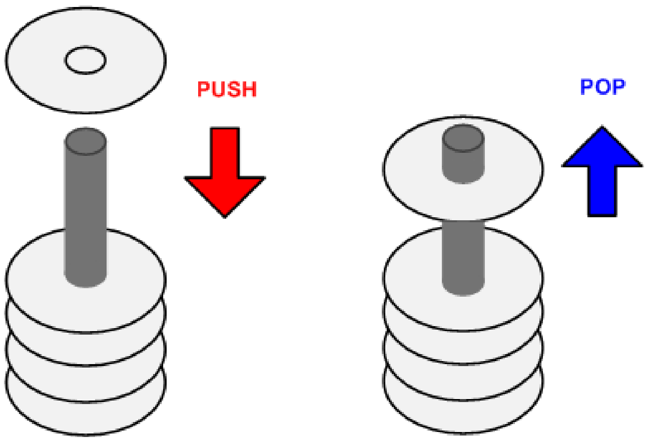
L5 functions

Zonghua Gu, 2018

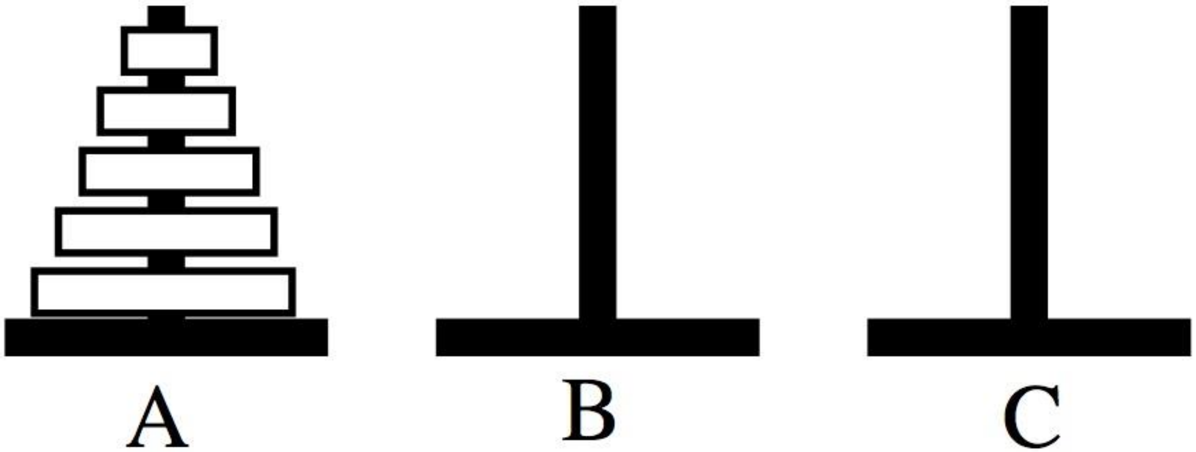
Acknowledgement: some slides taken from Yifeng Zhu's courseware

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)



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

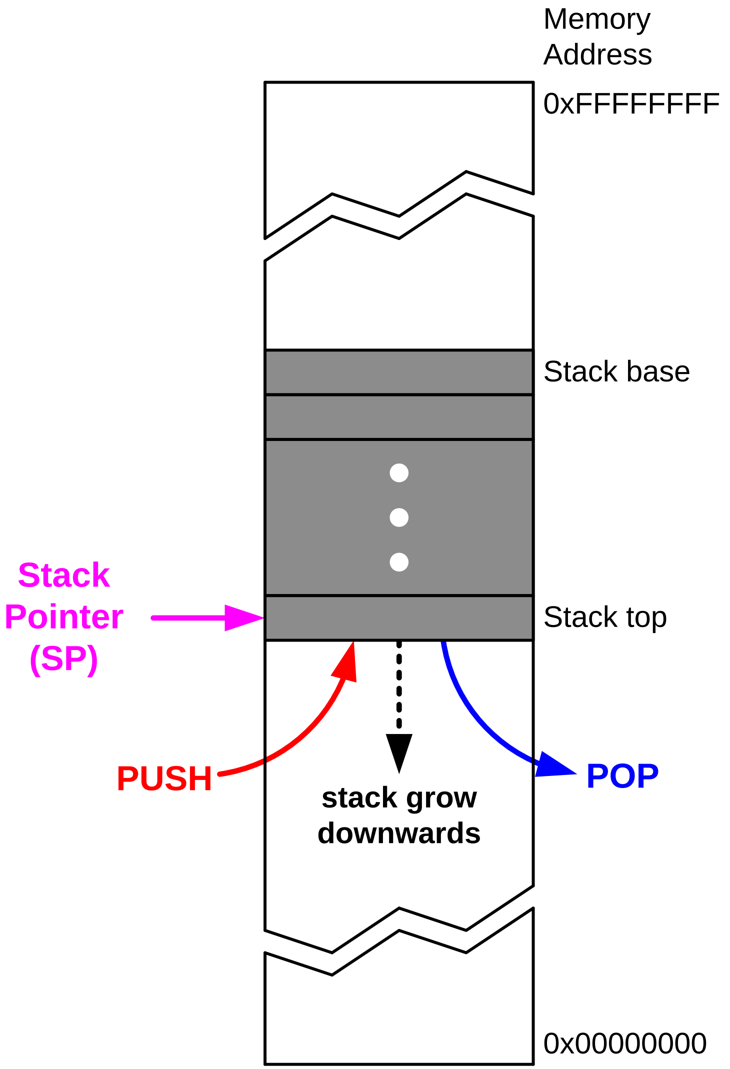
STACK: Last In First Out

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

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

Cortex-M Stack

- Descending stack
  - Grows from top to bottom
- Stack Pointer (SP) (R13) contains memory address of top element of the stack
  - decremented on **PUSH**
  - incremented on **POP**
- The assembly file startup.s defines stack space and initializes SP.



Stack

PUSH $\{Rd\}$

- `SP = SP-4` $\longrightarrow$ Stack grows downward
- `(*SP) = Rd` $\longrightarrow$ Rd is pushed and assigned to top element of stack (C Syntax)

POP $\{Rd\}$

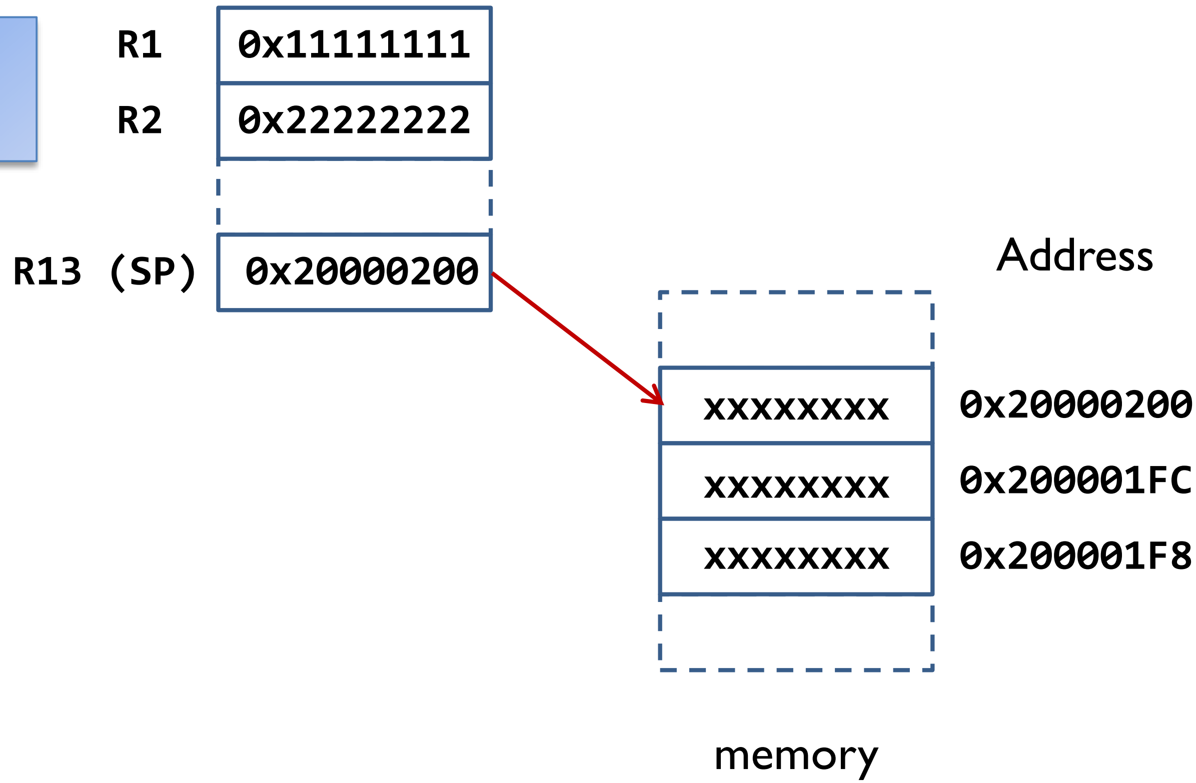
- `Rd = (*SP)` $\longrightarrow$ Top element of stack is popped and assigned to Rd (C Syntax)
- `SP = SP + 4` $\longrightarrow$ Stack shrinks upward

