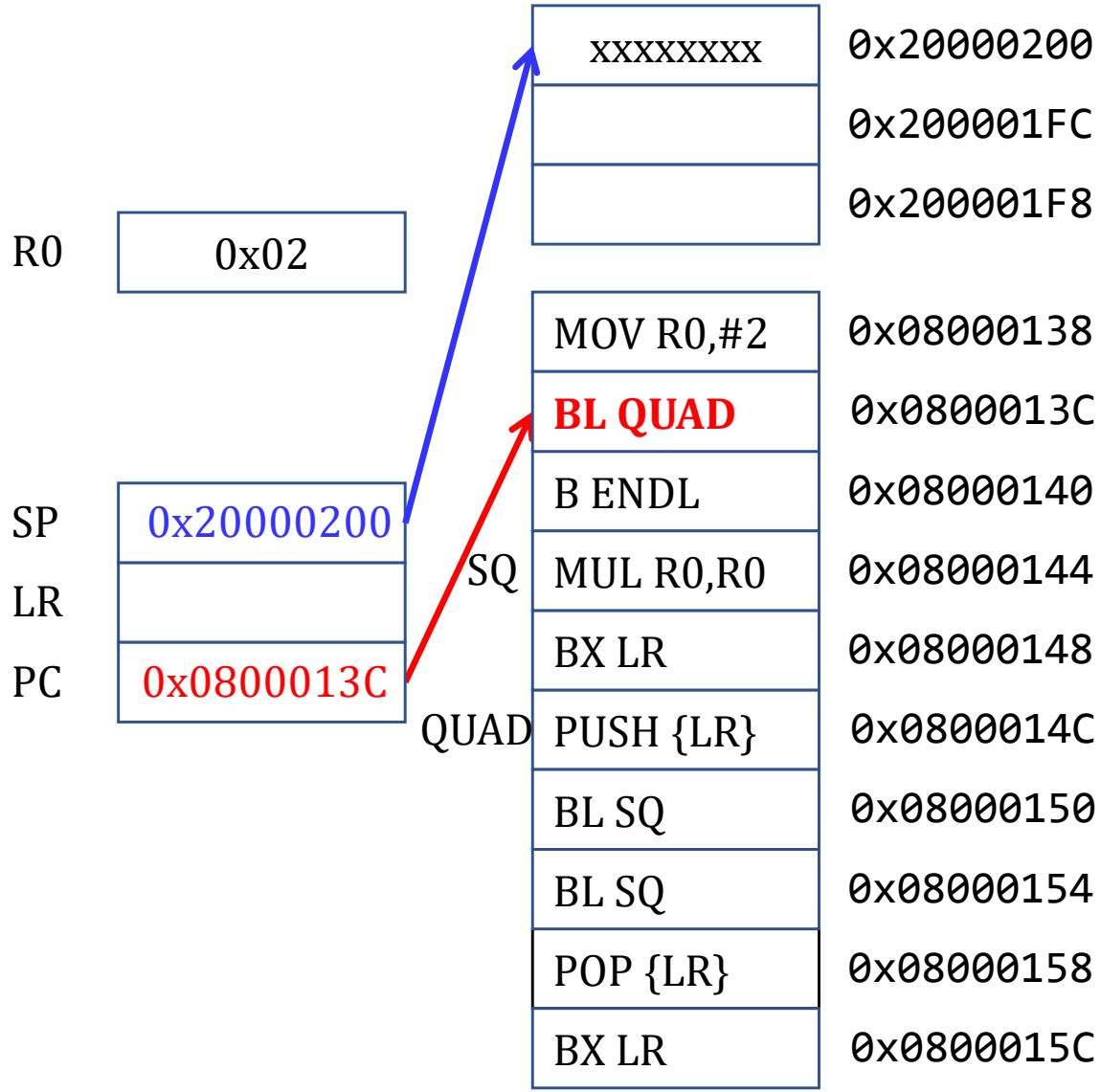
Example: R0 = R0^4

```
        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:   ...
```



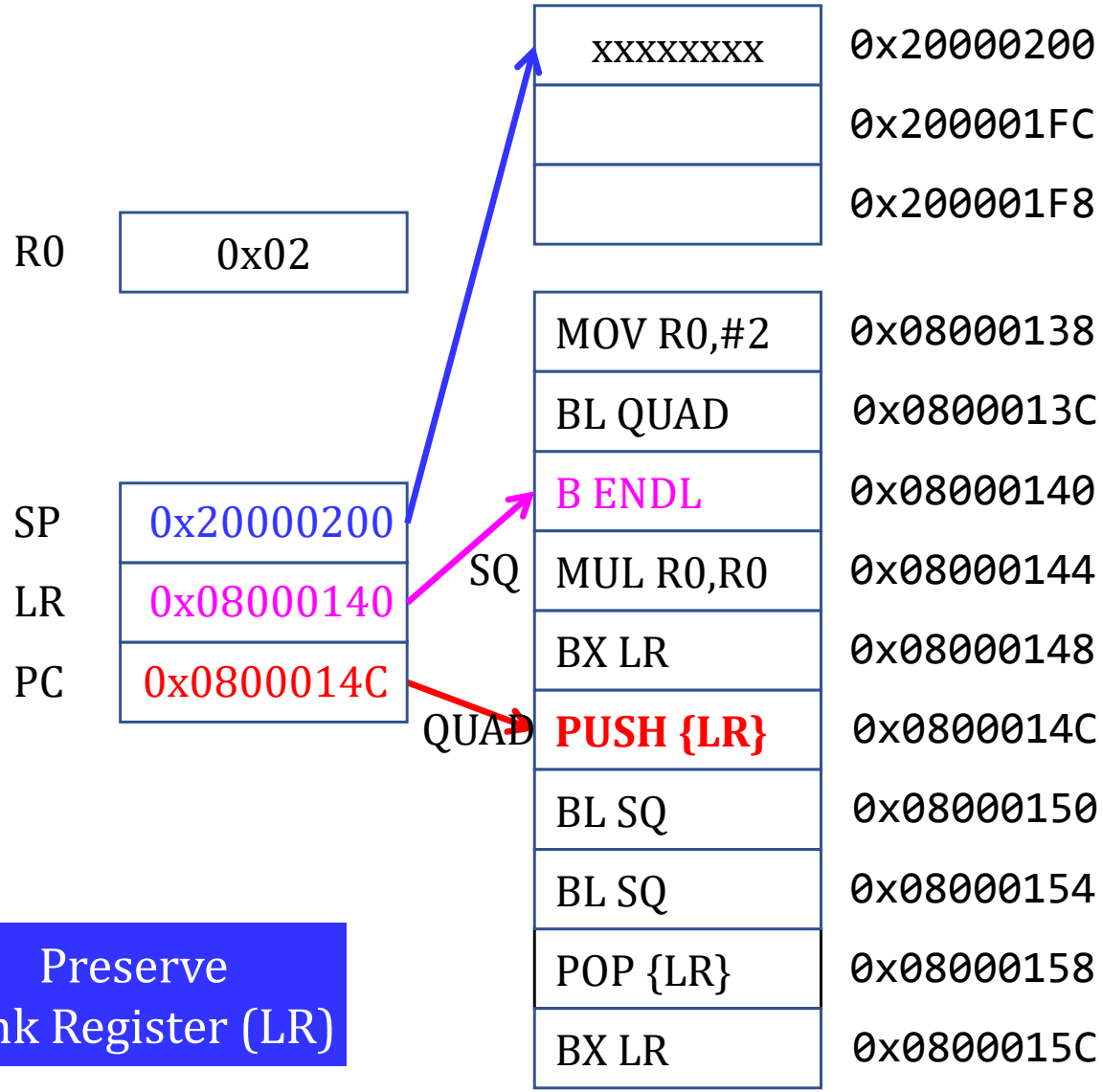
Example: R0 = R0^4

```
        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

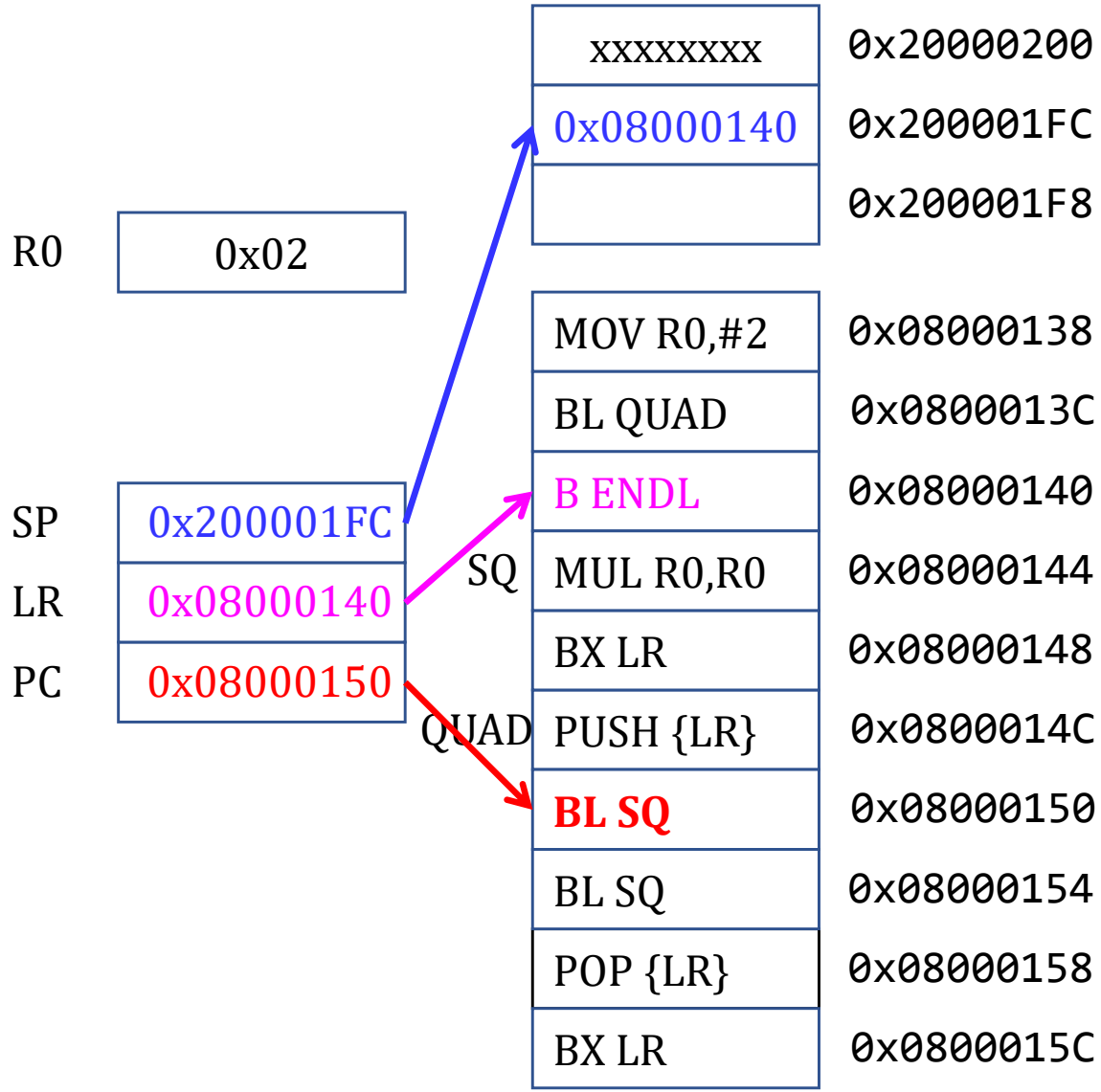
```
        MOV R0,#2
        BL QUAD
        B ENDL

SQ:     MUL R0,R0
        BX LR

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



Preserve Link Register (LR)

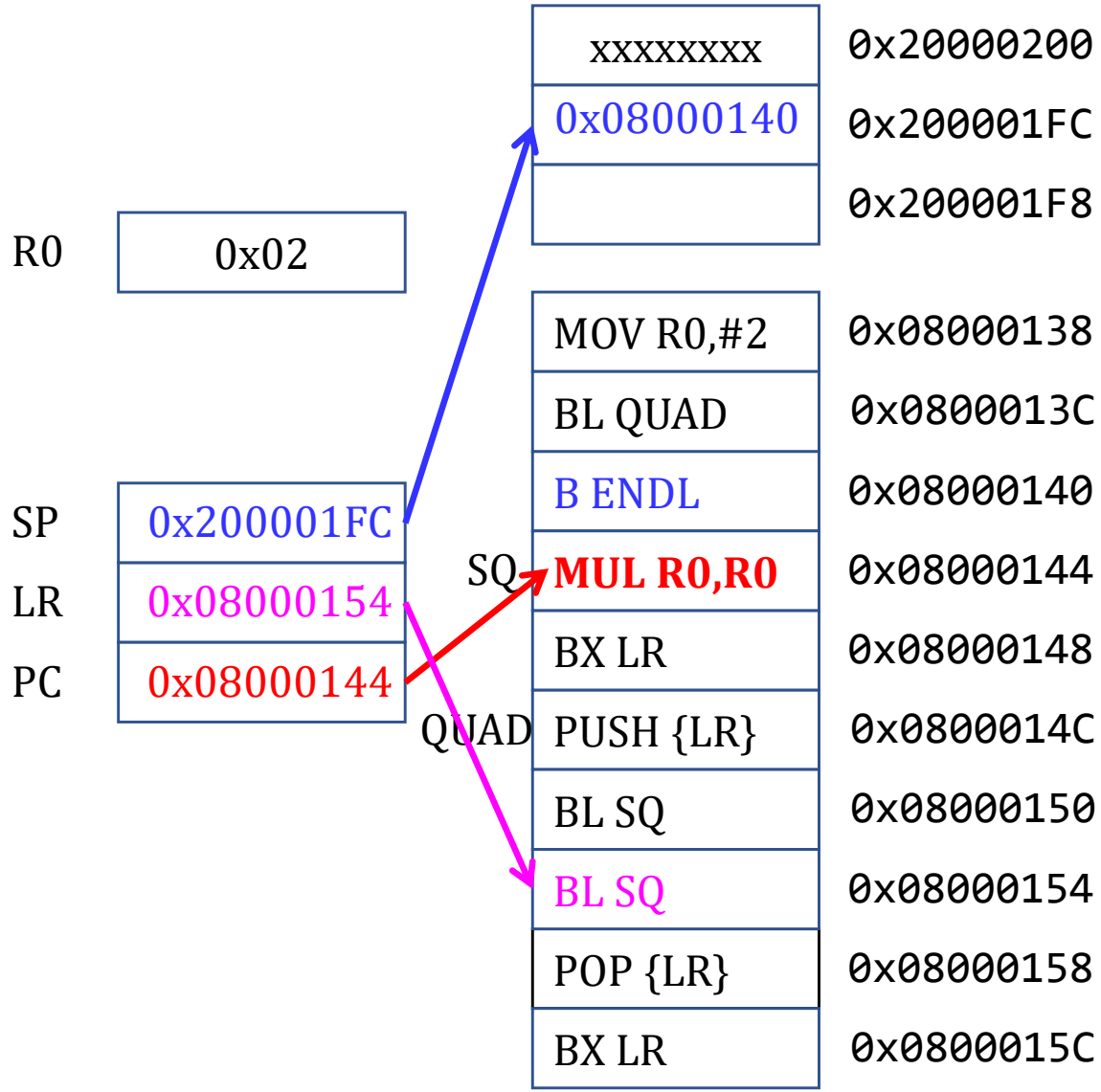
Example: R0 = R0^4

```
        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:   ...
```



