Chapter 8
Passing Parameters to Subroutines via Registers

Z. Gu

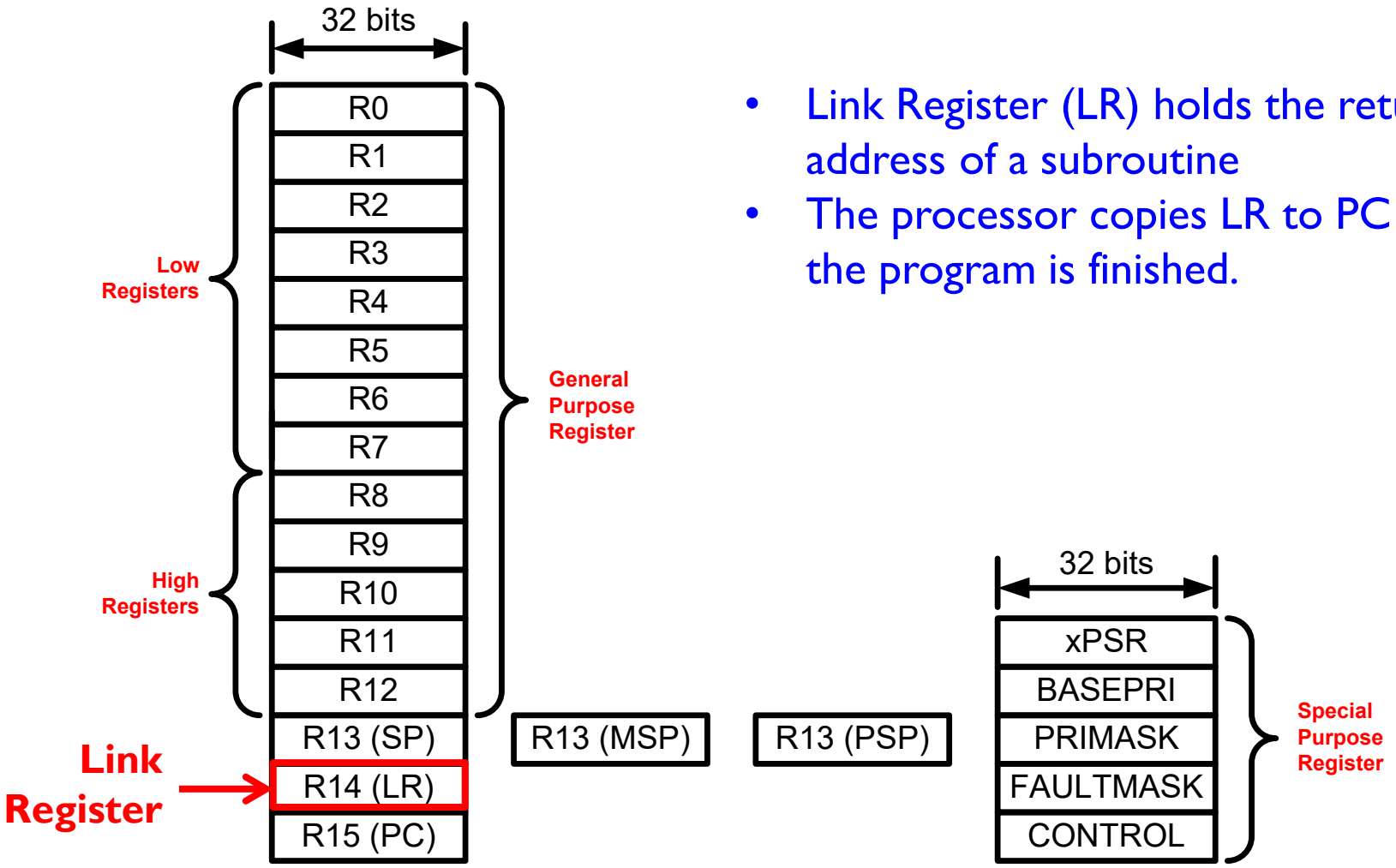
Fall 2025

Acknowledgement: Lecture slides based on Embedded Systems with ARM Cortex-M Microcontrollers in Assembly Language and C, University of Maine https://web.eece.maine.edu/~zhu/book/