Example: Swap R1 & R2

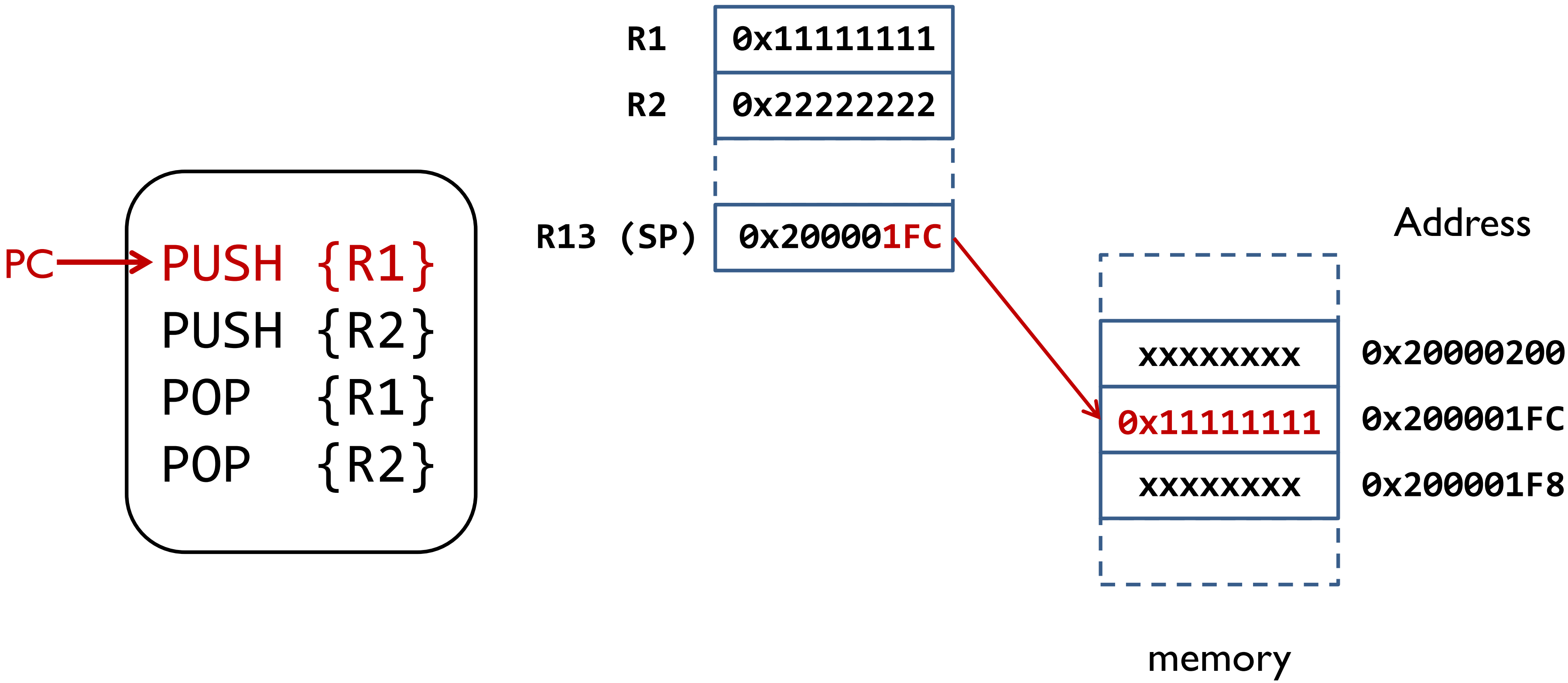
Before execution

R1==0x11111111
R2==0x22222222

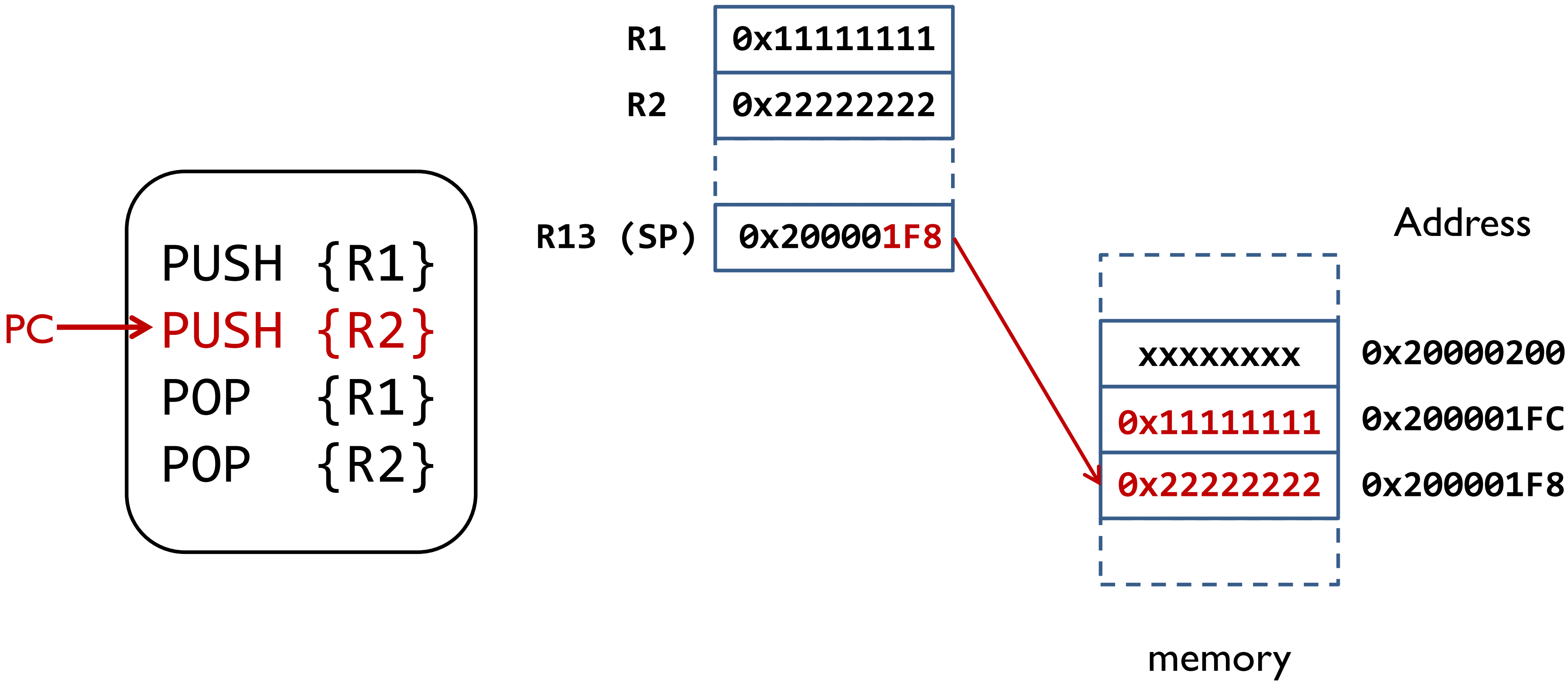
```
PUSH {R1}
PUSH {R2}
POP  {R1}
POP  {R2}
```



Example: Swap R1 & R2

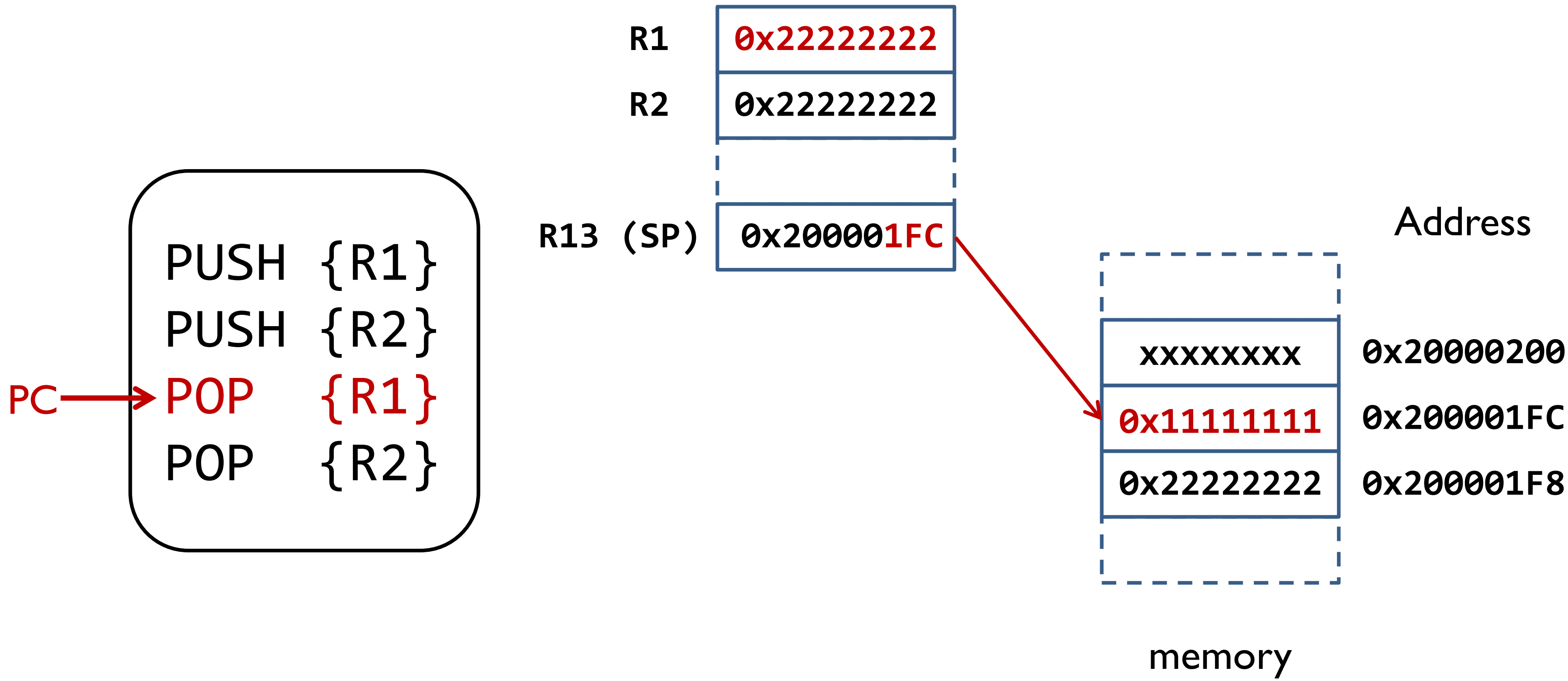


Example: Swap R1 & R2



Example: Swap R1 & R2

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



Example: Swap R1 & R2

Before execution

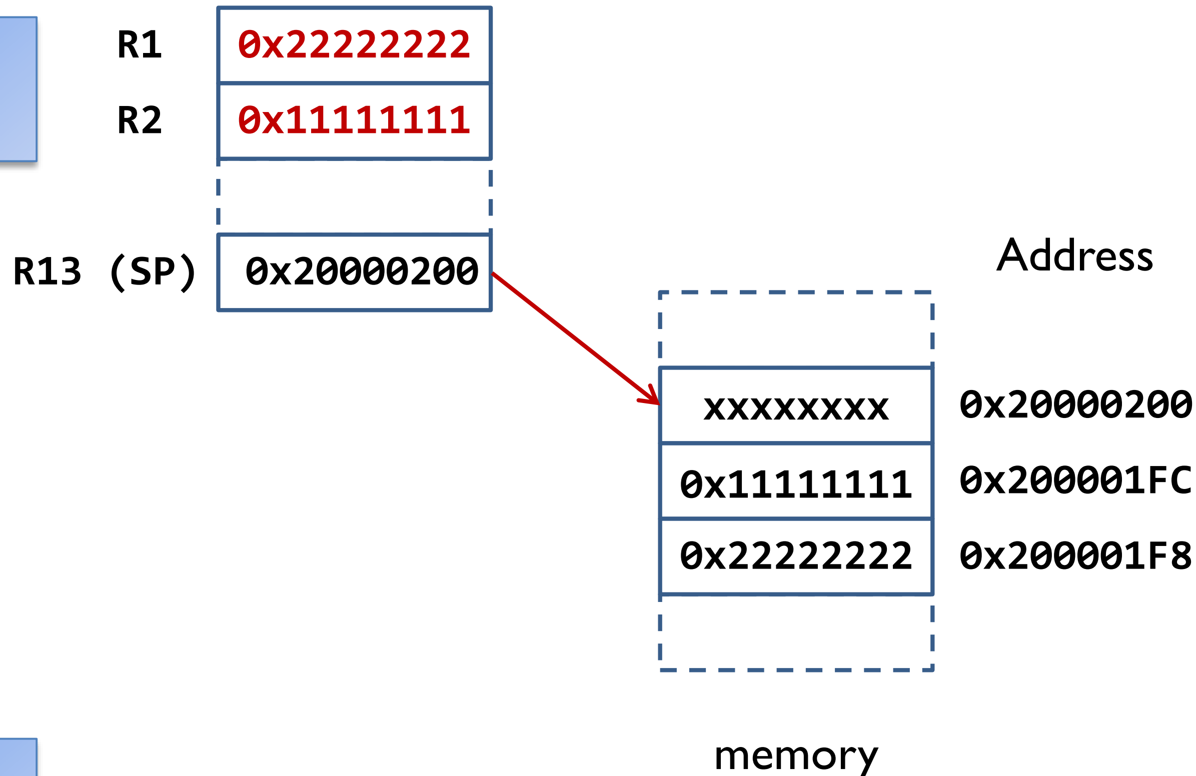
R1==0x11111111
R2==0x22222222

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

PC → POP {R2}

After execution

R1==0x22222222
R2==0x11111111



Not an efficient approach for swapping numbers! Incurs 4 memory accesses. Much more efficient to use another register to do swap.

PUSH/POP Multiple Registers

They are equivalent.

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

They are equivalent.

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

- PUSH/POP multiple registers in a single statement: 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.
  - 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.
- PUSH {register list} followed by POP{register list} leaves the values in register list unchanged. Useful for saving and restoring register values in functions.

Quiz

Are the values of R1 and R2 swapped?

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

Answer: No. This code is equivalent to:

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

This leaves R1 and R2 unchanged

The following programs perform swap:

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

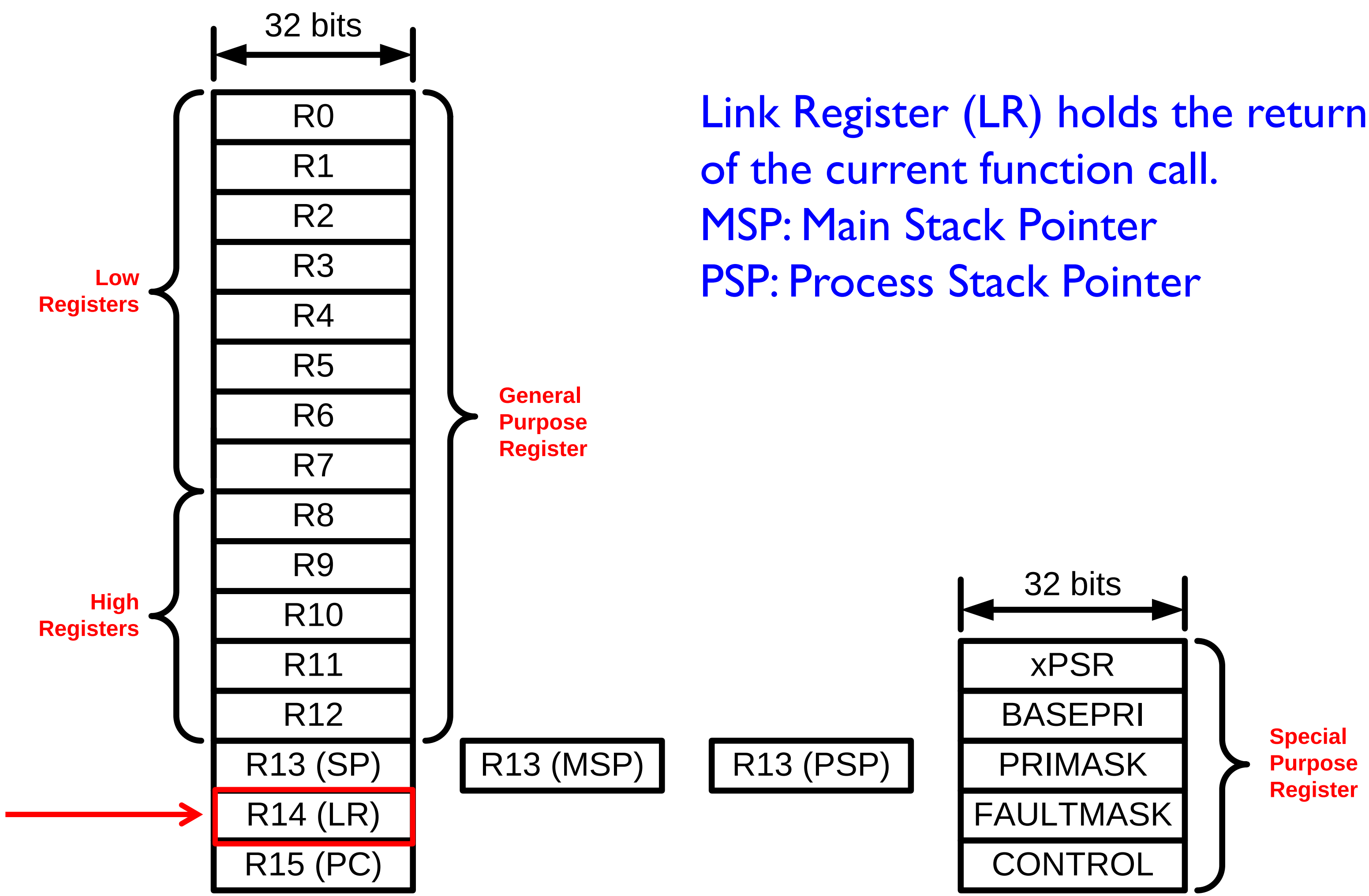
or

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

Function

- A function, also called a subroutine or a procedure
  - Single-entry, single-exit
  - Returns to caller after it exits
- When a function is called, the Link Register (LR) holds the memory address of the next instruction to be executed after the function exits.
  - PC + 4

Link Register



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

MSP: Main Stack Pointer

PSP: Process Stack Pointer

Calling a function: Two Ways

BL label

- Equiv. operation:
  - LR = PC + 4
  - PC = label

BX LR

- Equiv. operation:
  - PC = LR

Caller Program
...
BL foo
...

function/Callee
foo PROC
...
...
BX LR
ENDP

BL label

- Equiv. operation:
  - LR = PC + 4
  - PC = label

PUSH{LR}

...