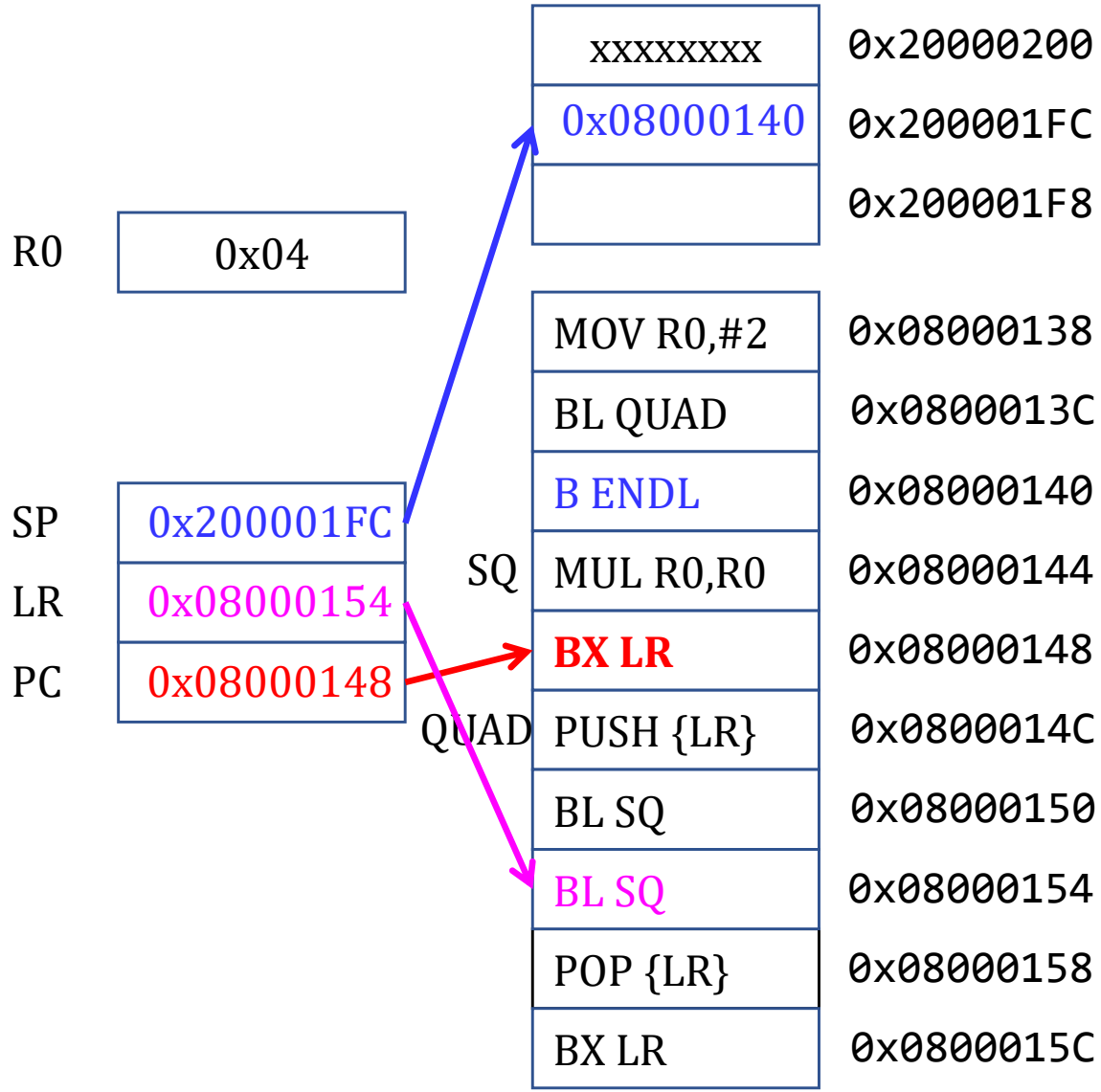
Example: R0 = R0^4

```
        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:   ...
```



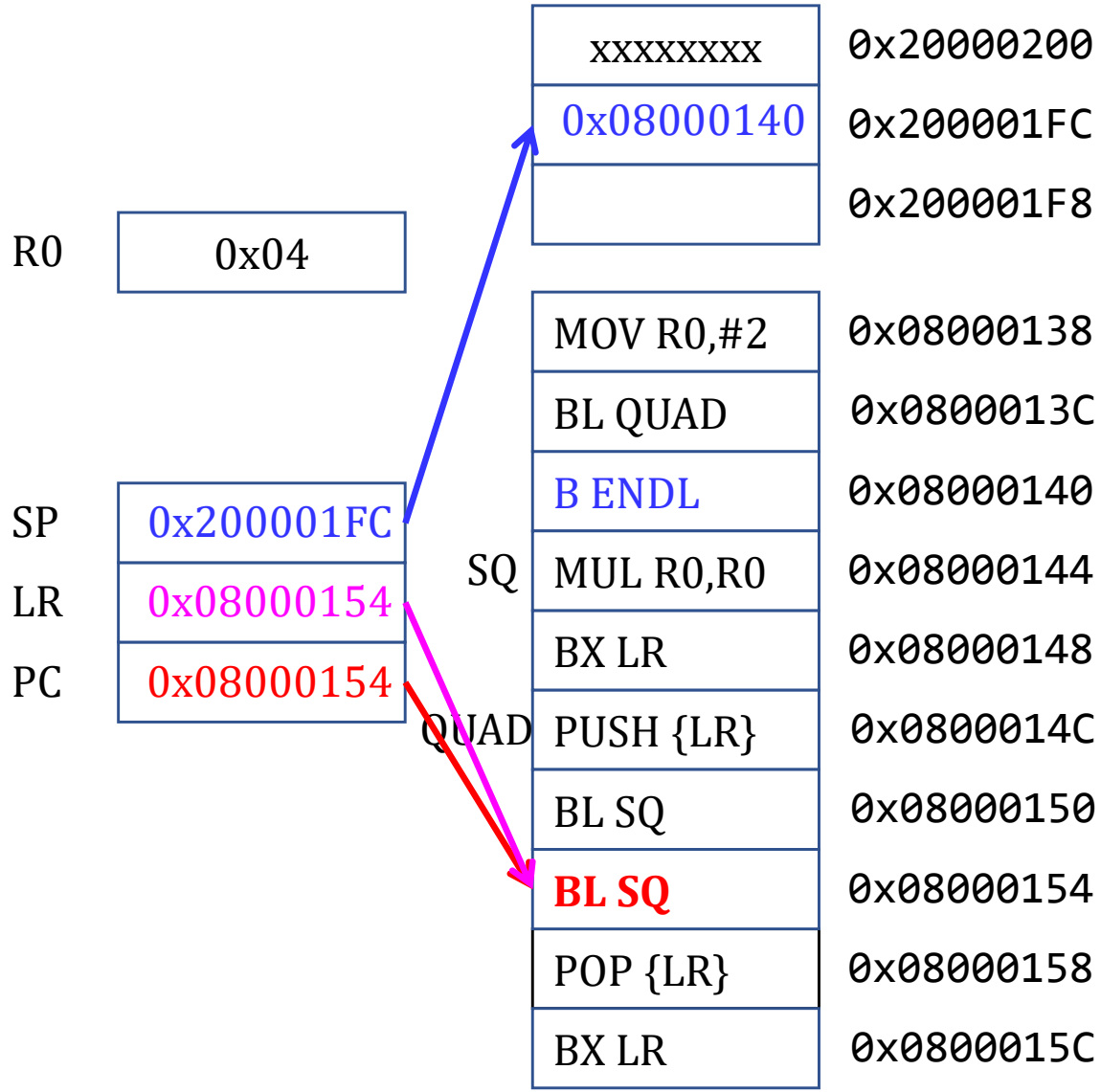
Example: R0 = R0^4

```
        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:   ...
```



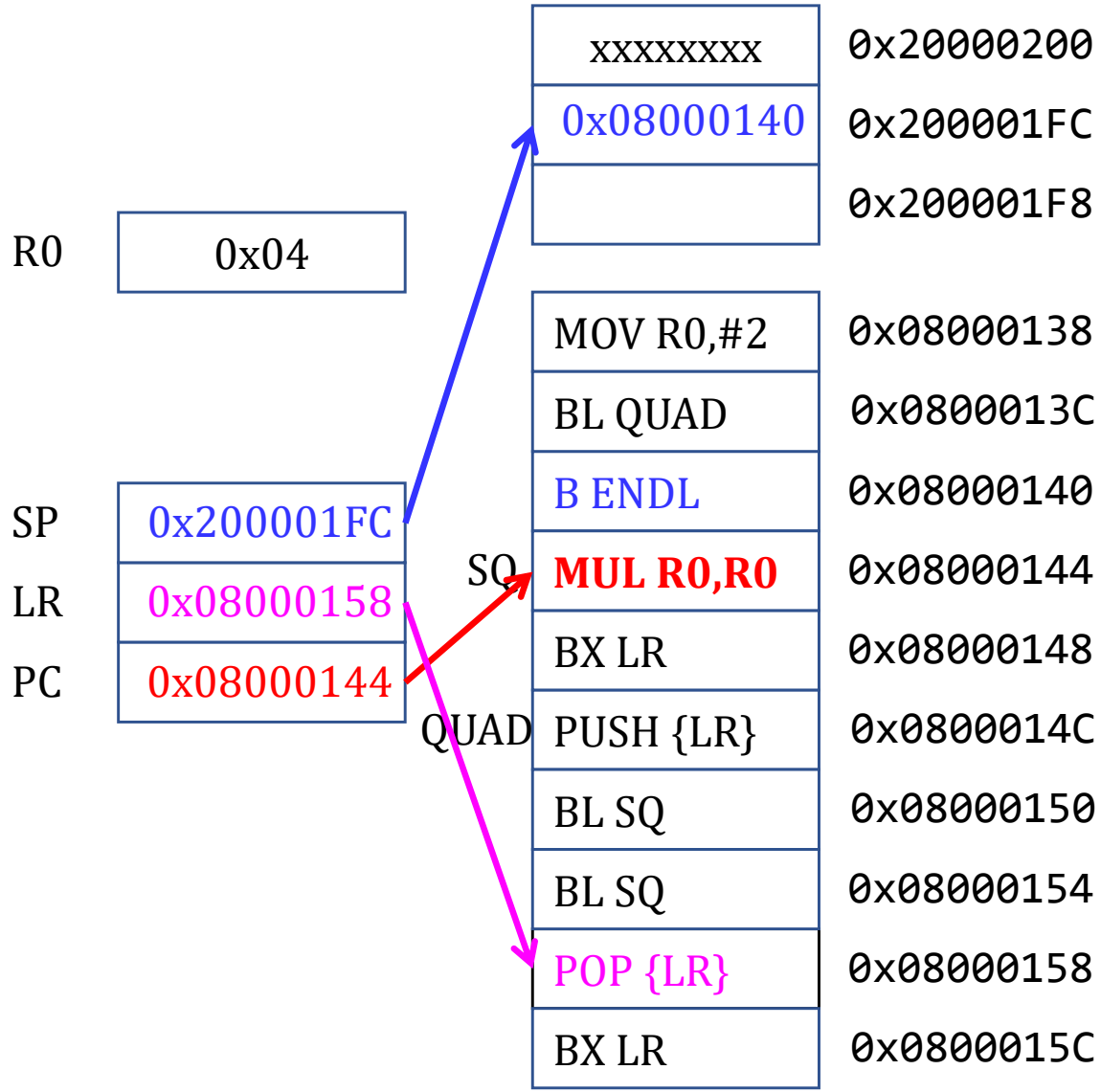
Example: R0 = R0^4

```
        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:   ...
```



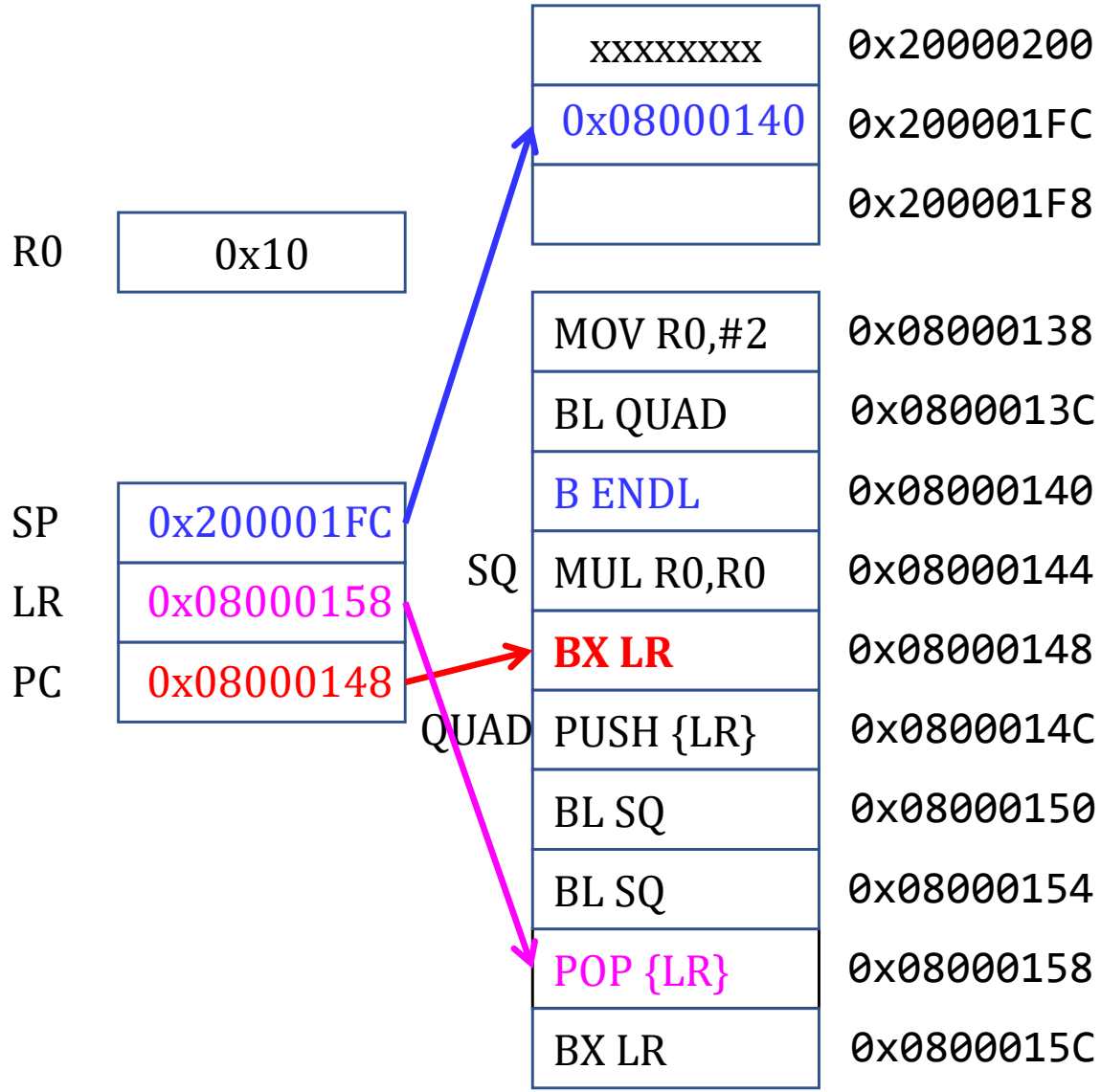
Example: R0 = R0^4

```
        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:   ...
```



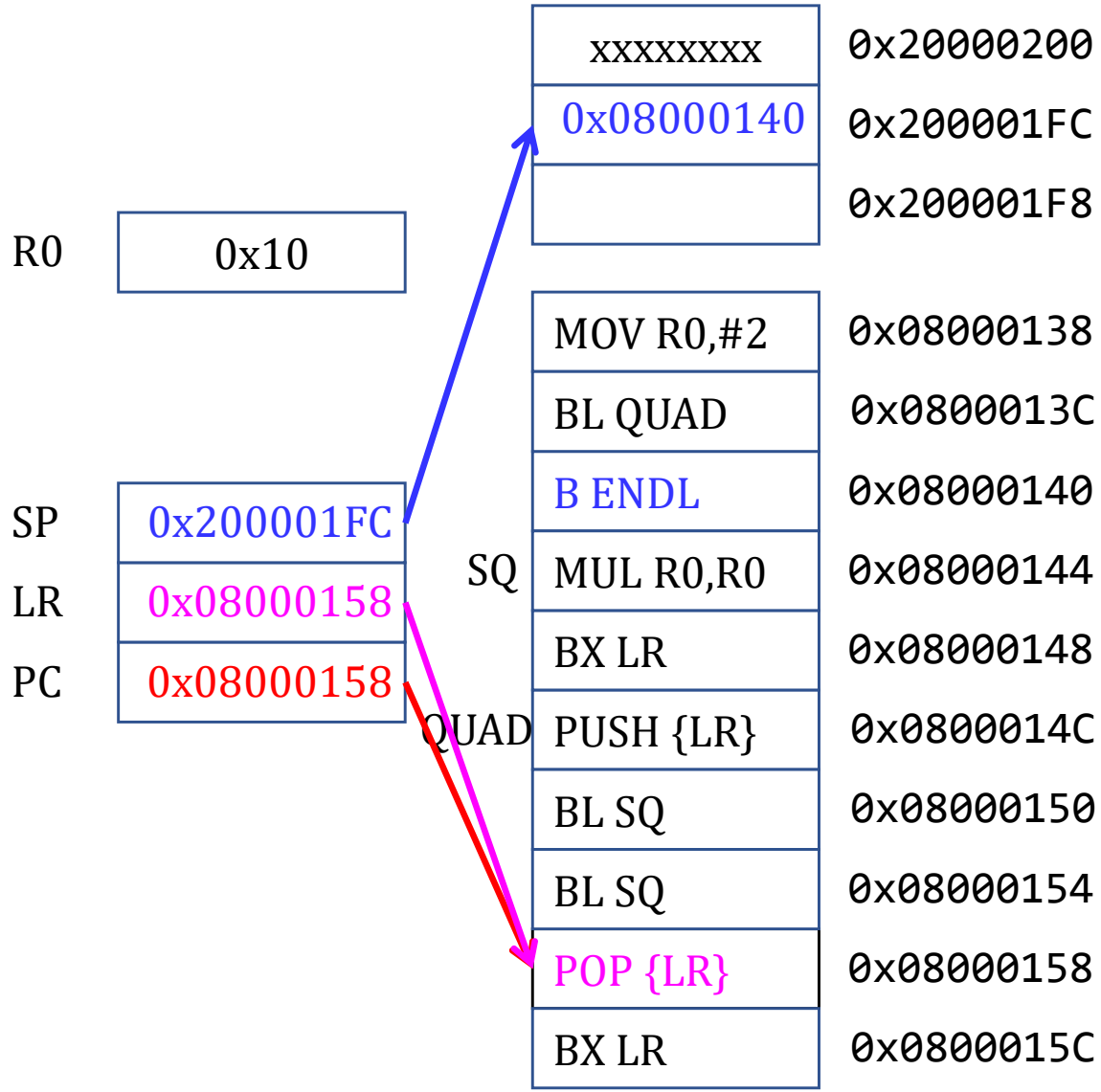
Example: R0 = R0^4

```
        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:   ...
```



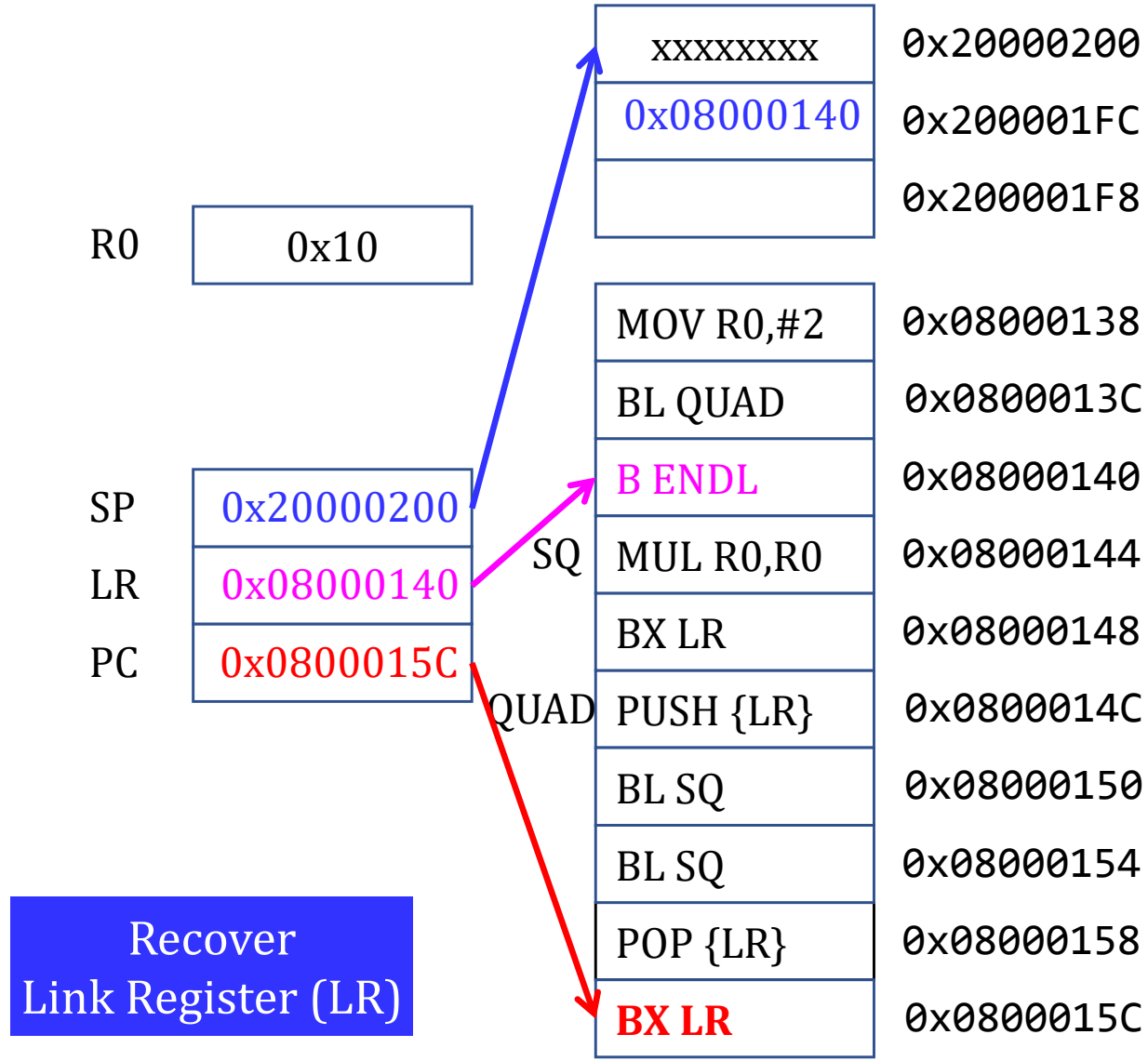
Example: R0 = R0^4

```
        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

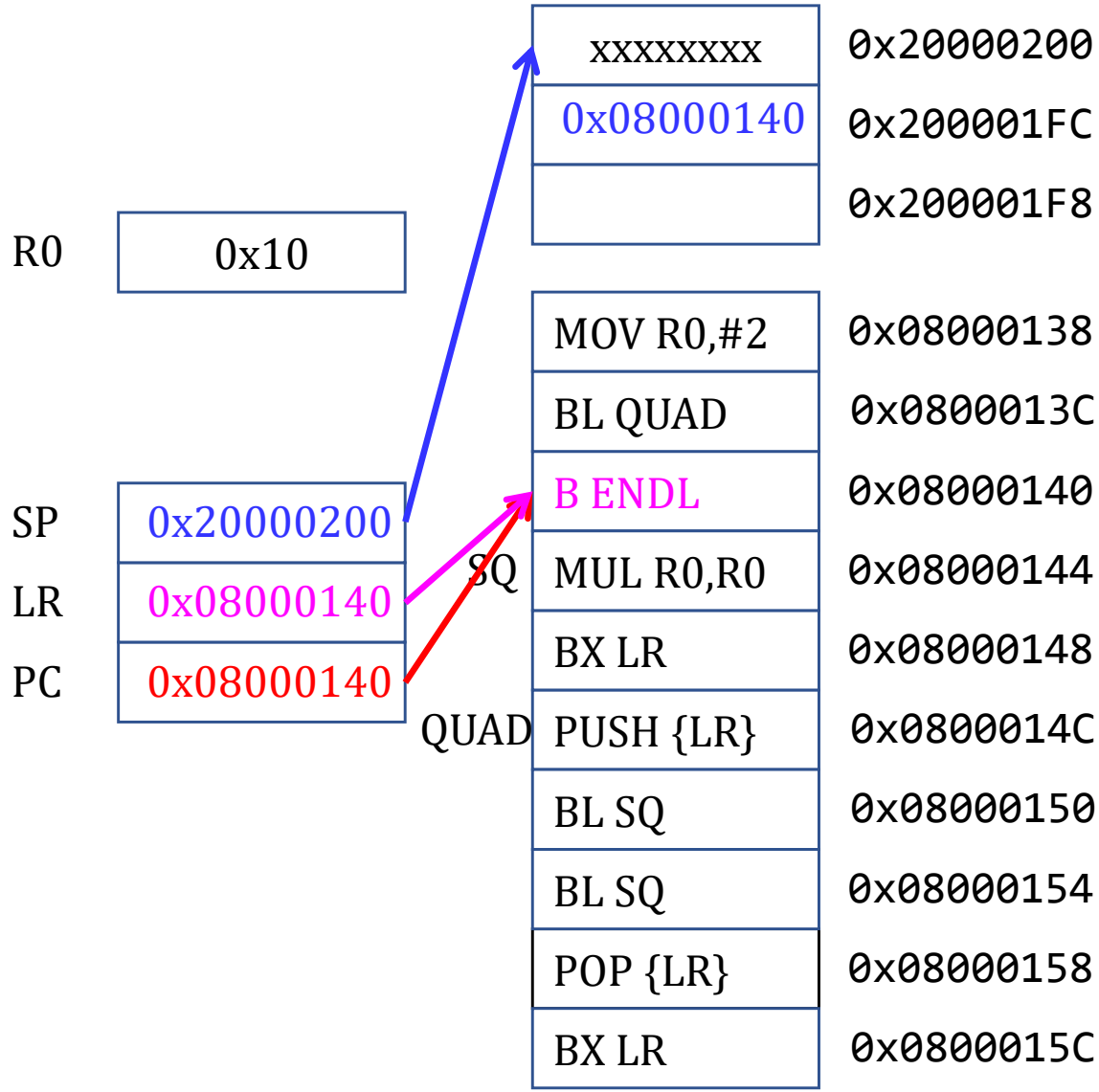
```
        MOV R0,#2
        BL QUAD
        B ENDL

SQ:     MUL R0,R0
        BX LR

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



Recover 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:   ...
```



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

$$1 \times 2 \times 3 \times \ldots \times n$$