POP{PC}

- Equiv. operation:
  - PC = LR

Caller Program
...
BL foo
...

function/Callee
foo PROC
PUSH {LR}
...
POP {PC}
ENDP

Notes on Previous Slide

- *foo* is name of function
- The PROC/ENDP directives mark the start/end of a function
  - For programmer's convenience only, they are not present in the final machine code
  - Equivalently: FUNC/ENDFUNC
- The two approaches (BX or PUSH{LR}+POP{PC} in callee) are functionally equivalent
  - BX approach is more efficient since it does not access memory

Calling a function: BX LR

C Code:

```
void foo(void);
Main() {
    •••
    foo();
    •••}
```

Compiler

Assembler Code:

```
 •••
 BL  foo
 •••
```

```
export    foo
foo       PROC
          •••
          BX   LR
          ENDP
```

Calling a function: PUSH(LR)+POP(PC)

C Code:

```
void foo(void);
Main() {
    •••
    foo();
    •••}
```

Compiler

Assembler Code:

```
    • • •
    BL  foo
    • • •
```

```
export     foo
foo        PROC
           PUSH {LR}
           •••
           POP {PC}
           ENDP
```

Calling a function w/ Saving/Restoring R4-R11

Option 1 with BX LR

```
export    foo
foo       PROC
          PUSH {R4,R5}
          • • •
          POP {R4,R5}
          BX   LR
          ENDP
```

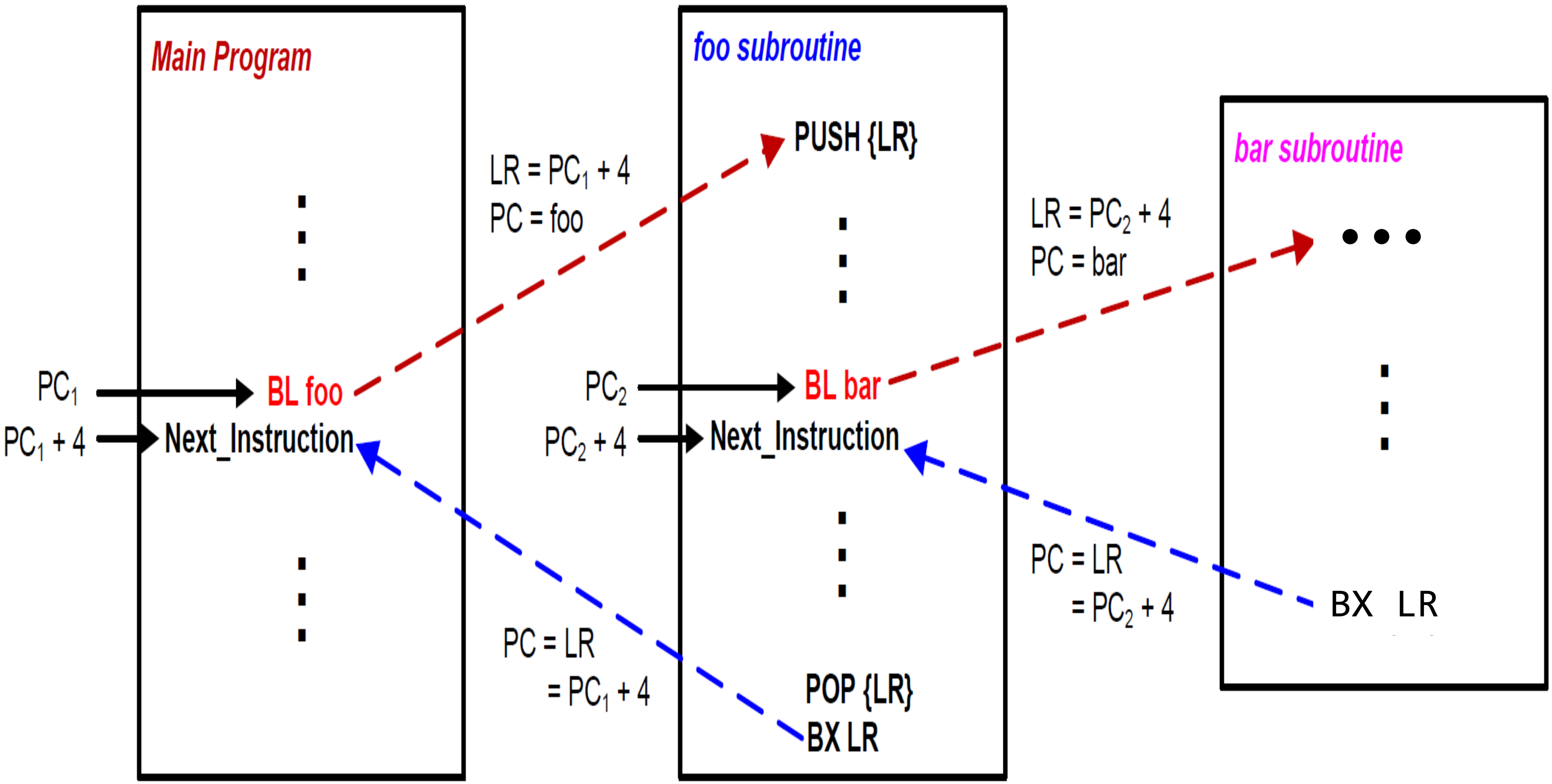
Option 2 with PUSH{LR}+POP{PC}

```
export    foo
foo       PROC
          PUSH {R4,R5, LR}
          • • •
          POP {R4,R5, PC}
          ENDP
```

Since LR is R14, the above is equivalent to:

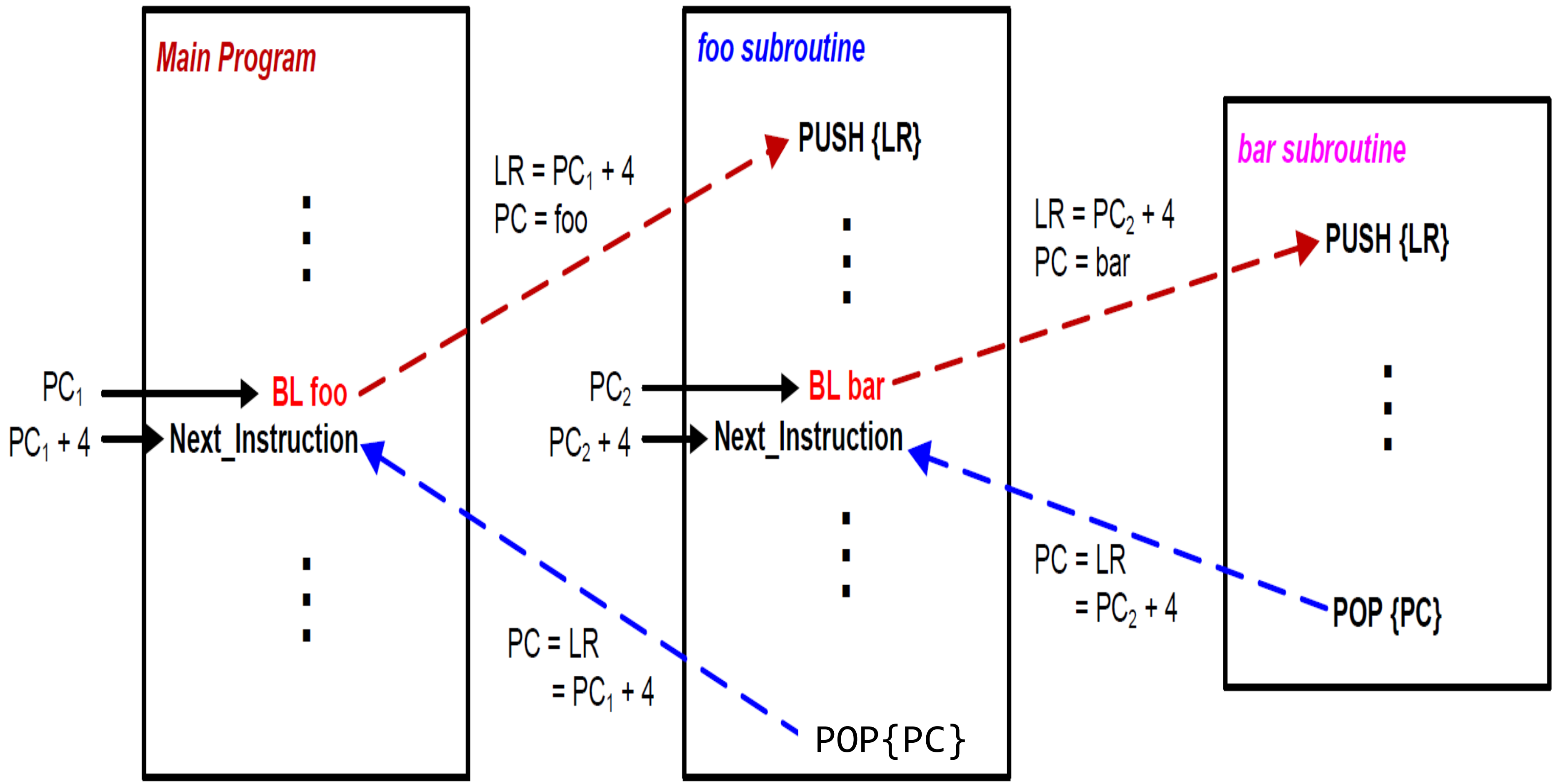
```
export    foo
foo       PROC
          PUSH {LR}
          PUSH {R4,R5}
          • • •
          POP {R4,R5}
          POP {PC}
          ENDP
```

Nested Function Call w/ BX in Callee



Using the BL+BX approach: since register LR can hold only a single return address, a function that calls other functions must preserve LR at its entry point and restore its original content before returning, by using PUSH(LR) and POP(LR) instructions to store and load the first caller's return address LR in memory.

Nested Function Call w/ PUSH(LR)+POP(PC) in Callee



Using the PUSH{LR}+POP{PC} approach, every function has the same form, so you don't have to modify the code if you add one more level of nested function call

Nested Function Call: R0 = $R0^4$

```
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

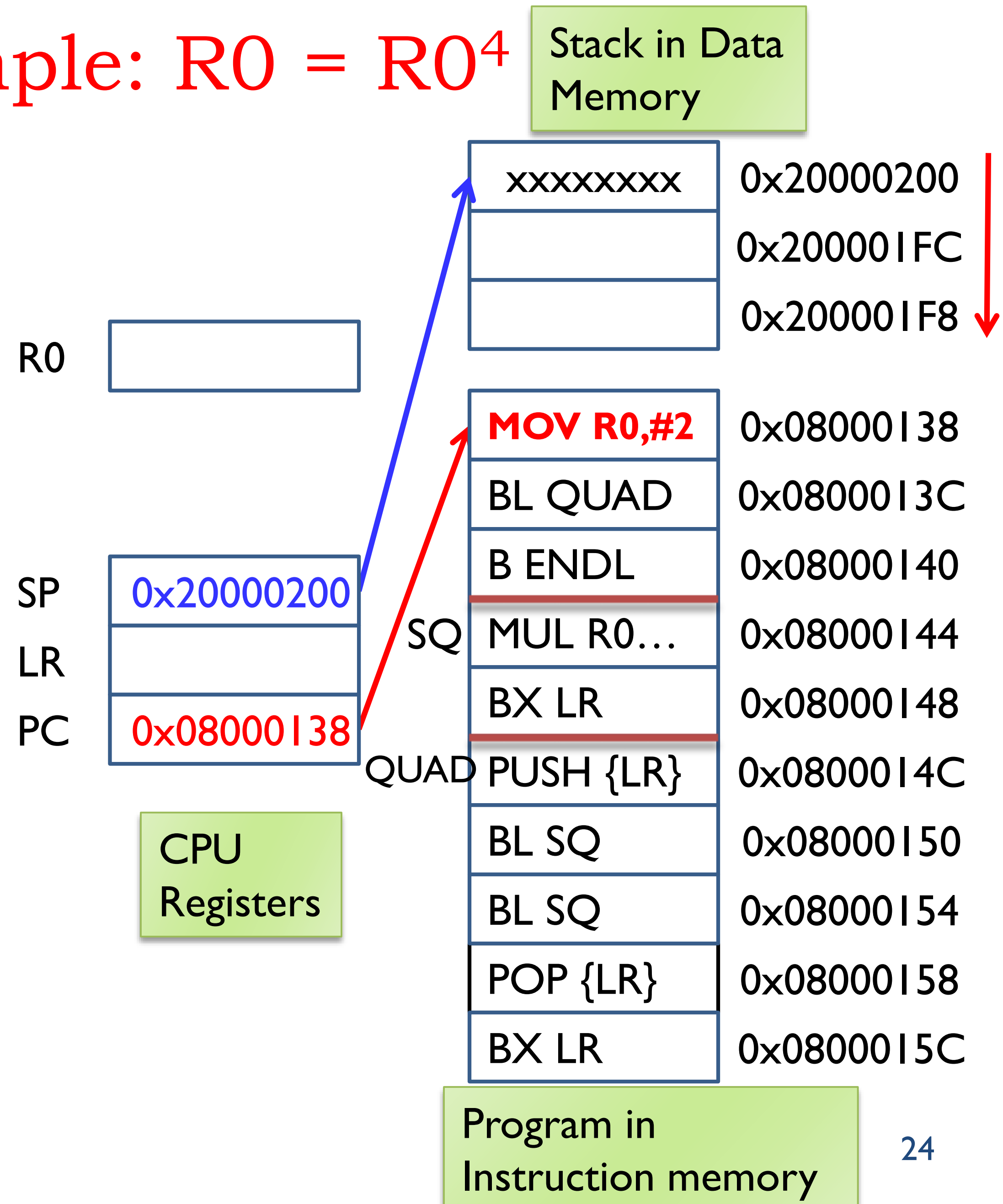
```
SQ       PROC
         MUL R0,R0,R0
         BX LR
         ENDP
```

function SQ

- MAIN calls QUAD, QUAD calls SQ
- Register R0 is used to pass both the argument and the return result (R0 = $R0^4$)
- Assume 32-bit memory address. Every item (register, instruction, data on stack) is 32 bits = 4 Bytes.

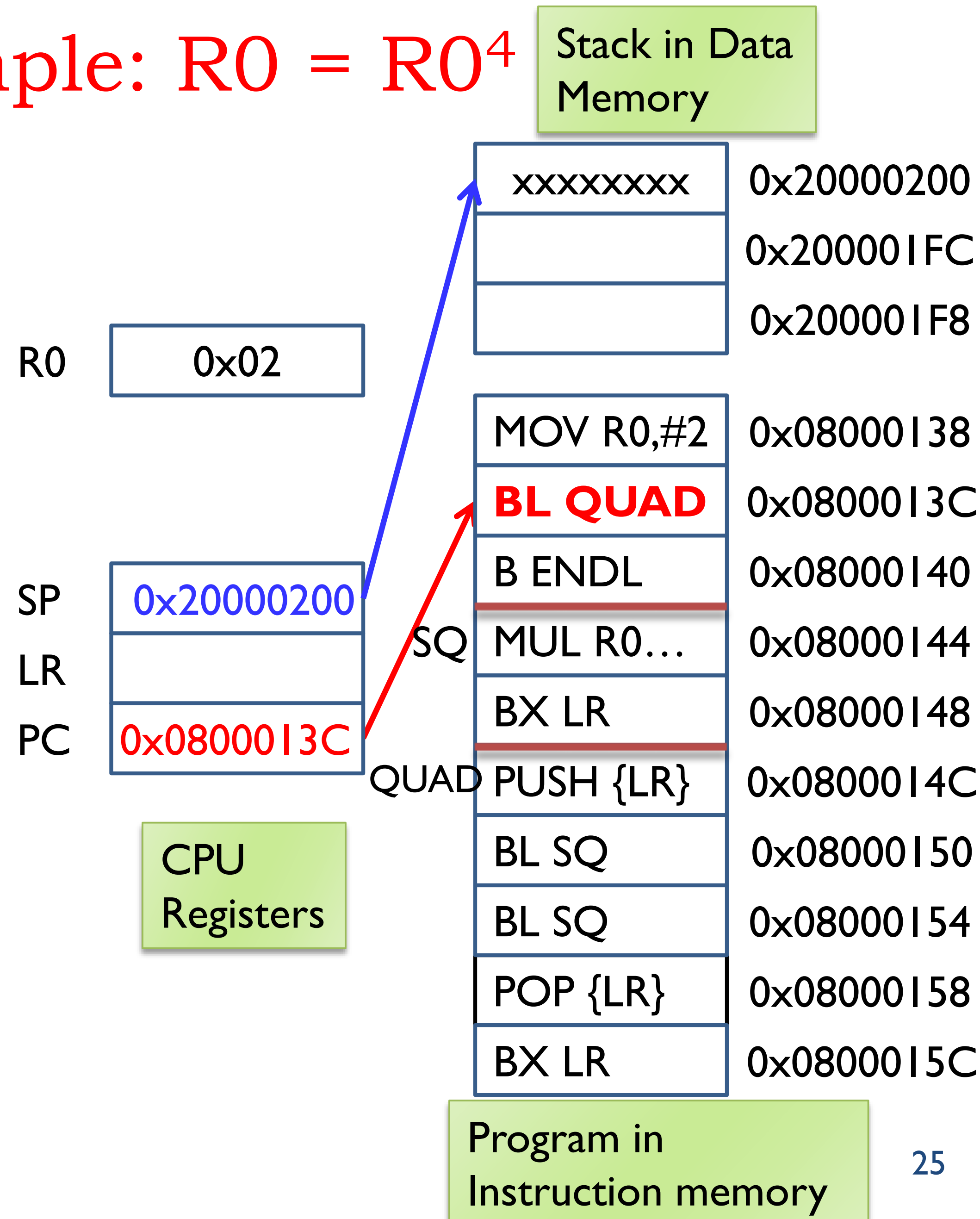
Example: $R0 = R0^4$