```
factorial(0) = 1
factorial(1) = 1              = 1 × factorial(0)
factorial(2) = 2×1            = 2 × factorial(1)
factorial(3) = 3×2×1          = 3 × factorial(2)
factorial(4) = 4×3×2×1        = 4 × factorial(3)
factorial(n) = n×(n-1)×…×1    = n × factorial(n-1)
```

Classic Example: Factorial

- Factorial is the classic example:
  - **6! = 6 × 5!**
  - **6! = 6 × 5 x 4!**

  …
  - **6! = 6 × 5 × 4 × 3 × 2 × 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));

}
```

Classic Example: Fibonacci Numbers

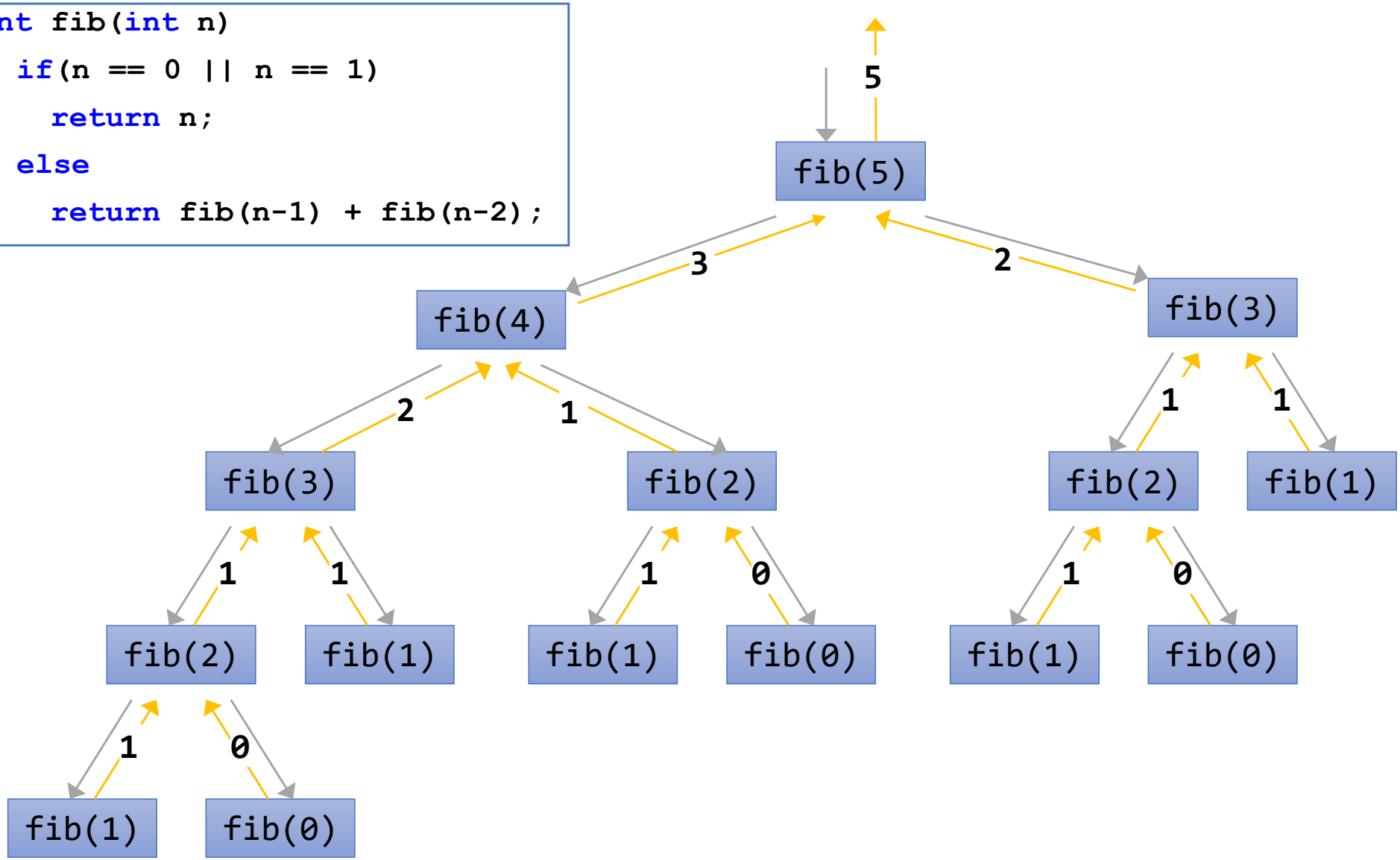
```
f(n) = f(n-1) + f(n-2)
f(0) = 1
f(1) = 1
```

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, …

```
int Fibonacci(int n) {

  if (n <= 1)
    return 1;   /* base case */

  return (Fibonacci(n-1) + Fibonacci(n-2));

}
```

Analysis of fib(5)

```
int fib(int n)
  if(n == 0 || n == 1)
    return n;
  else
    return fib(n-1) + fib(n-2);
```

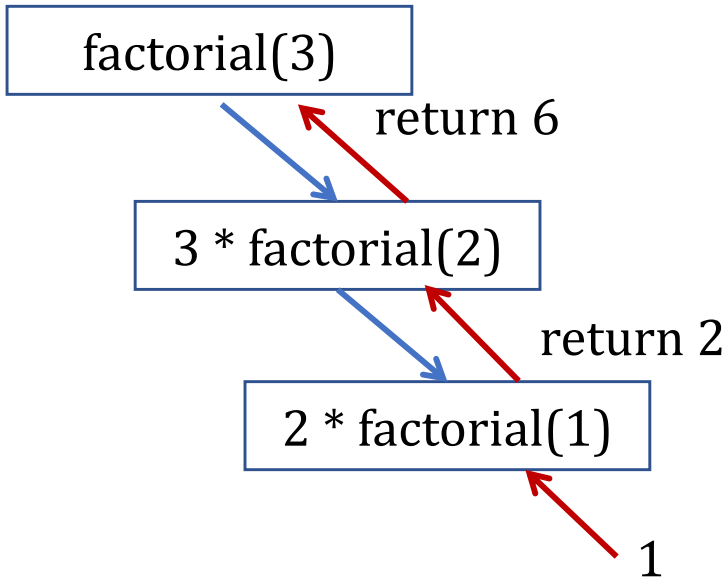


Recursive Factorial in Assembly

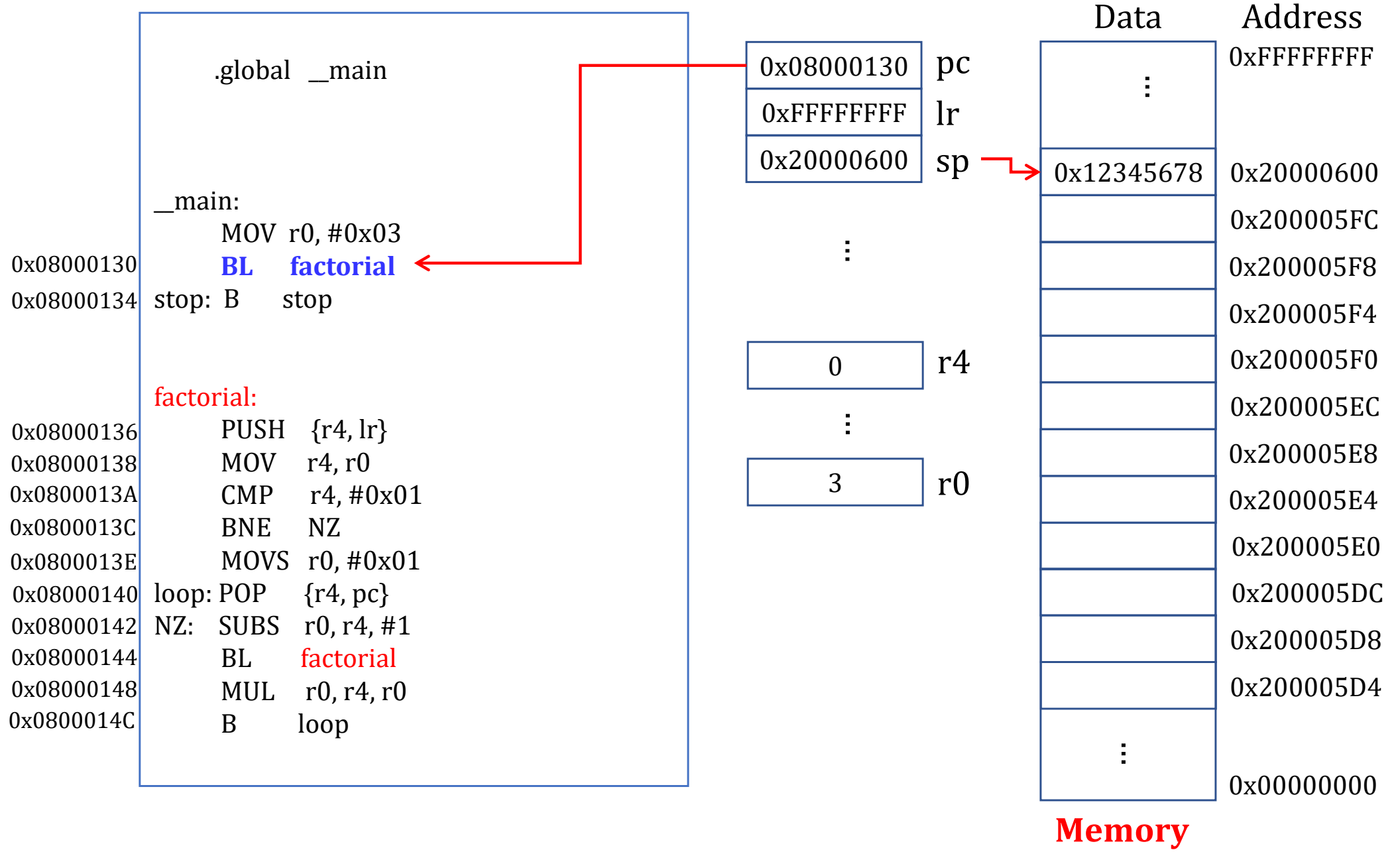
- push LR (& working registers) onto stack before nested call
- pop LR (& working registers) off stack after nested return

Recursive Factorial in Assembly

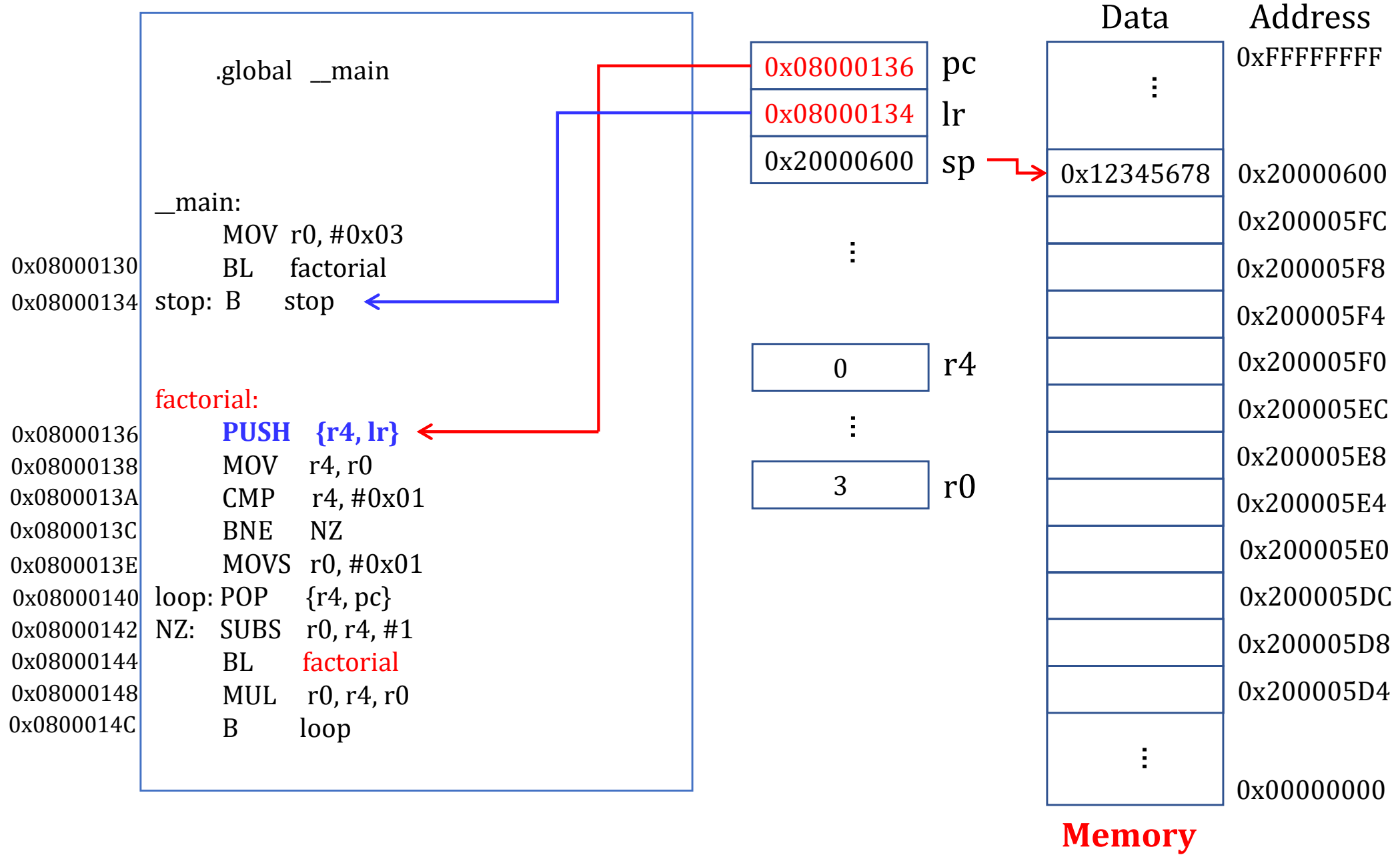
```
                .global  __main


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


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



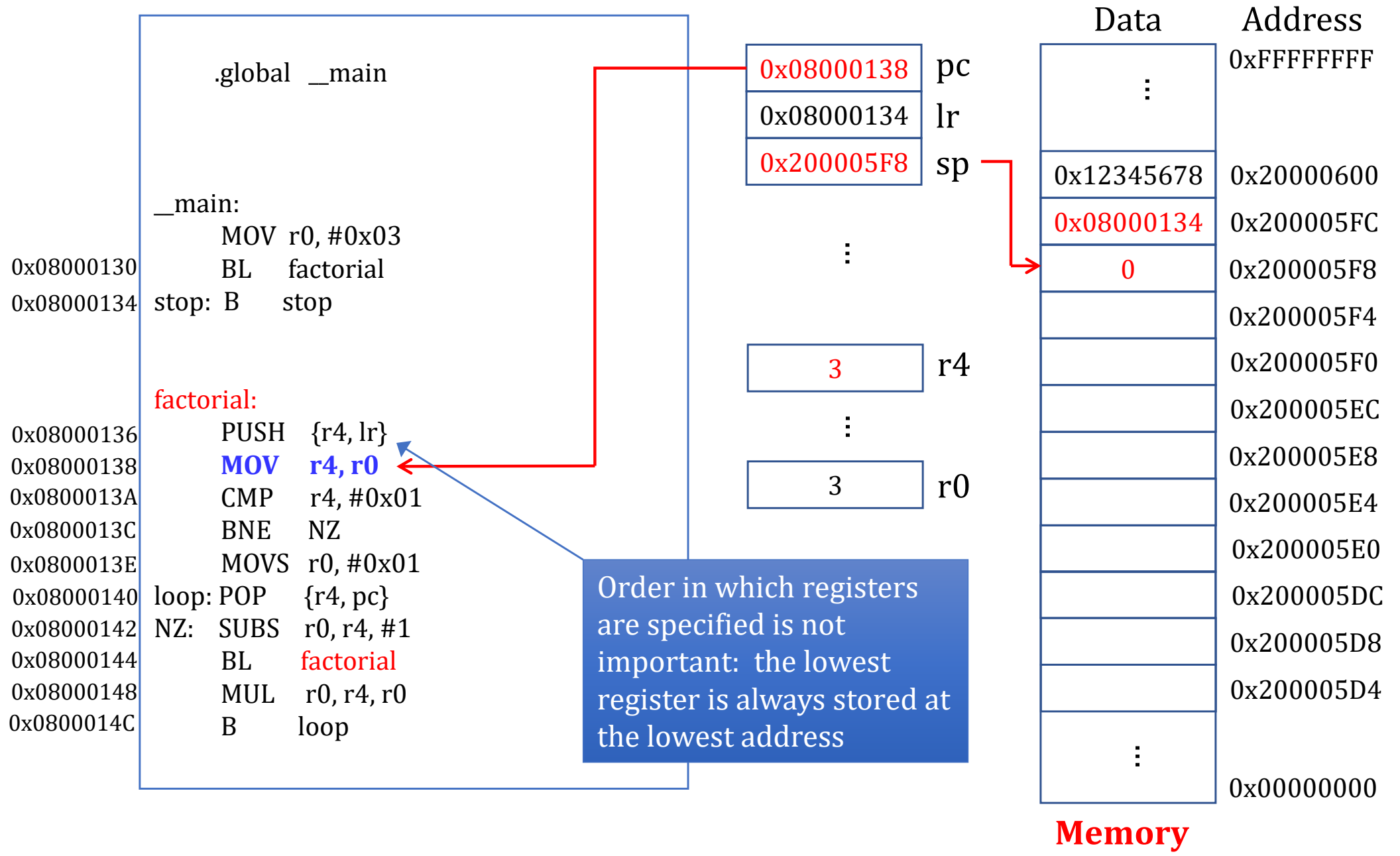
Recursive Factorial in Assembly



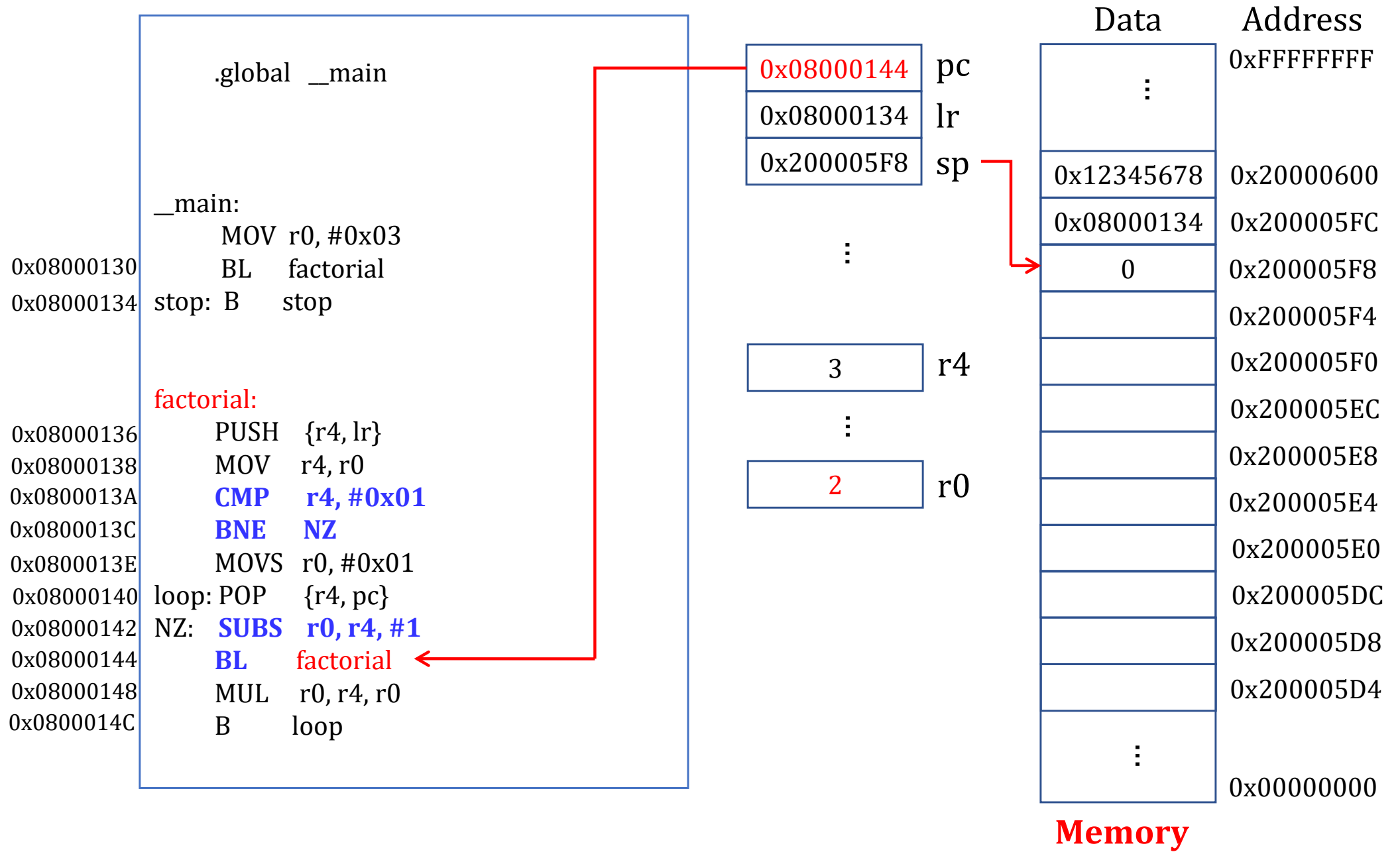
Recursive Factorial in Assembly