```
        MOV  R0,#2
        BL  QUAD
        B  ENDL
---------------------------
SQ      MUL  R0…
        BX  LR
---------------------------
QUAD    PUSH  {LR}
        BL  SQ
        BL  SQ
        POP  {LR}
        BX  LR
ENDL    . . .
```



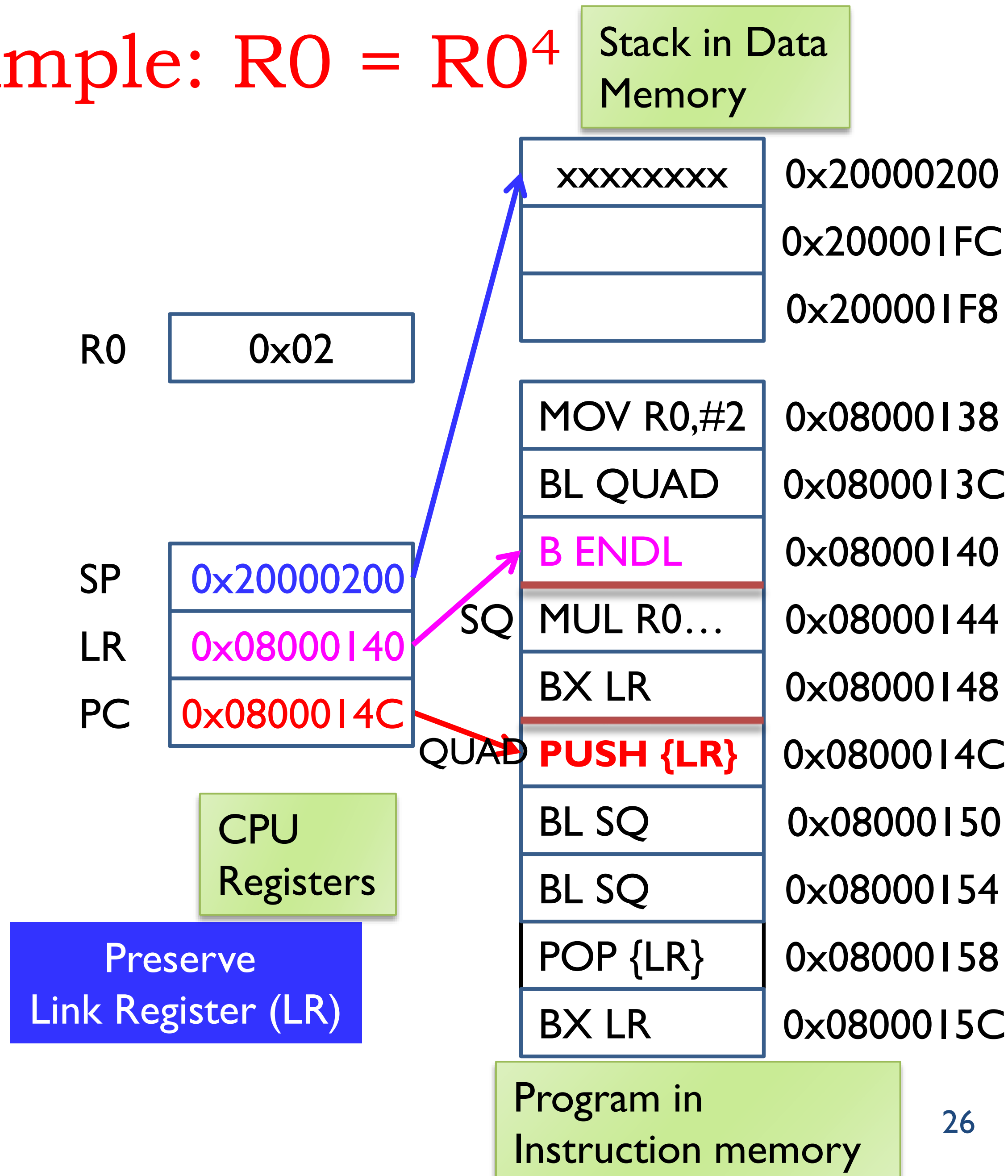
Example: R0 = $R0^4$

```
         MOV R0,#2
         BL QUAD
         B ENDL
SQ       MUL R0…
         BX LR
QUAD     PUSH {LR}
         BL SQ
         BL SQ
         POP {LR}
         BX LR
ENDL     . . .
```



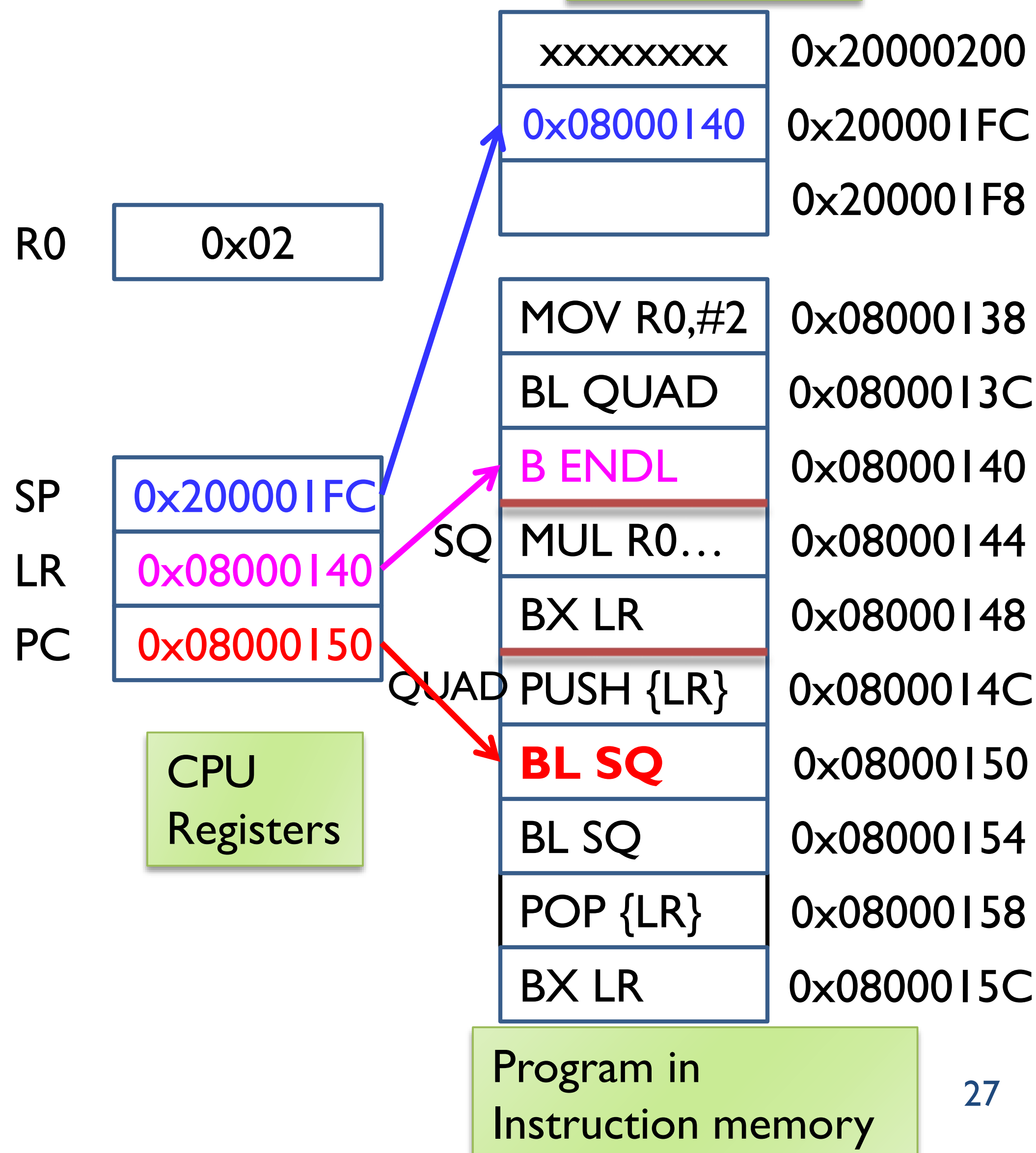
Example: R0 = $R0^4$