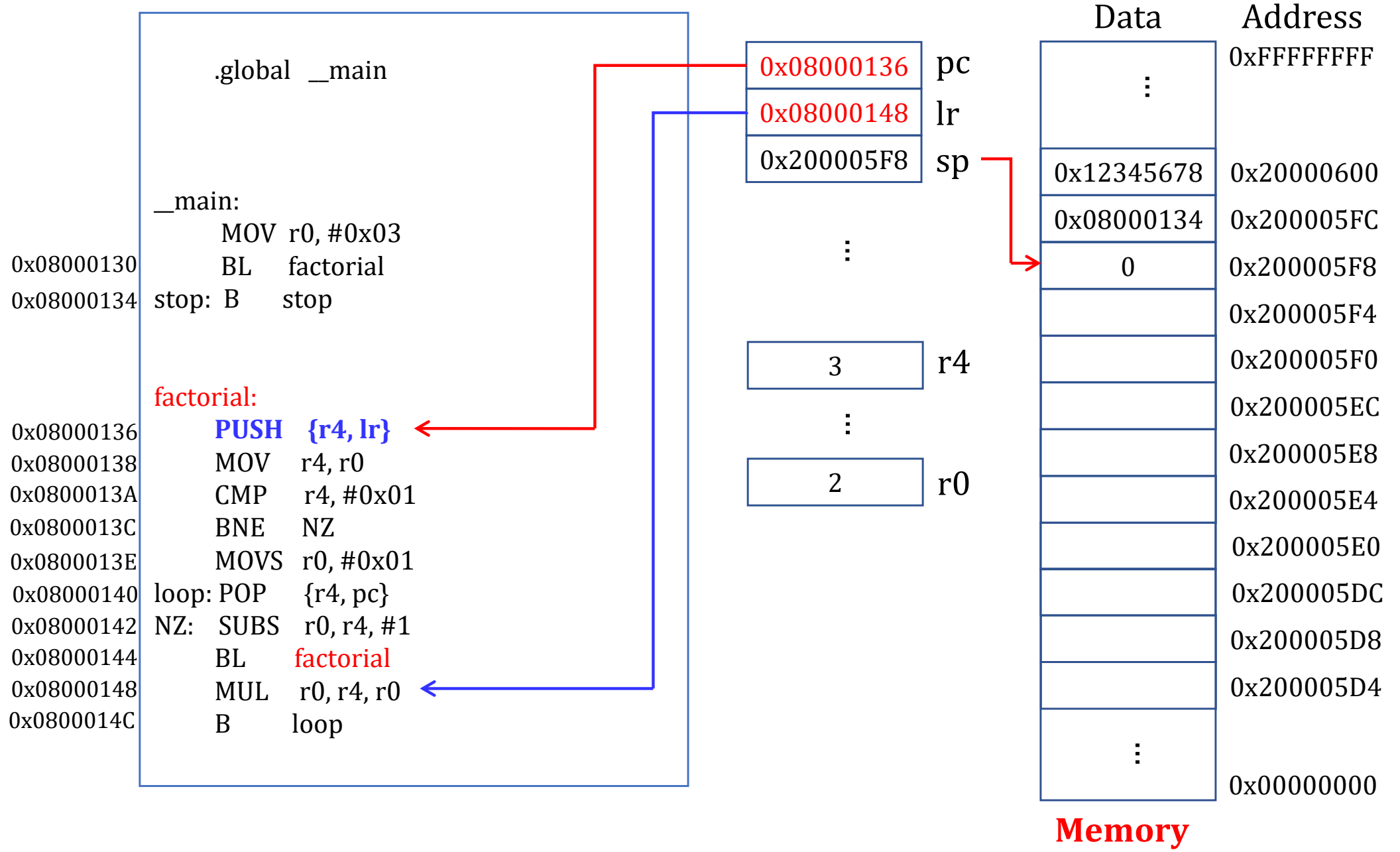
Recursive Factorial in Assembly



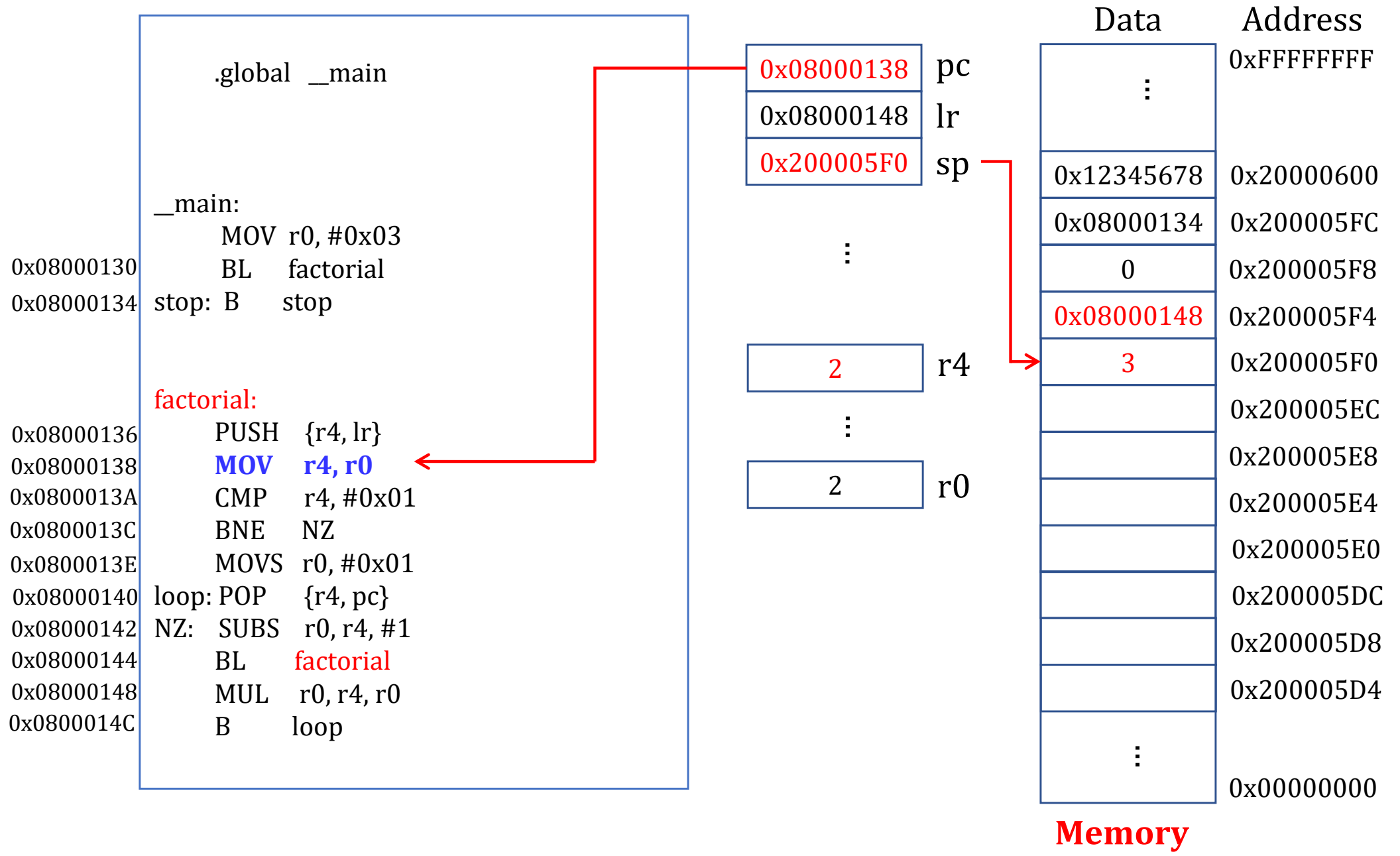
Recursive Factorial in Assembly



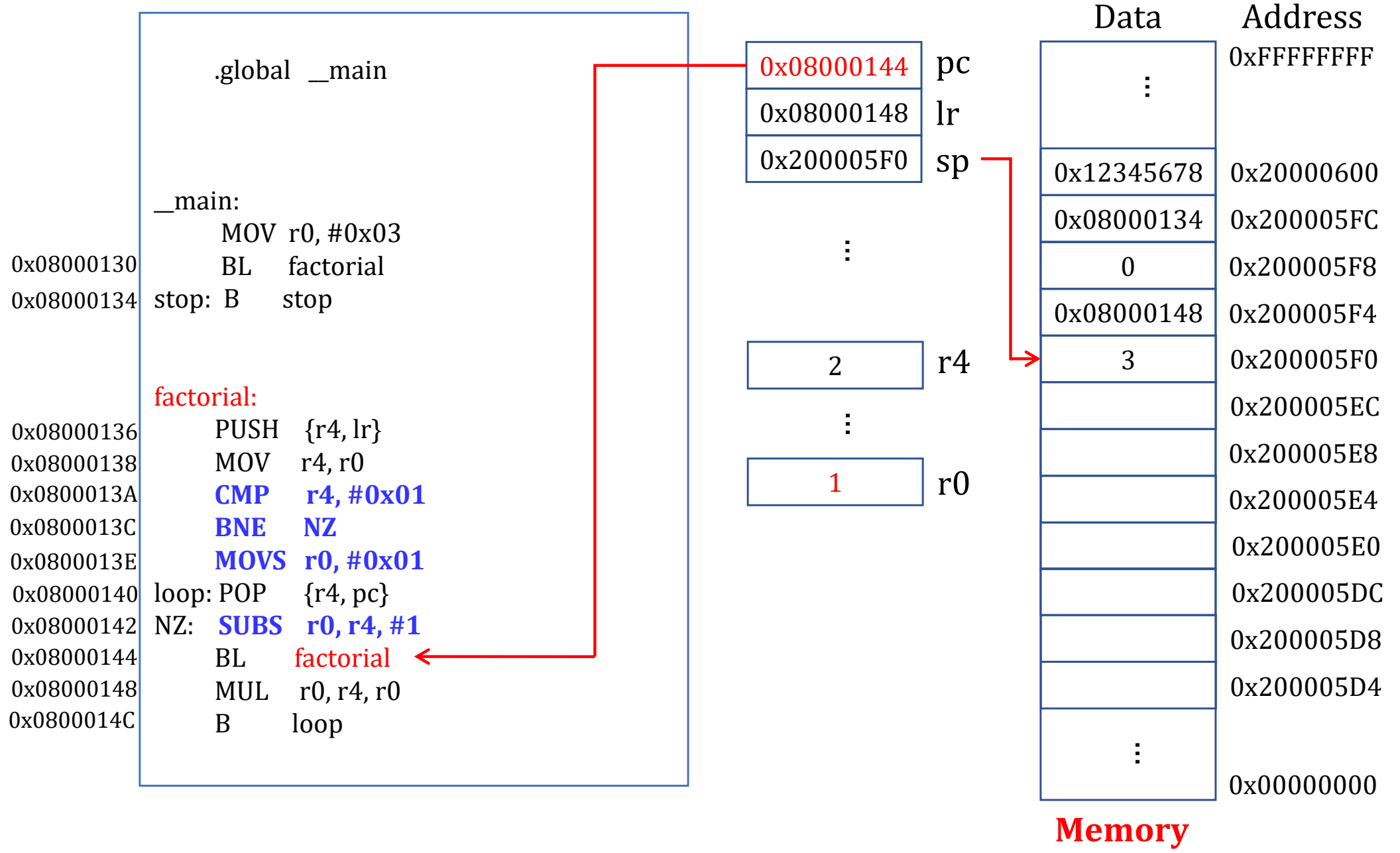
Recursive Factorial in Assembly



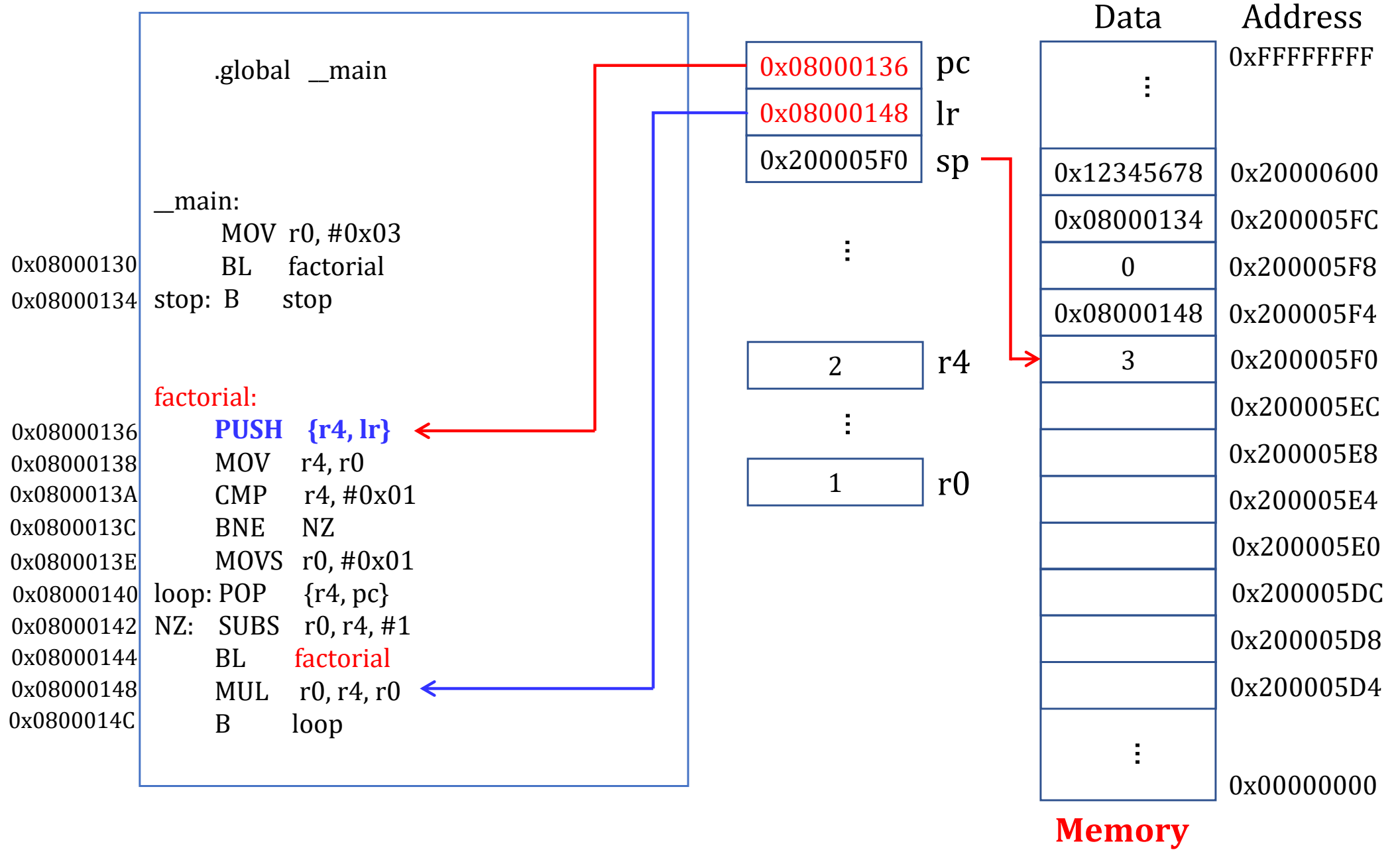
Recursive Factorial in Assembly



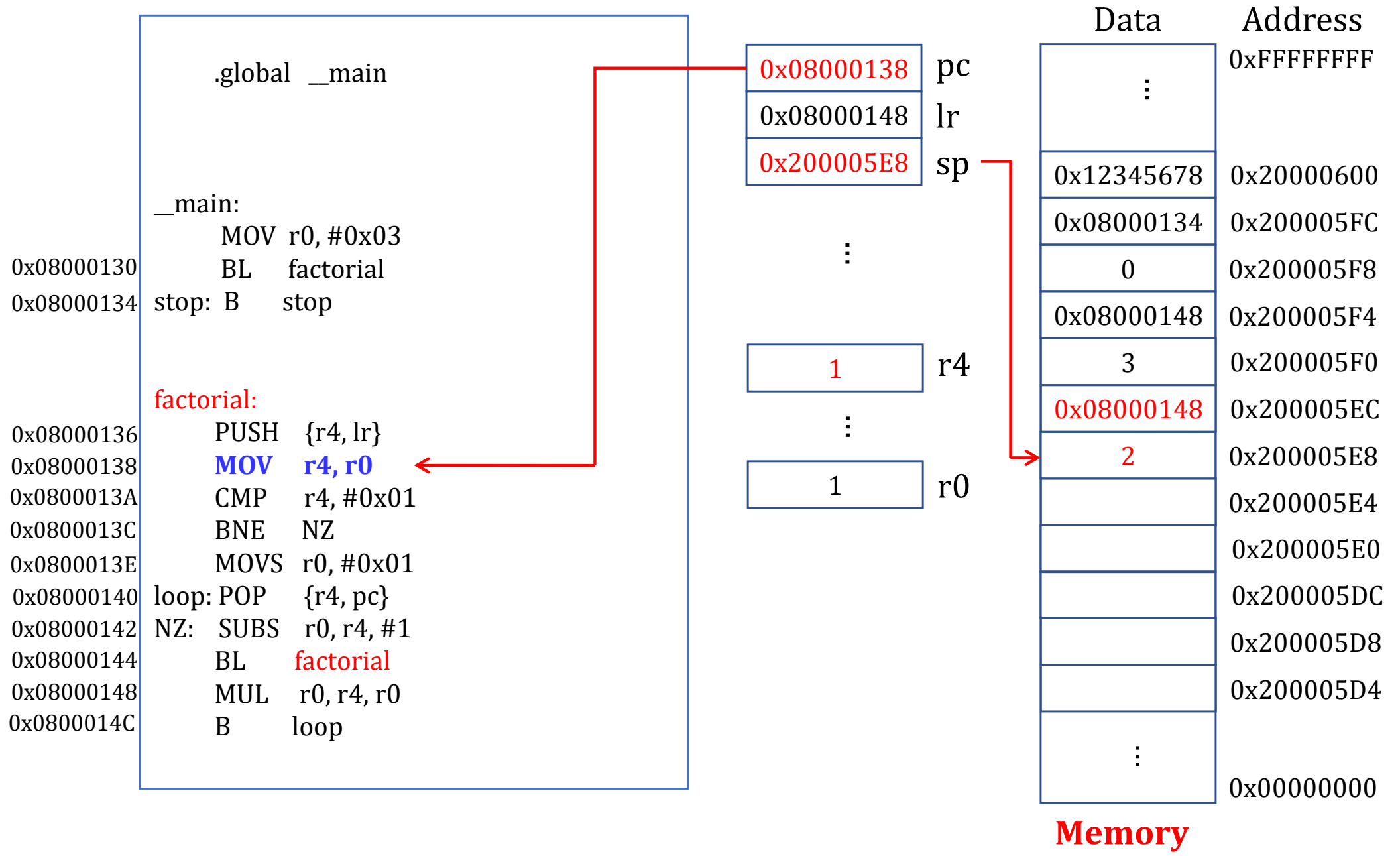
Recursive Factorial in Assembly



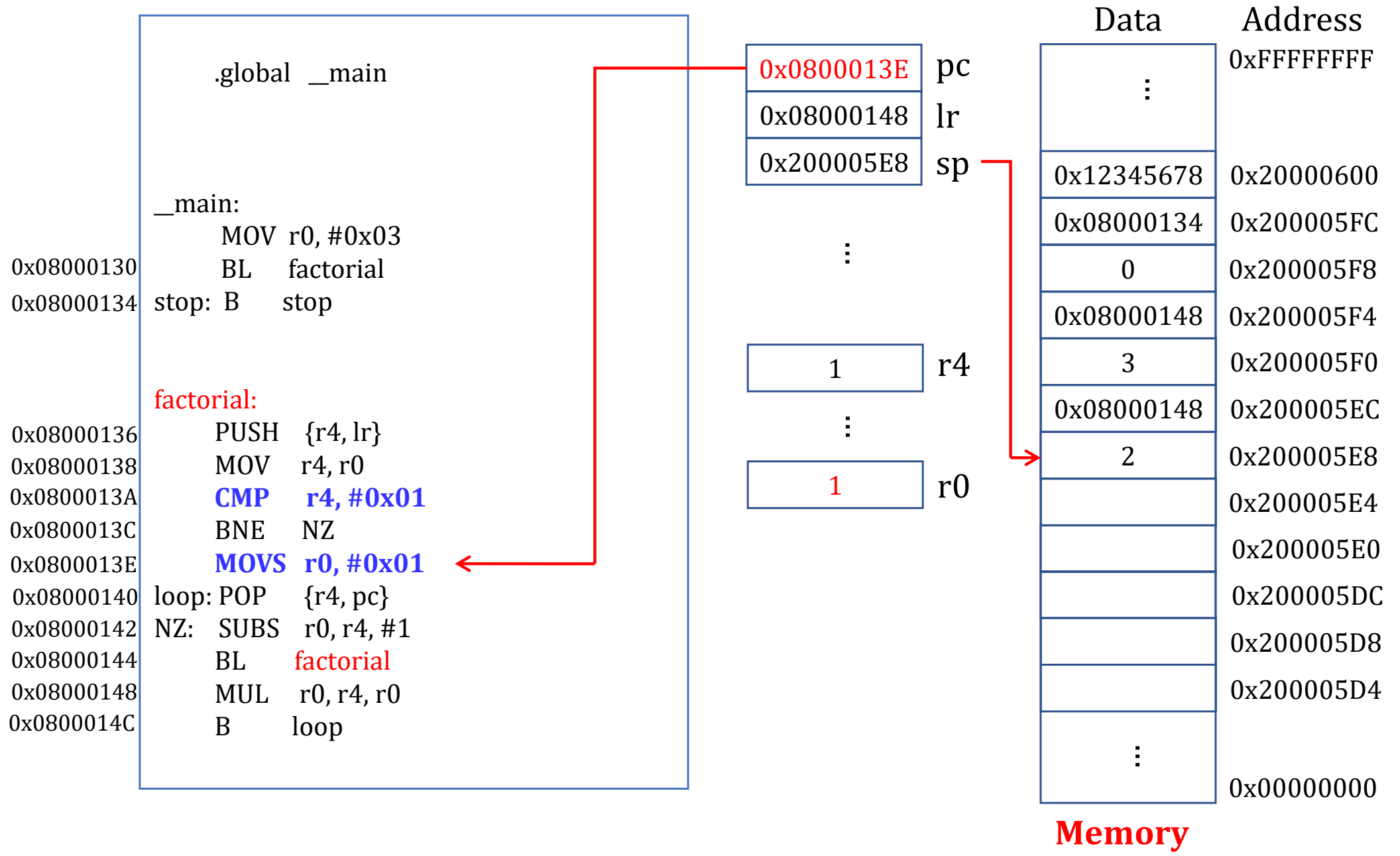
Recursive Factorial in Assembly



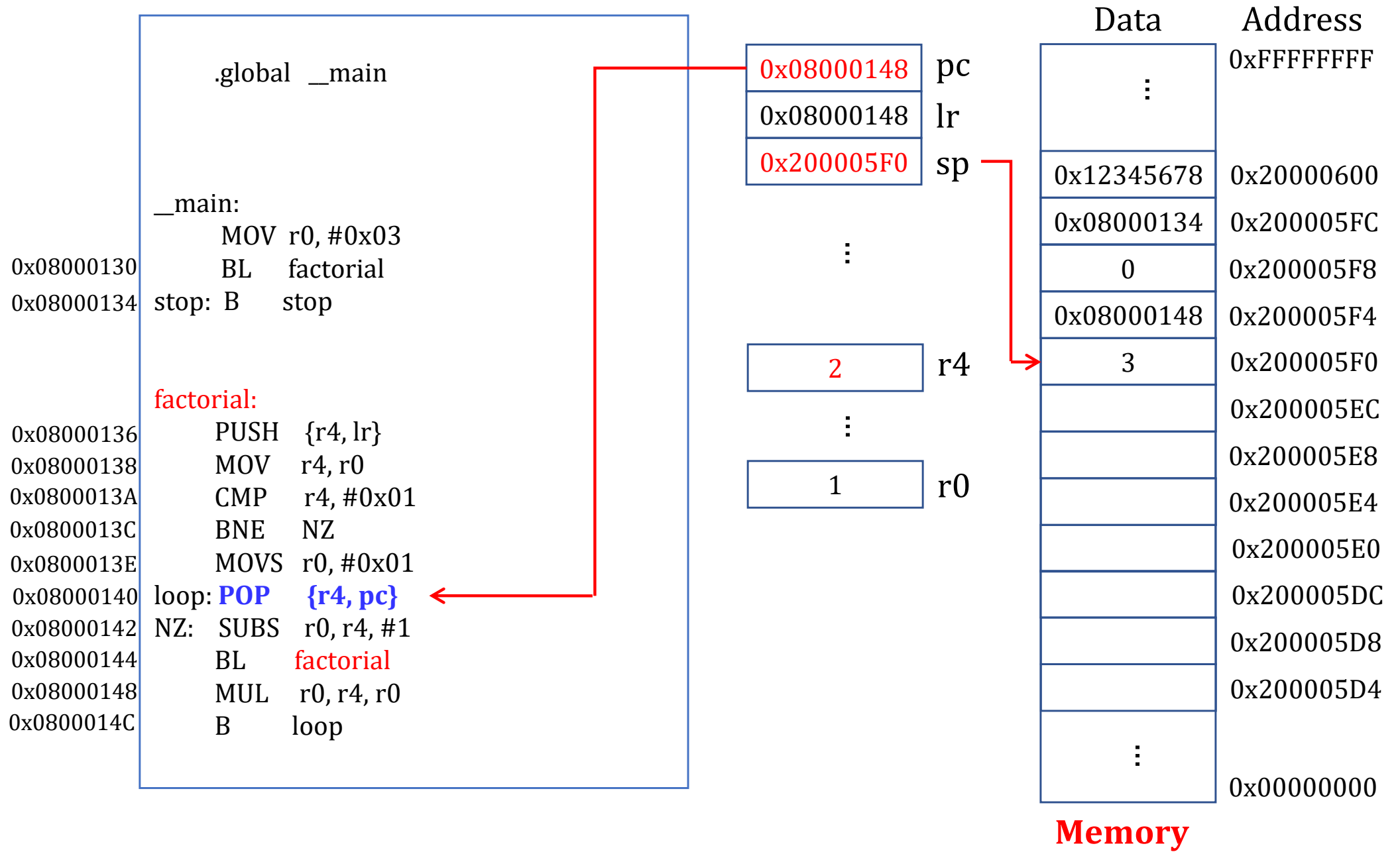
Recursive Factorial in Assembly



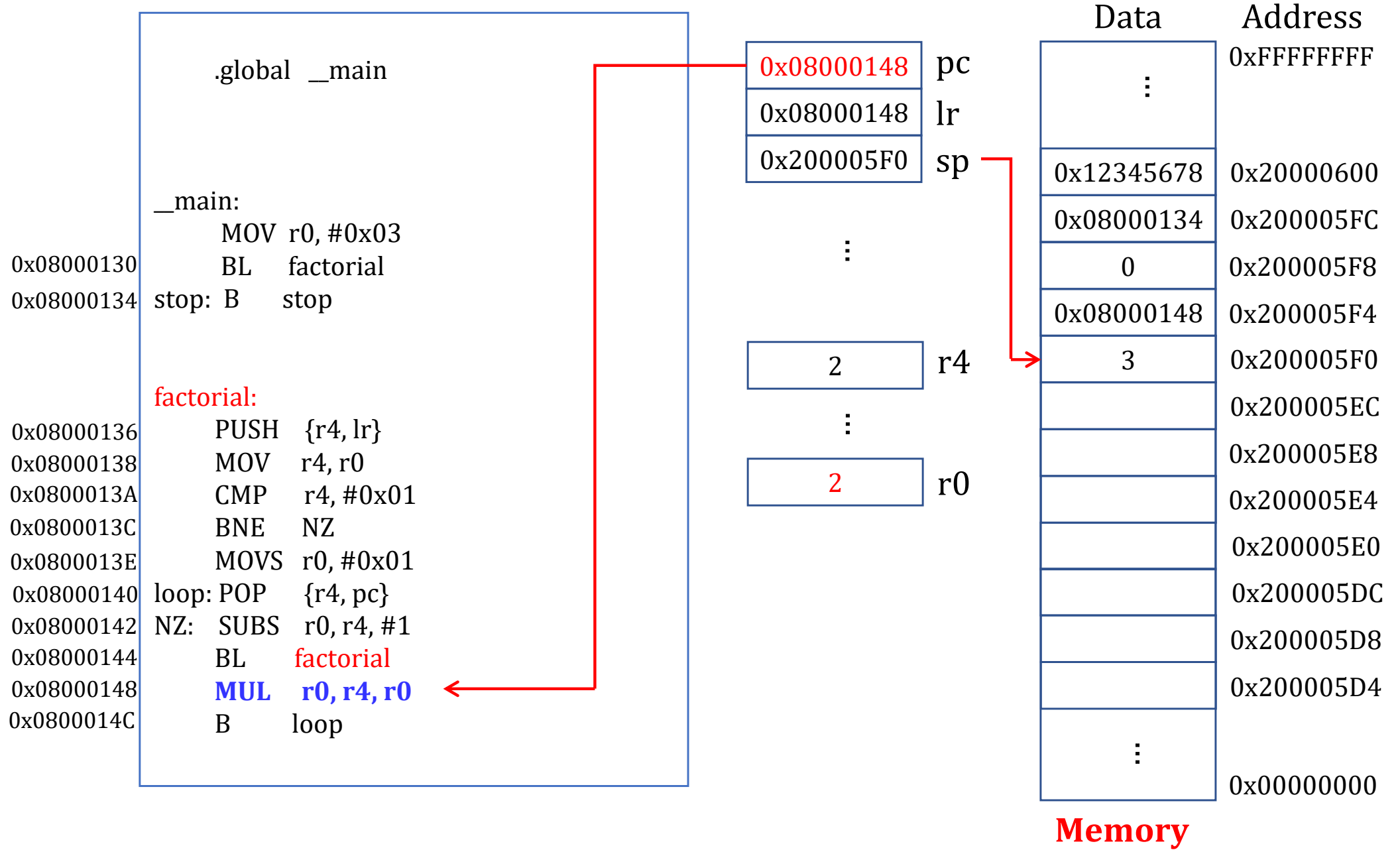
Recursive Factorial in Assembly



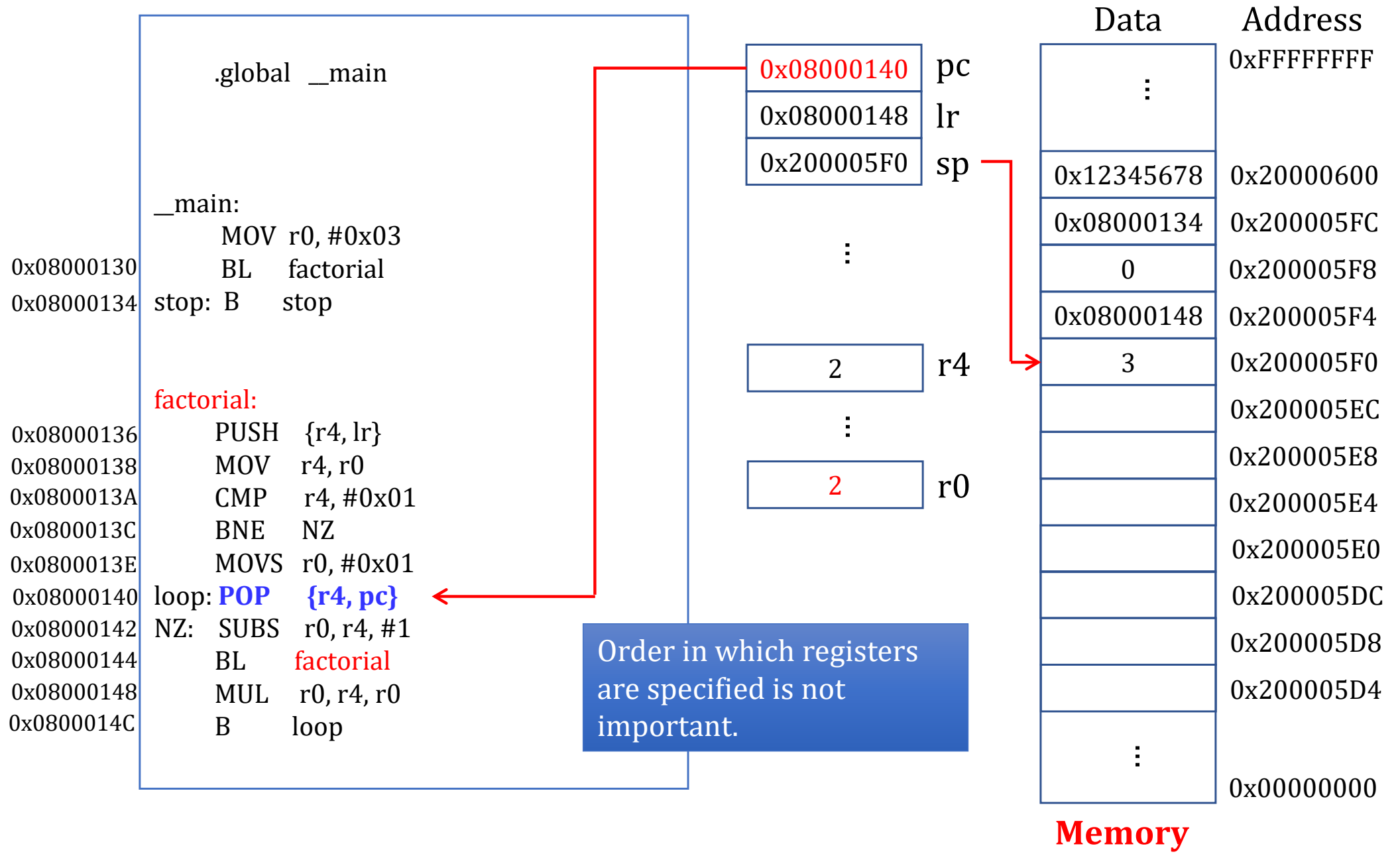
Recursive Factorial in Assembly



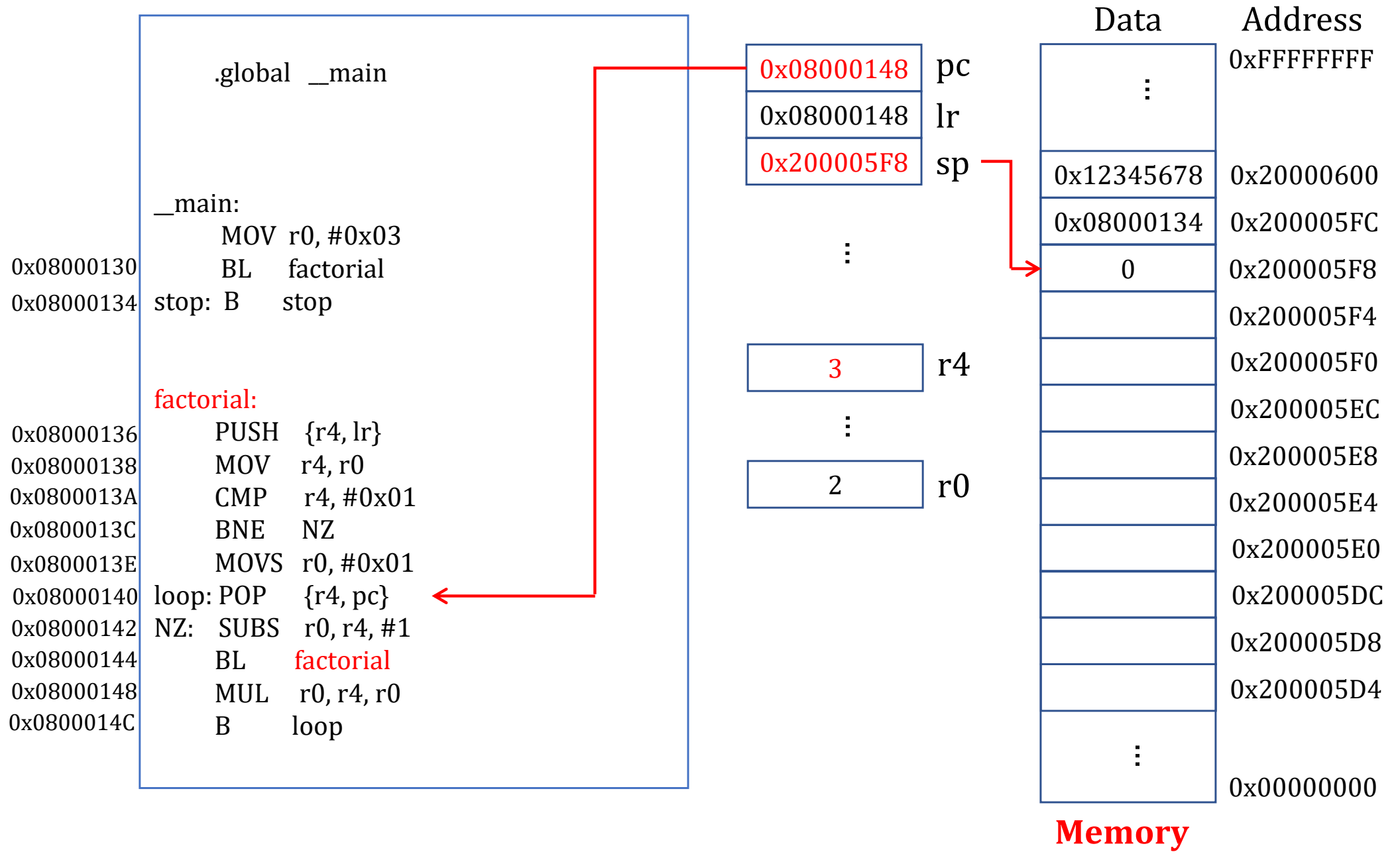