```
        MOV  R0,#2
        BL  QUAD
        B  ENDL
---------------------
SQ      MUL  R0…
        BX  LR
---------------------
QUAD    PUSH  {LR}
        BL  SQ
        BL  SQ
        POP  {LR}
        BX  LR
ENDL    ...
```



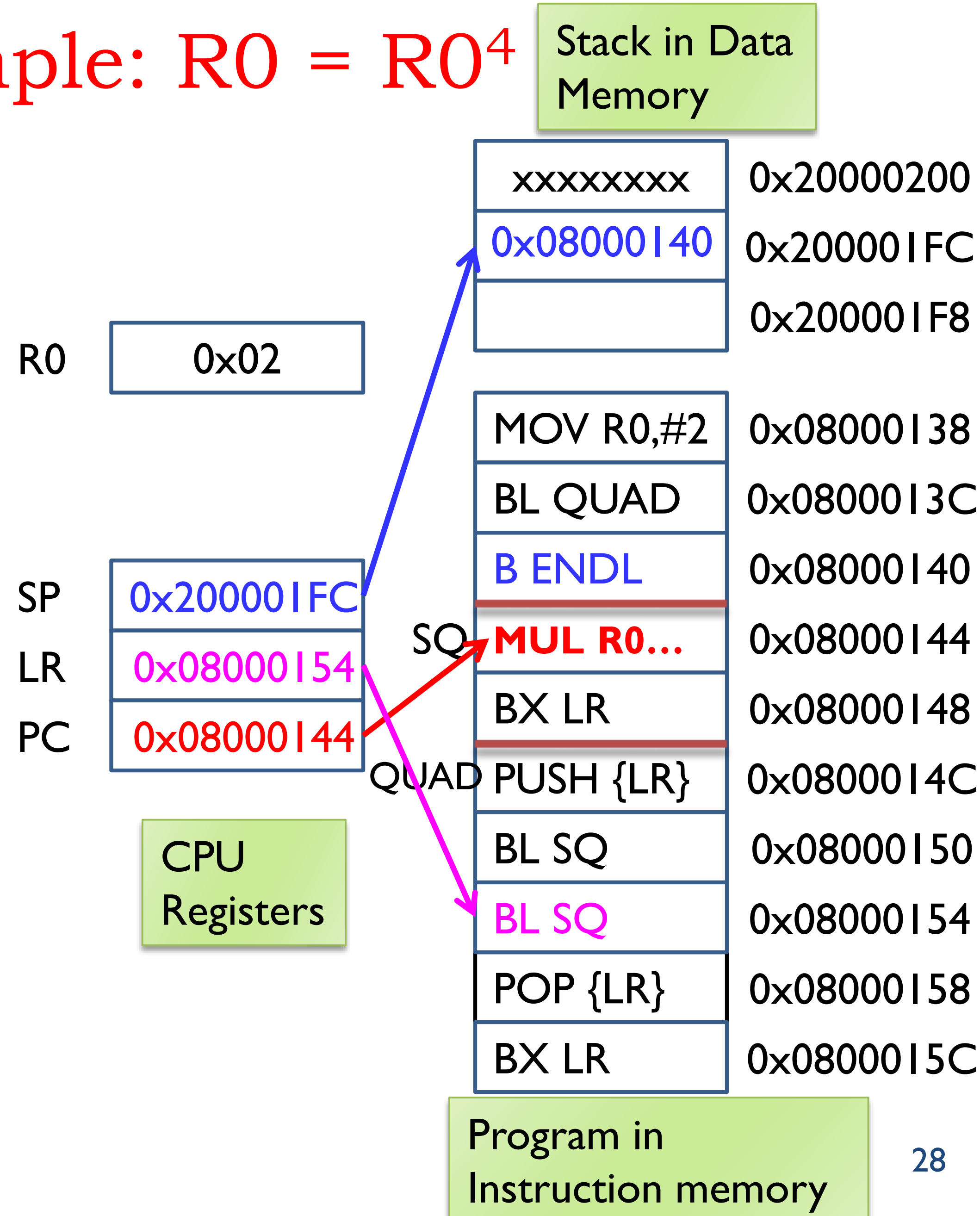
Example: $R0 = R0^4$

```
            MOV  R0,#2
            BL  QUAD
            B  ENDL
-----------------------------
SQ          MUL  R0...
            BX  LR
-----------------------------
QUAD        PUSH  {LR}
            BL  SQ
            BL  SQ
            POP  {LR}
            BX  LR
ENDL        . . .
```



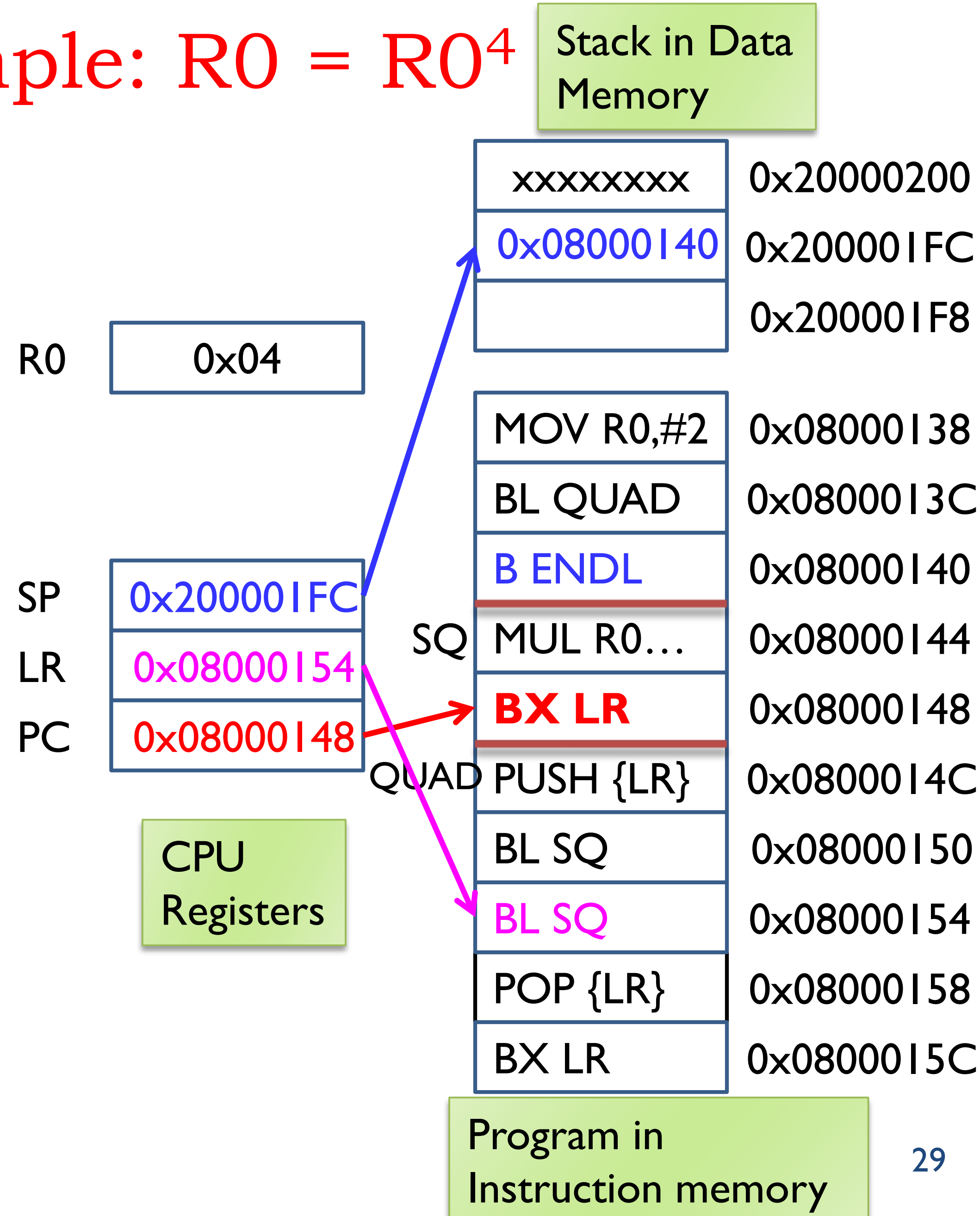
Example: R0 = R0^4

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

SQ       MUL  R0...
         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...
        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…
          BX  LR
------------------------------
QUAD      PUSH  {LR}
          BL  SQ
          BL  SQ
          POP  {LR}
          BX  LR
ENDL      . . .
```

R0	0x04
SP	0x200001FC
LR	0x08000154
PC	0x08000154

CPU Registers

Stack in Data Memory

Contents	Address
xxxxxxxx	0x20000200
0x08000140	0x200001FC
	0x200001F8

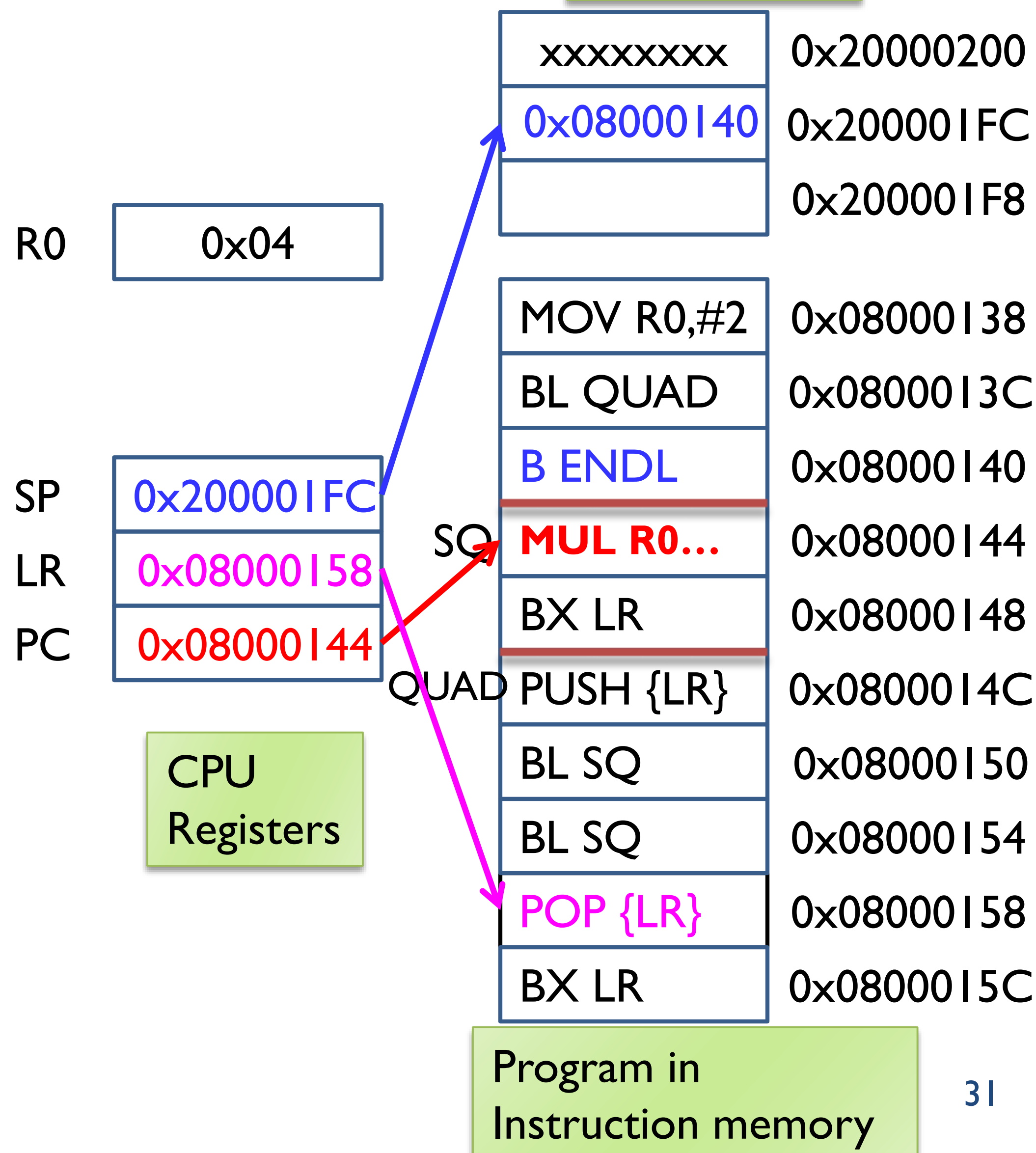
Label	Instruction	Address
	MOV R0,#2	0x08000138
	BL QUAD	0x0800013C
	B ENDL	0x08000140
SQ	MUL R0…	0x08000144
	BX LR	0x08000148
QUAD	PUSH {LR}	0x0800014C
	BL SQ	0x08000150
	BL SQ	0x08000154
	POP {LR}	0x08000158
	BX LR	0x0800015C

Program in Instruction memory

Example: R0 = R0^4

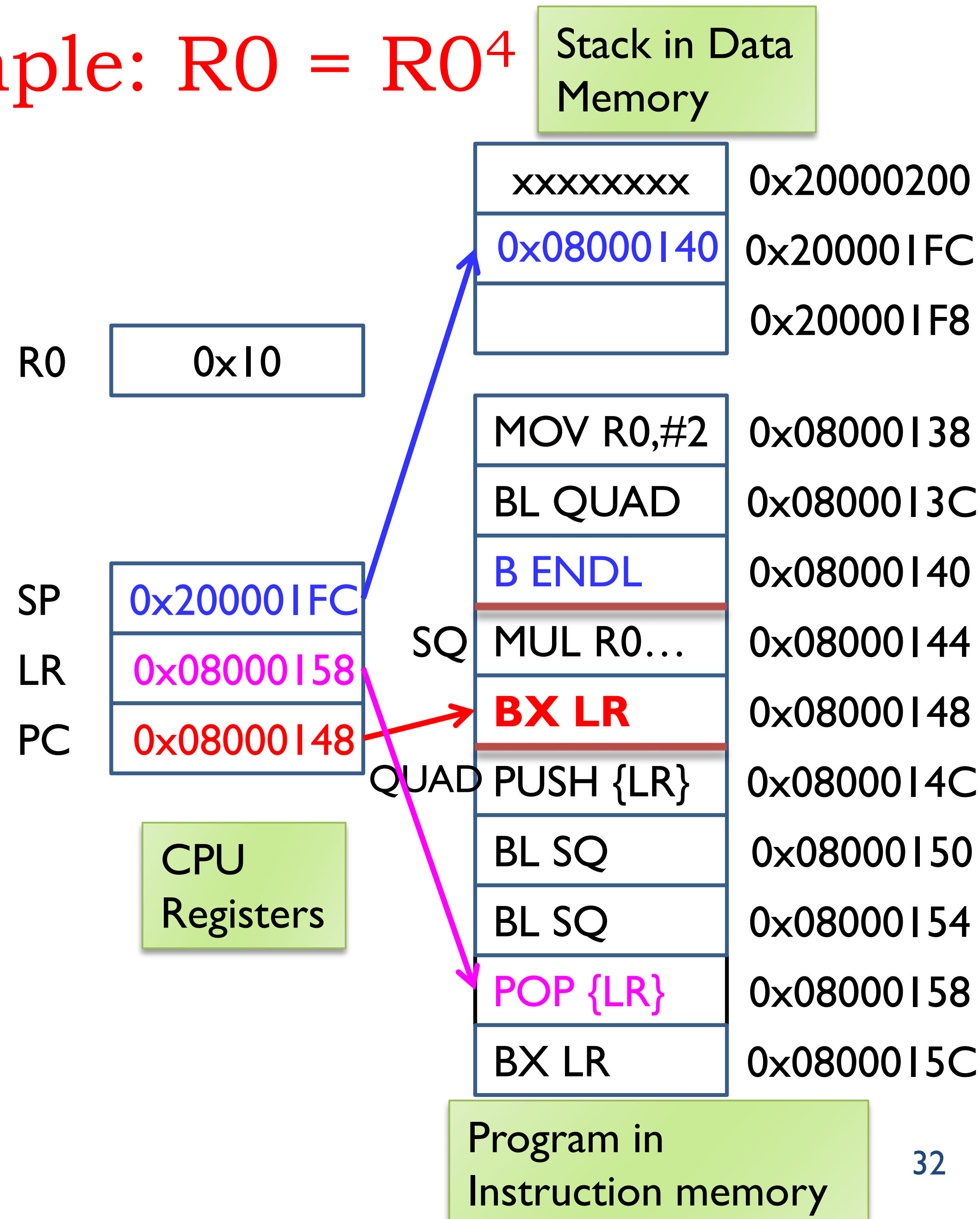
```
        MOV  R0,#2
        BL  QUAD
        B  ENDL
-------------------------
SQ      MUL  R0...
        BX  LR
-------------------------
QUAD    PUSH  {LR}
        BL  SQ
        BL  SQ
        POP  {LR}
        BX  LR
ENDL    ...
```

Stack in Data Memory



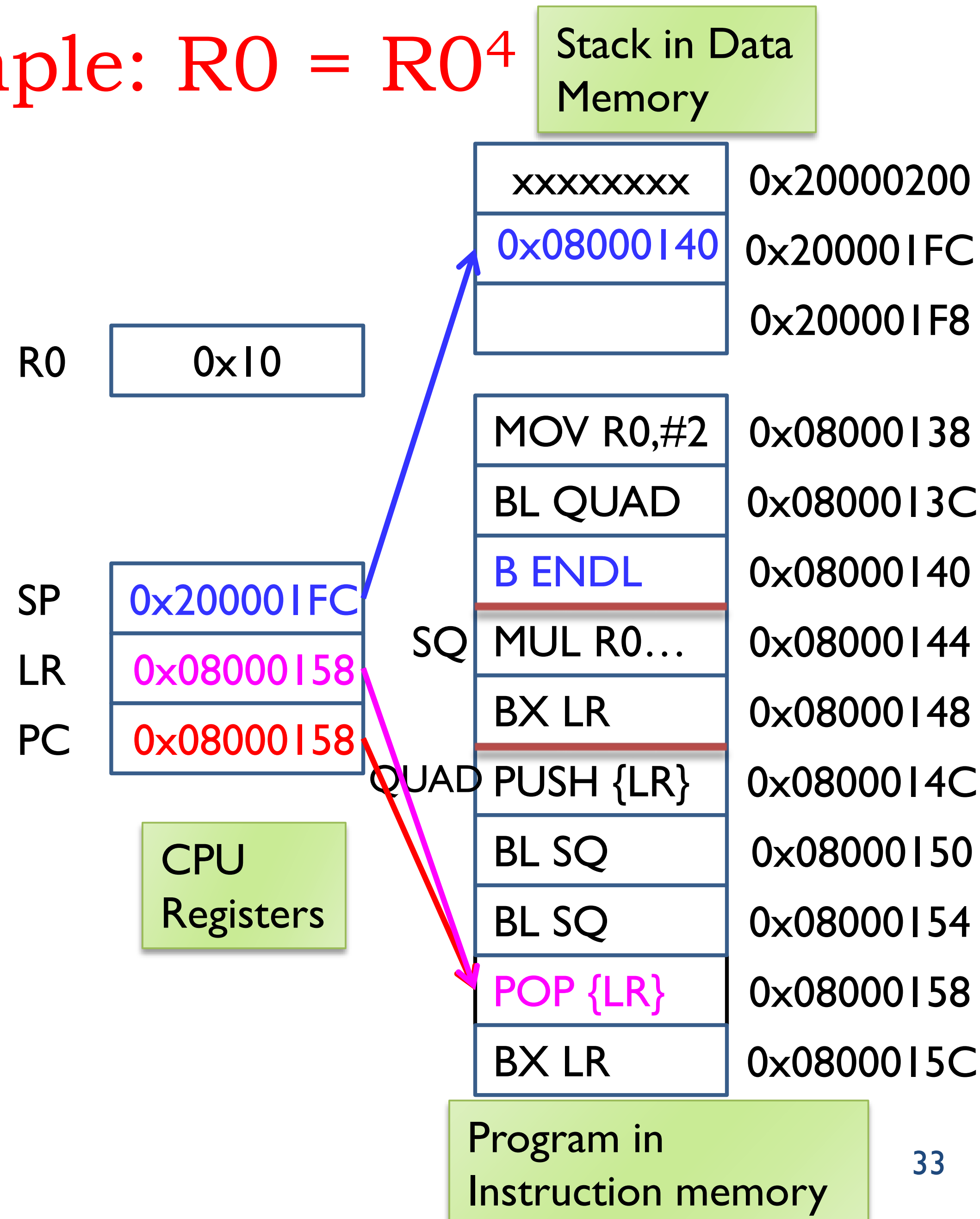
Example: $R0 = R0^4$

```
        MOV R0,#2
        BL  QUAD
        B   ENDL
-----------------------
SQ      MUL 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...
        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...
            BX  LR
-------------------------
QUAD        PUSH  {LR}
            BL  SQ
            BL  SQ
            POP  {LR}
            BX  LR
ENDL        ...
```

CPU Registers

R0	0x10
SP	0x20000200
LR	0x08000140
PC	0x0800015C

Recover Link Register (LR)

Stack in Data Memory

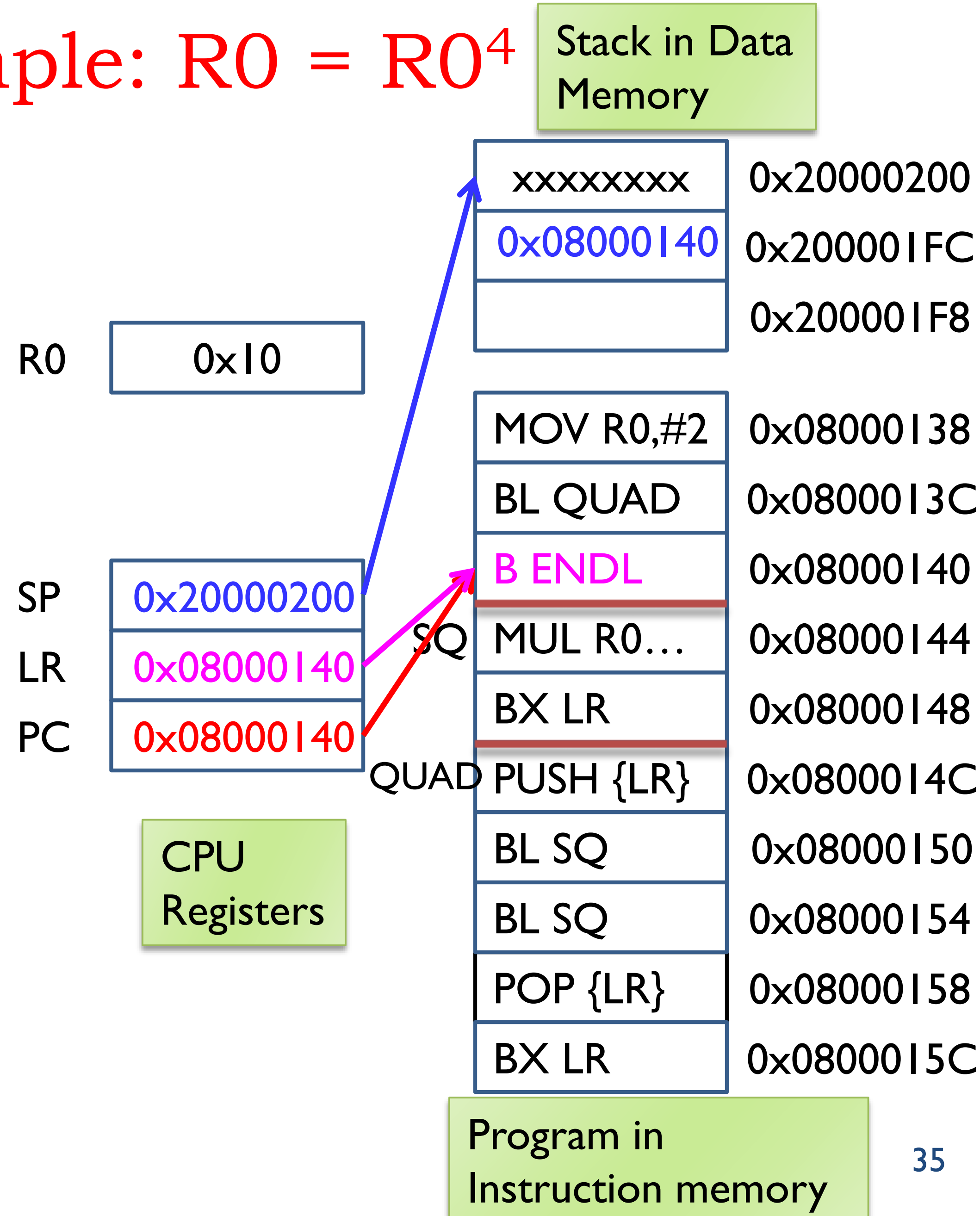
xxxxxxxx	0x20000200
0x08000140	0x200001FC
	0x200001F8

Program in Instruction memory

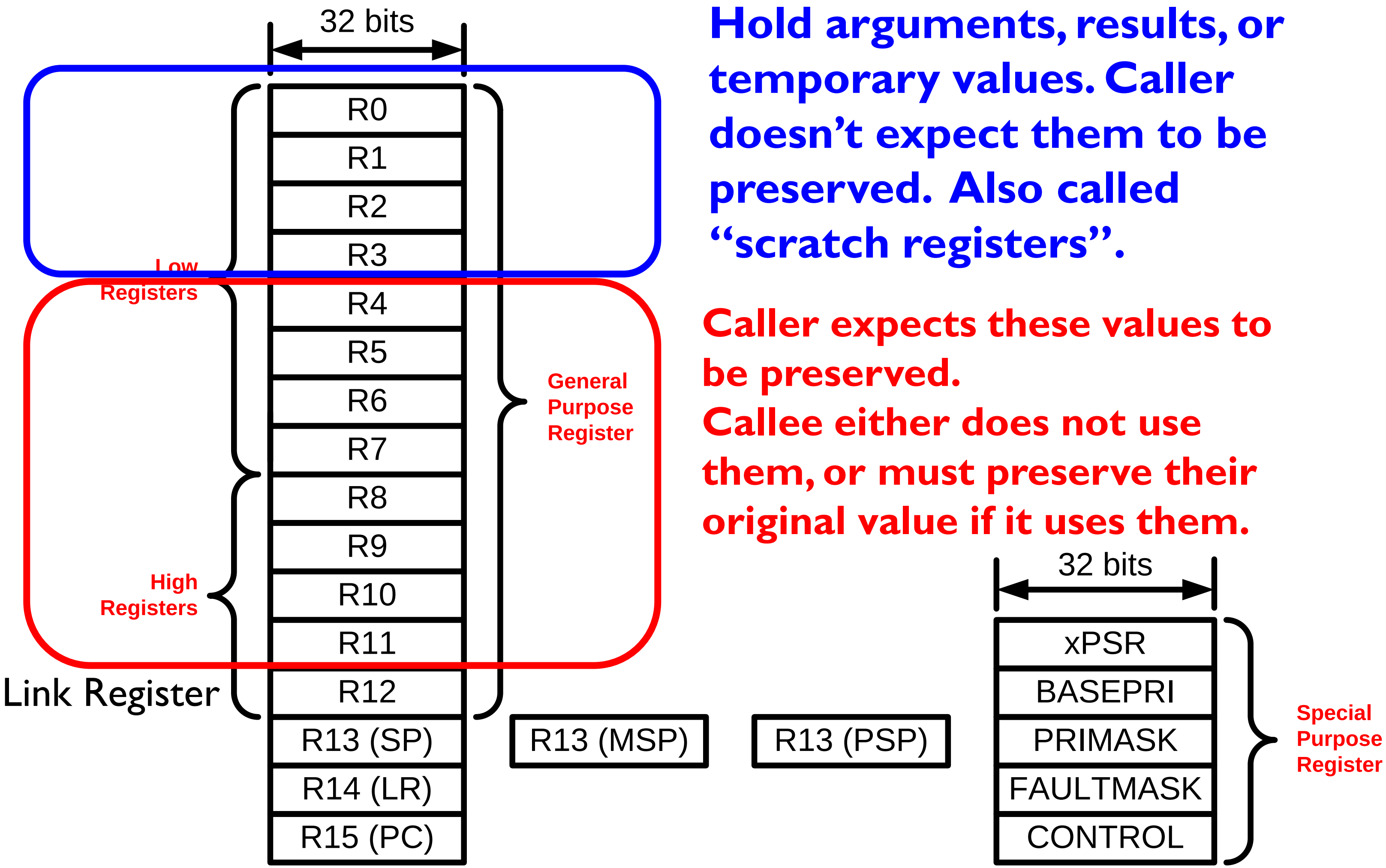
Label	Instruction	Address
	MOV R0,#2	0x08000138
	BL QUAD	0x0800013C
	B ENDL	0x08000140
SQ	MUL R0…	0x08000144
	BX LR	0x08000148
QUAD	PUSH {LR}	0x0800014C
	BL SQ	0x08000150
	BL SQ	0x08000154
	POP {LR}	0x08000158
	BX LR	0x0800015C

Example: R0 = R0^4

```
        MOV  R0,#2
        BL  QUAD
        B  ENDL
--------------------------
SQ      MUL  R0...
        BX  LR
--------------------------
QUAD    PUSH  {LR}
        BL  SQ
        BL  SQ
        POP  {LR}
        BX  LR
ENDL    ...
```



Arguments and Return Values



Hold arguments, results, or temporary values. Caller doesn't expect them to be preserved. Also called "scratch registers".