Recursive Factorial in Assembly



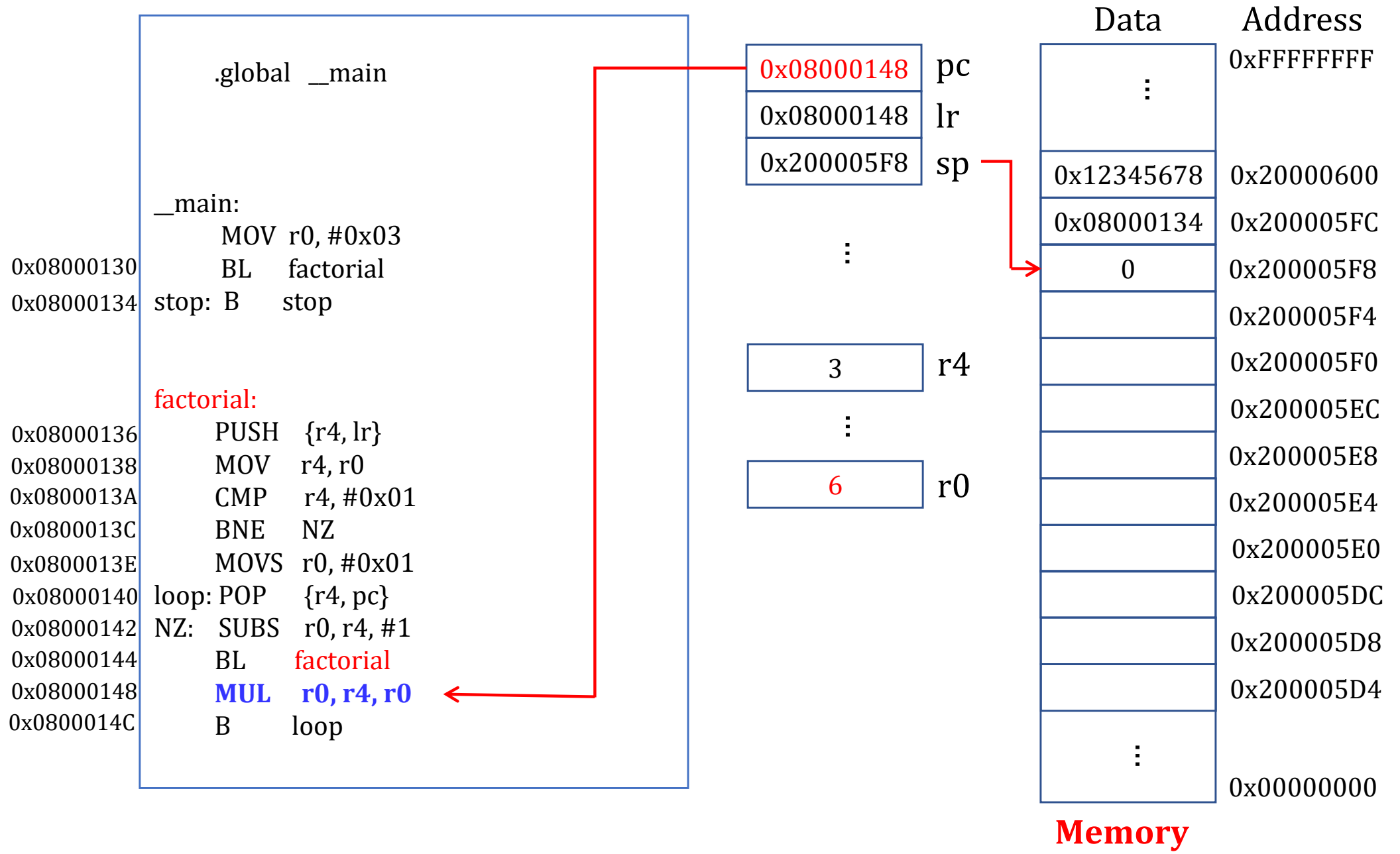
Recursive Factorial in Assembly



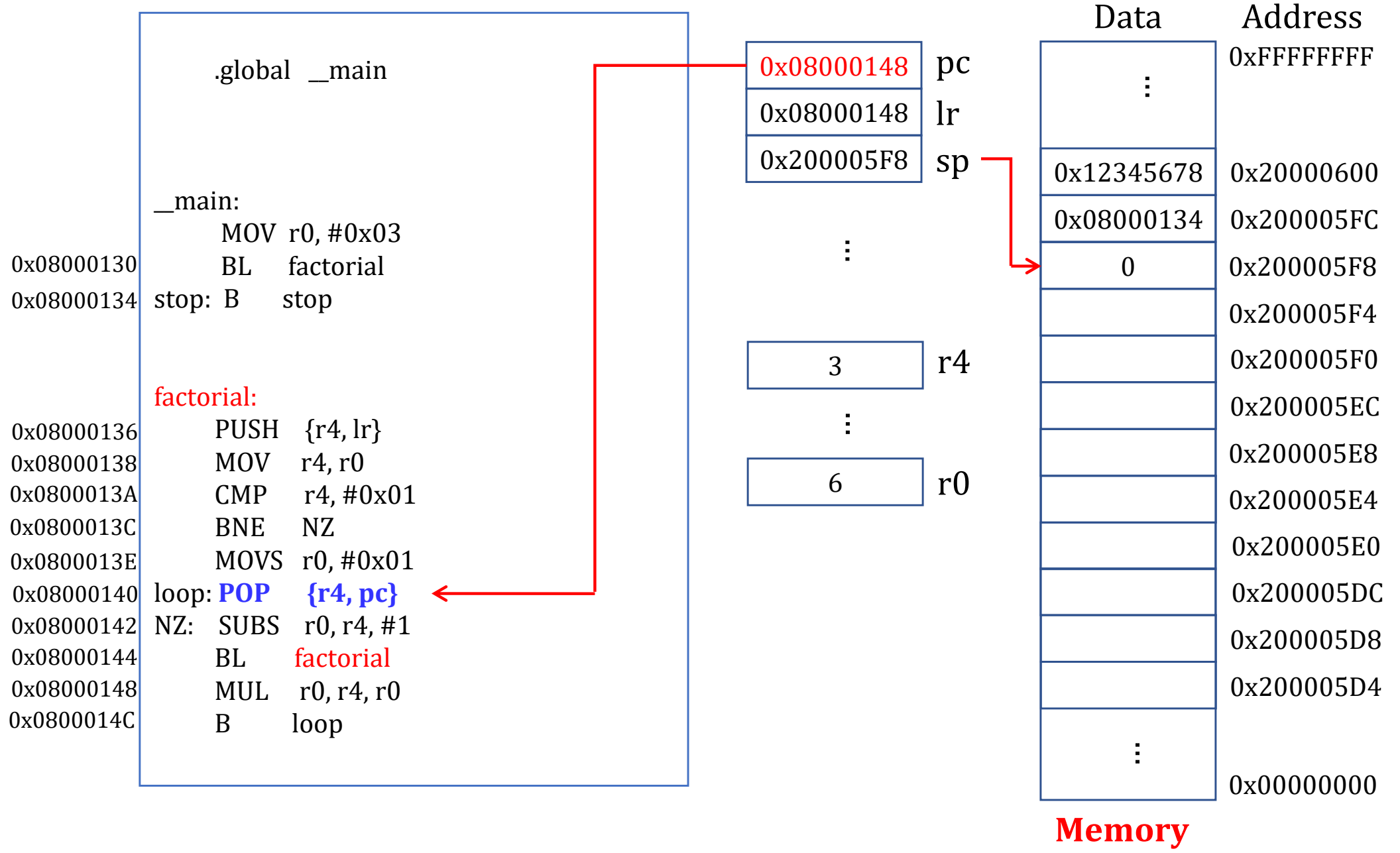
Recursive Factorial in Assembly



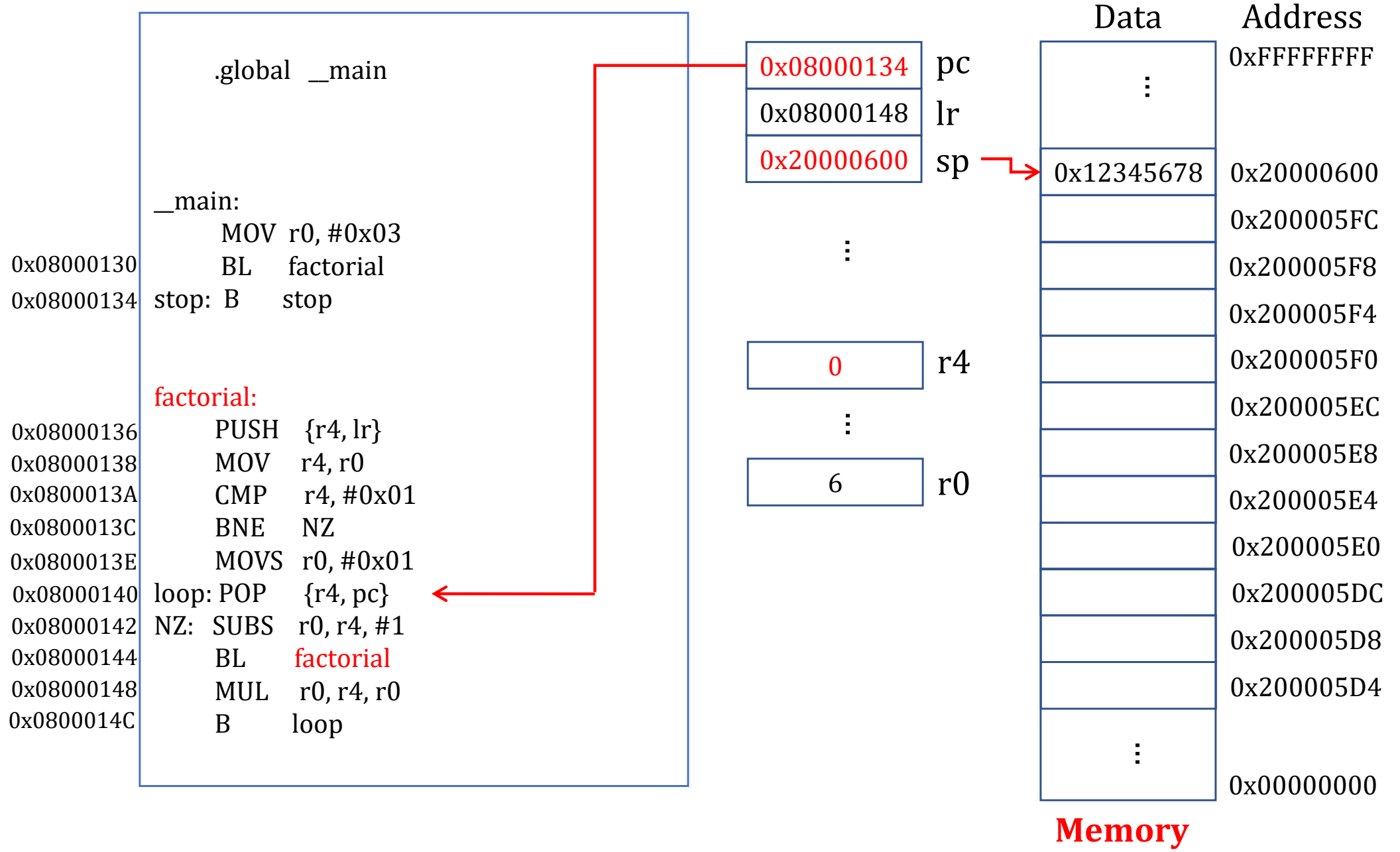
Recursive Factorial in Assembly



Recursive Factorial in Assembly



Recursive Factorial in Assembly



Recursive Factorial in Assembly