Caller expects these values to be preserved.
Callee either does not use them, or must preserve their original value if it uses them.

On Conventions

- "Callee must preserve values of R4-R11, but not R0-R3"
  - This is a programming convention adopted by ARM compilers and assemblers that helps with interoperability among software written by different people (the processor hardware doesn't care how you use the registers)
  - Like driving on the right side of the road in North America (the road doesn't care how cars are driven on it)
- What if you don't respect this convention?
  - If you write everything (assembler code, compiler toolchain…) by yourself, free to adopt your own convention, but your code or tool will not interoperate with the rest of the world

Calling a Function without Preserving R4-R11

Caller Program	function/Callee
`MOV R4, #100` `...` **`BL foo`** **`% R4 has the incorrect value of 10`** `...` `ADD R4, R4, #1`	`foo PROC` `...` `MOV R4, #10 ; foo changes R4` `...` **`BX LR`** `ENDP`

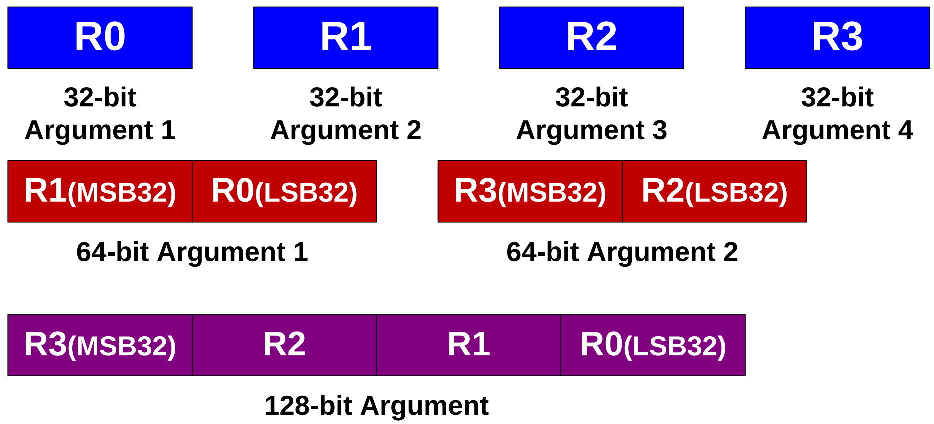
Callee must preserve values of R4-R11

Calling a Function, Preserving r4-r11

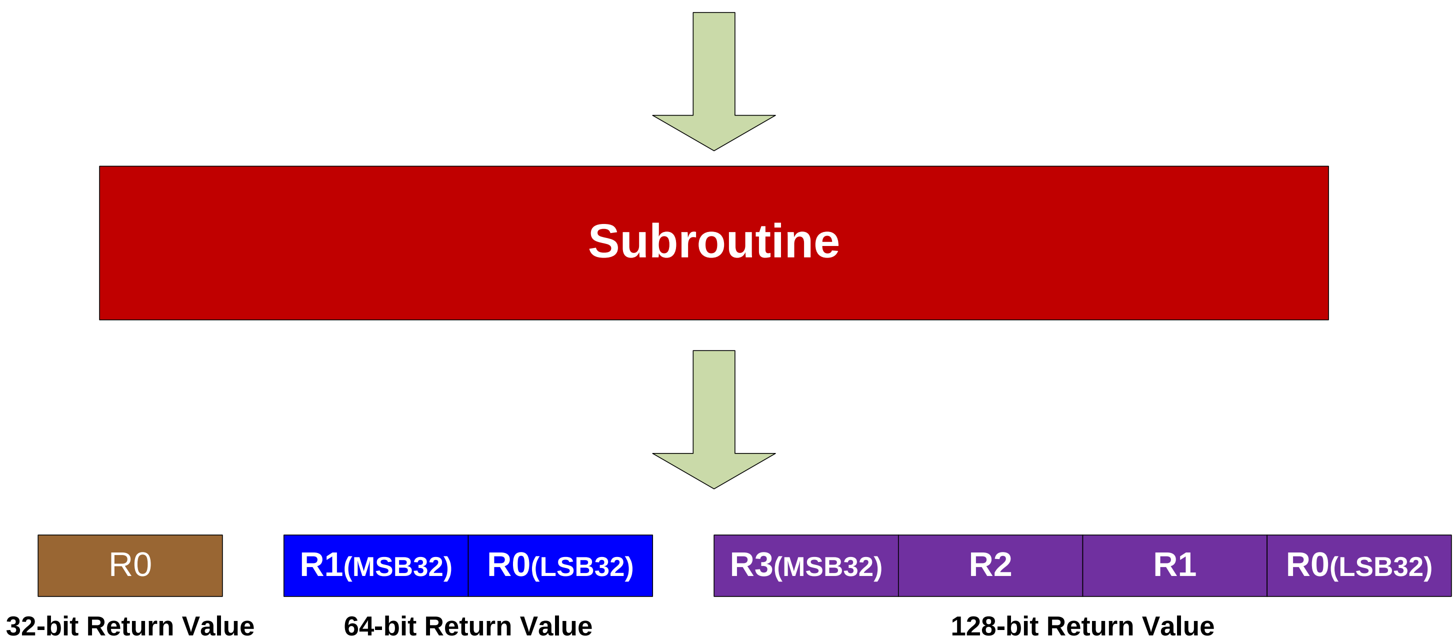
Caller Program	function/Callee
 MOV R4, #100 ... **BL foo** **% R4 has the correct value of 100** ... ADD R4, R4, #1	foo PROC **PUSH {R4}** ; preserve R4 ... MOV R4, #10 ; foo changes R4 ... **POP {R4}** ; Recover R4 BX LR ENDP

Callee must preserve values of R4-R11

Passing Arguments via Registers R0-R3



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.



Additional Arguments Passed on Stack

Registers

R0	R1	R2	R3

Stack in Memory

foo (int i0, int i1, int i2, int i3)

Registers

i0	i1	i2	i3

Stack in Memory

foo (int i0, char a1, double D)

Each argument of 8-bit char, or 16-bit short, is passed in a 32-bit register

Registers

i0	a1	D

Stack in Memory

foo (int i0, int i1, double D, int i2, int i3)

Registers

i0	i1	D

Stack in Memory

i2	i3					

Caller passes arguments i0, i1, D in registers R0-R3 directly; pushes additional arguments i2 and i3 onto the stack before function call (details not covered in this lecture)

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

Redundancy?

- This program does not use the stack
  - No nested function calls
  - All arguments fit in R0-R3
- Why do you have "MOV R0, R2" in the callee, and "MOV R2,R0" in the caller? Aren't they redundant?
  - Return value must be in R0, hence must have "MOV R0, R2" in the callee before return
  - The specification says "R2=R0*R0+R1*R1", hence "MOV R2, R0" in the caller after function return
- Actually the program can be rewritten to improve efficiency. See next slide.

Example simplified: R0 = R0*R0+R1*R1

This version uses fewer registers and is more efficient.
But we use the previous version for illustration purpose

```
        MOV R0,#3
        MOV R1,#4
        BL  SSQ

        B ENDL
        ...
SSQ     MUL R0,R0,R0
        MUL R1,R1,R1
        ADD R0,R0,R1

        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    ...
```

R0	
R1	
R2	
R3	
LR	
PC	0x08000128

CPU Registers

		Memory Address
	MOV R0,#3	0x08000128
	MOV R1,#4	0x0800012C
	BL SSQ	0x0800012F
	MOV R2,R0	0x08000134
	B ENDL	0x08000138
SSQ	MUL R2,...	0x0800013B
	MUL R3,...	0x0800013F
	ADD R2,R3	0x08000144
	MOV R0,R2	0x08000146
	BX LR	0x0800014A

Program in Instruction memory

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    ...
```

Register	Value
R0	3
R1	
R2	
R3	
LR	
PC	0x08000128

CPU Registers

	Instruction	Memory Address
	MOV R0,#3	0x08000128
	MOV R1,#4	0x0800012C
	BL SSQ	0x0800012F
	MOV R2,R0	0x08000134
	B ENDL	0x08000138
SSQ	MUL R2,...	0x0800013B
	MUL R3,...	0x0800013F
	ADD R2,R3	0x08000144
	MOV R0,R2	0x08000146
	BX LR	0x0800014A

Program in Instruction memory

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	0x0800012B

CPU Registers

Label	Instruction	Memory Address
	MOV R0,#3	0x08000128
	MOV R1,#4	0x0800012C
	BL SSQ	0x0800012F
	MOV R2,R0	0x08000134
	B ENDL	0x08000138
SSQ	MUL R2,...	0x0800013B
	MUL R3,...	0x0800013F
	ADD R2,R3	0x08000144
	MOV R0,R2	0x08000146
	BX LR	0x0800014A

Program in Instruction memory

Example: R2 = R0*R0+R1*R1

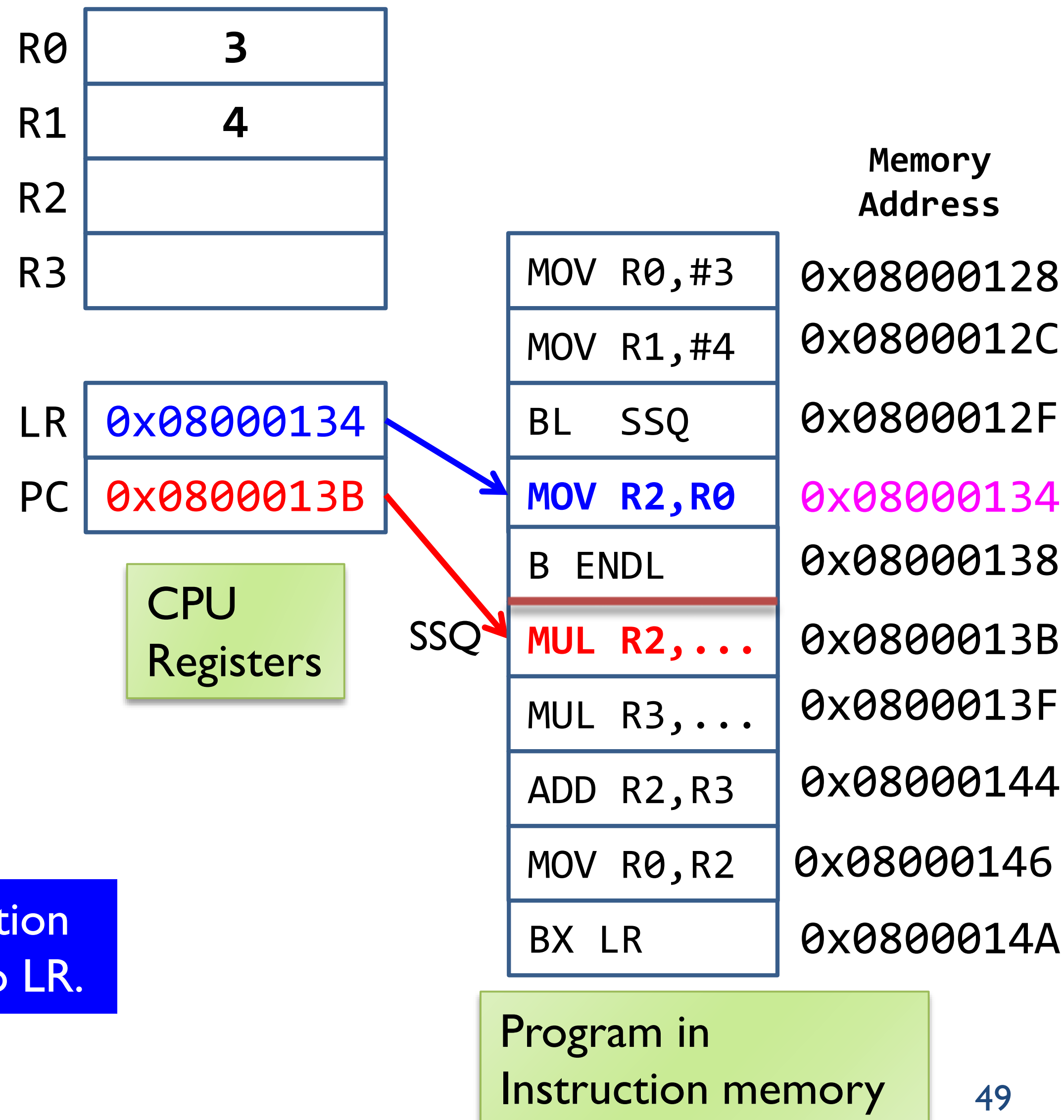
```
        MOV  R0,#3
        MOV  R1,#4
        BL   SSQ
        MOV  R2,R0
        B  ENDL

SSQ     MUL  R2,R0,R0
        MUL  R3,R1,R1
        ADD  R2,R2,R3
        MOV  R0,R2
        BX   LR
ENDL    ...
```

Register	Value
R0	3
R1	4
R2	
R3	
LR	
PC	0x0800012F

Label	Instruction	Memory Address
	MOV R0,#3	0x08000128
	MOV R1,#4	0x0800012C
	BL SSQ	0x0800012F
	MOV R2,R0	0x08000134
	B ENDL	0x08000138
SSQ	MUL R2,...	0x0800013B
	MUL R3,...	0x0800013F
	ADD R2,R3	0x08000144
	MOV R0,R2	0x08000146
	BX LR	0x0800014A

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   ...
```



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

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    ...
```

Register	Value
R0	3
R1	4
R2	9
R3	

Register	Value
LR	0x08000134
PC	0x0800013B

CPU Registers

Label	Instruction	Memory Address
	MOV R0,#3	0x08000128
	MOV R1,#4	0x0800012C
	BL SSQ	0x0800012F
	MOV R2,R0	0x08000134
	B ENDL	0x08000138
SSQ	MUL R2,...	0x0800013B
	MUL R3,...	0x0800013F
	ADD R2,R3	0x08000144
	MOV R0,R2	0x08000146
	BX LR	0x0800014A

Program in Instruction memory

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	0x0800013F

CPU Registers

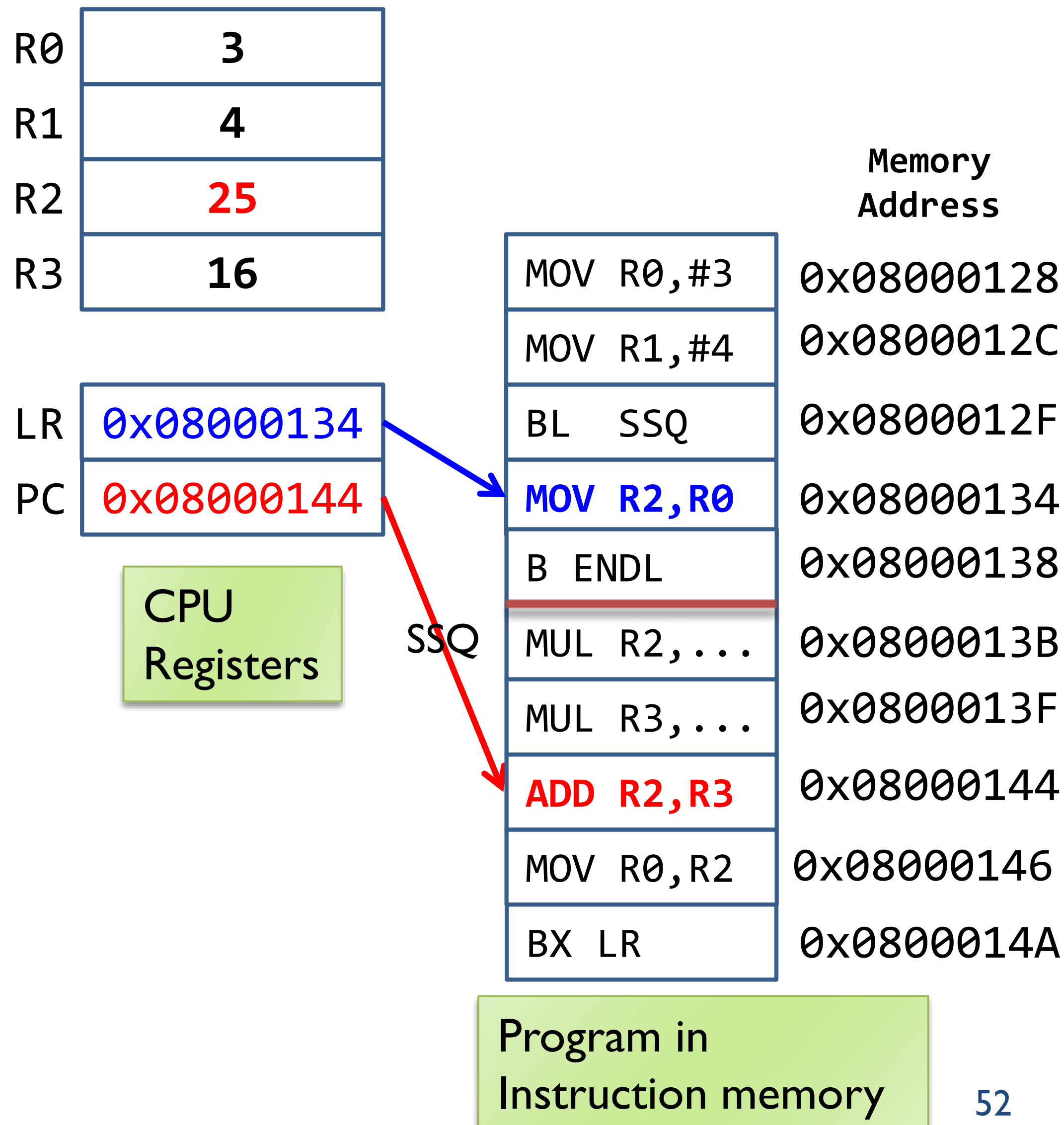
	Memory Address
MOV R0,#3	0x08000128
MOV R1,#4	0x0800012C
BL SSQ	0x0800012F
MOV R2,R0	0x08000134
B ENDL	0x08000138
MUL R2,...	0x0800013B
MUL R3,...	0x0800013F
ADD R2,R3	0x08000144
MOV R0,R2	0x08000146
BX LR	0x0800014A

SSQ

Program in Instruction memory

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    ...
```

R0	25
R1	4
R2	25
R3	16
LR	0x08000134
PC	0x08000146

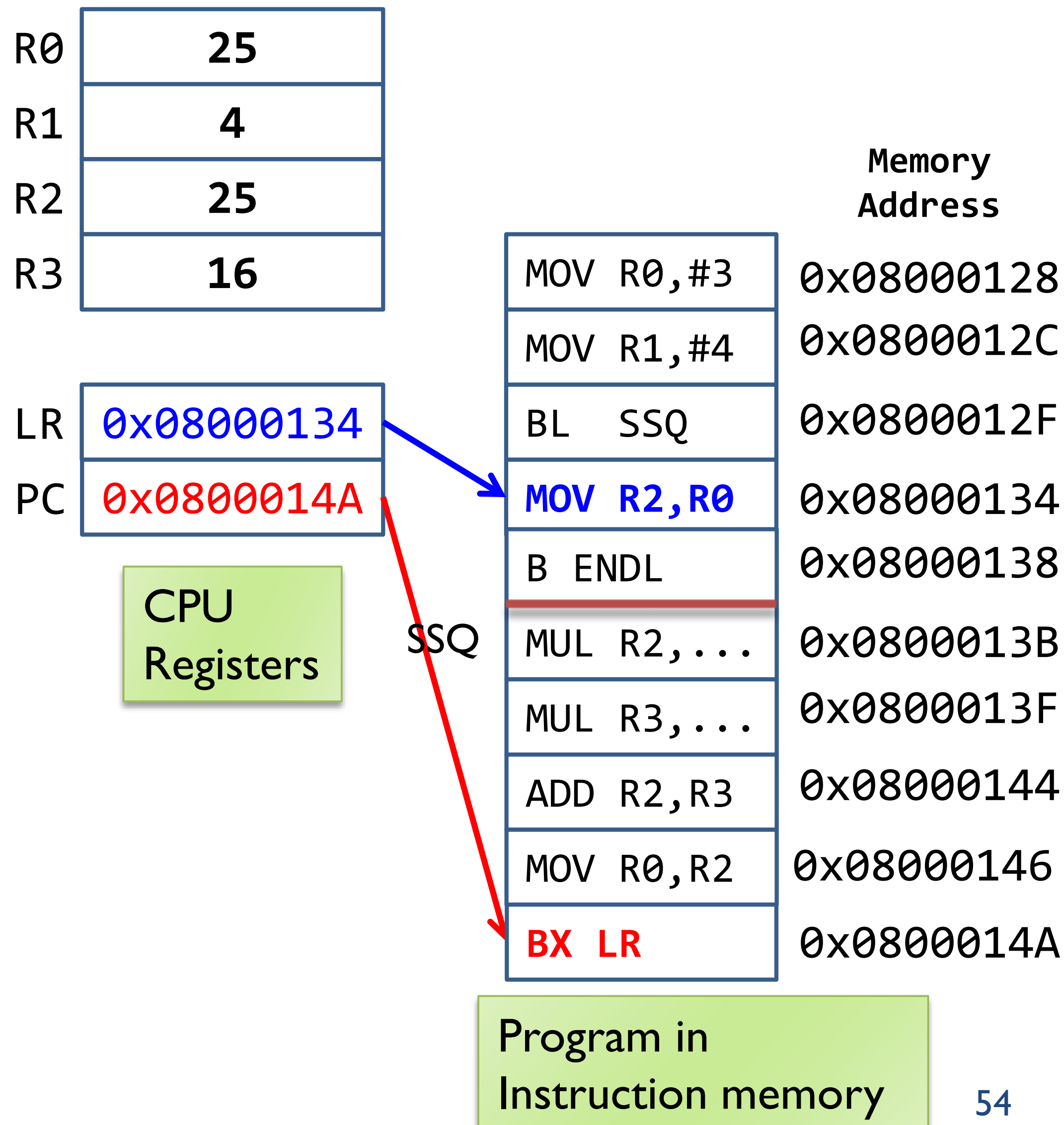
CPU Registers

	Instruction	Memory Address
	MOV R0,#3	0x08000128
	MOV R1,#4	0x0800012C
	BL SSQ	0x0800012F
	MOV R2,R0	0x08000134
	B ENDL	0x08000138
SSQ	MUL R2,...	0x0800013B
	MUL R3,...	0x0800013F
	ADD R2,R3	0x08000144
	MOV R0,R2	0x08000146
	BX LR	0x0800014A

Program in Instruction memory

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 function!

R0	25
R1	4
R2	25
R3	16

LR	0x08000134
PC	0x08000134

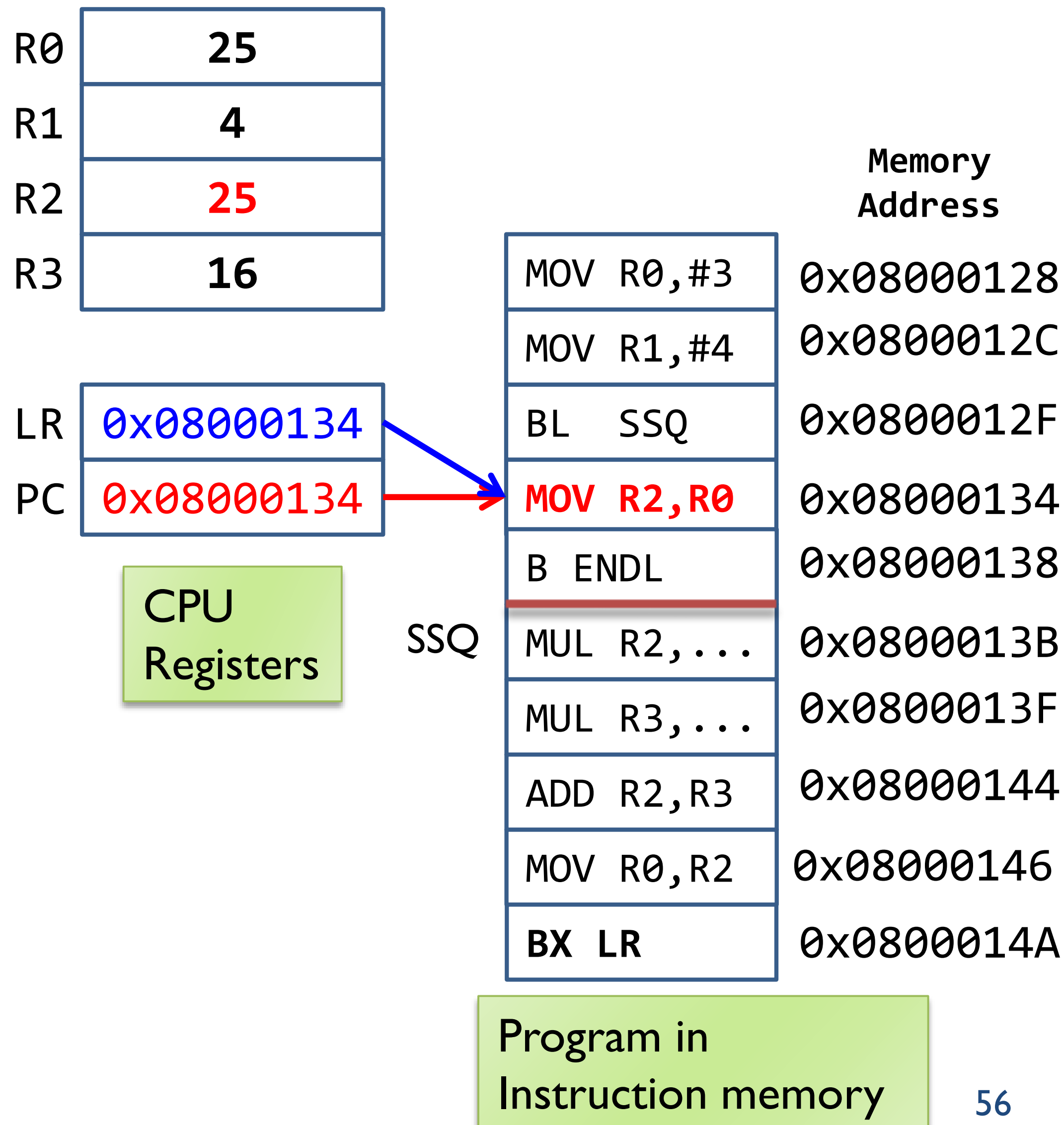
CPU Registers

	Instruction	Memory Address
	MOV R0,#3	0x08000128
	MOV R1,#4	0x0800012C
	BL SSQ	0x0800012F
	MOV R2,R0	0x08000134
	B ENDL	0x08000138
SSQ	MUL R2,...	0x0800013B
	MUL R3,...	0x0800013F
	ADD R2,R3	0x08000144
	MOV R0,R2	0x08000146
	BX LR	0x0800014A

Program in Instruction memory

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    ...
```

R0	25
R1	4
R2	25
R3	16

LR	0x08000134
PC	0x08000138

CPU Registers

	Instruction	Memory Address
	MOV R0,#3	0x08000128
	MOV R1,#4	0x0800012C
	BL SSQ	0x0800012F
	MOV R2,R0	0x08000134
	B ENDL	0x08000138
SSQ	MUL R2,...	0x0800013B
	MUL R3,...	0x0800013F
	ADD R2,R3	0x08000144
	MOV R0,R2	0x08000146
	BX LR	0x0800014A

Program in Instruction memory

Summary

- Stack
  - PUSH and POP operations
- Calling a function
  - Either BX, or PUSH(LR)+POP(PC) in callee
- Calling conventions
  - R0-R3: scratch registers that can be changed by callee
  - R4-R11: caller expects their value to stay unchanged before and after a function call
- Passing arguments and returning a value
  - Arguments passed in R0-R3, plus stack if necessary
  - Value returned in R0-R3, depending on its size