Overview

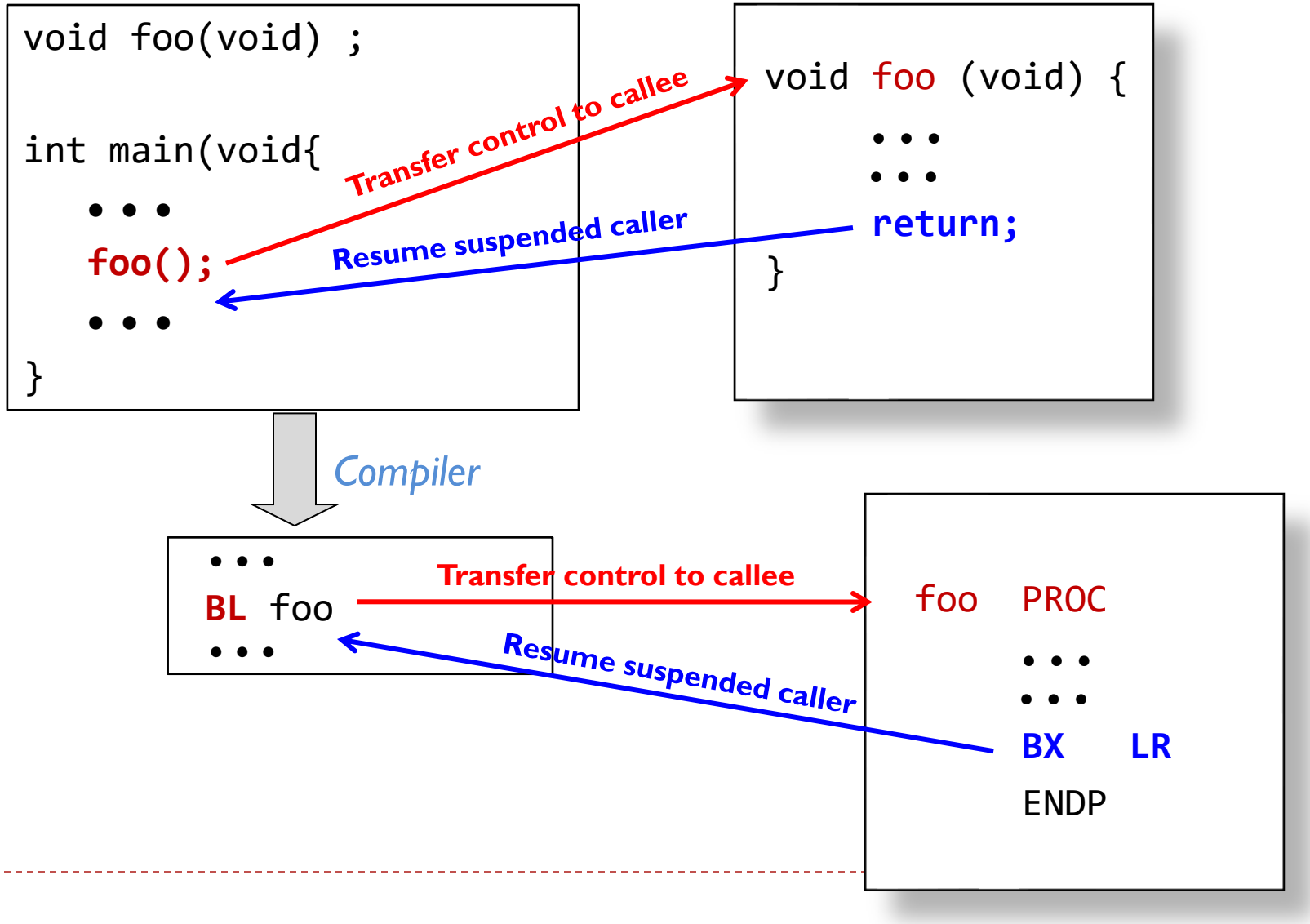
- How to call a subroutine?
- How to return the control back to the caller?
- How to pass arguments into a subroutine?
- How to return a value in a subroutine?
- How to preserve the running environment for the caller?

Link Register (LR)

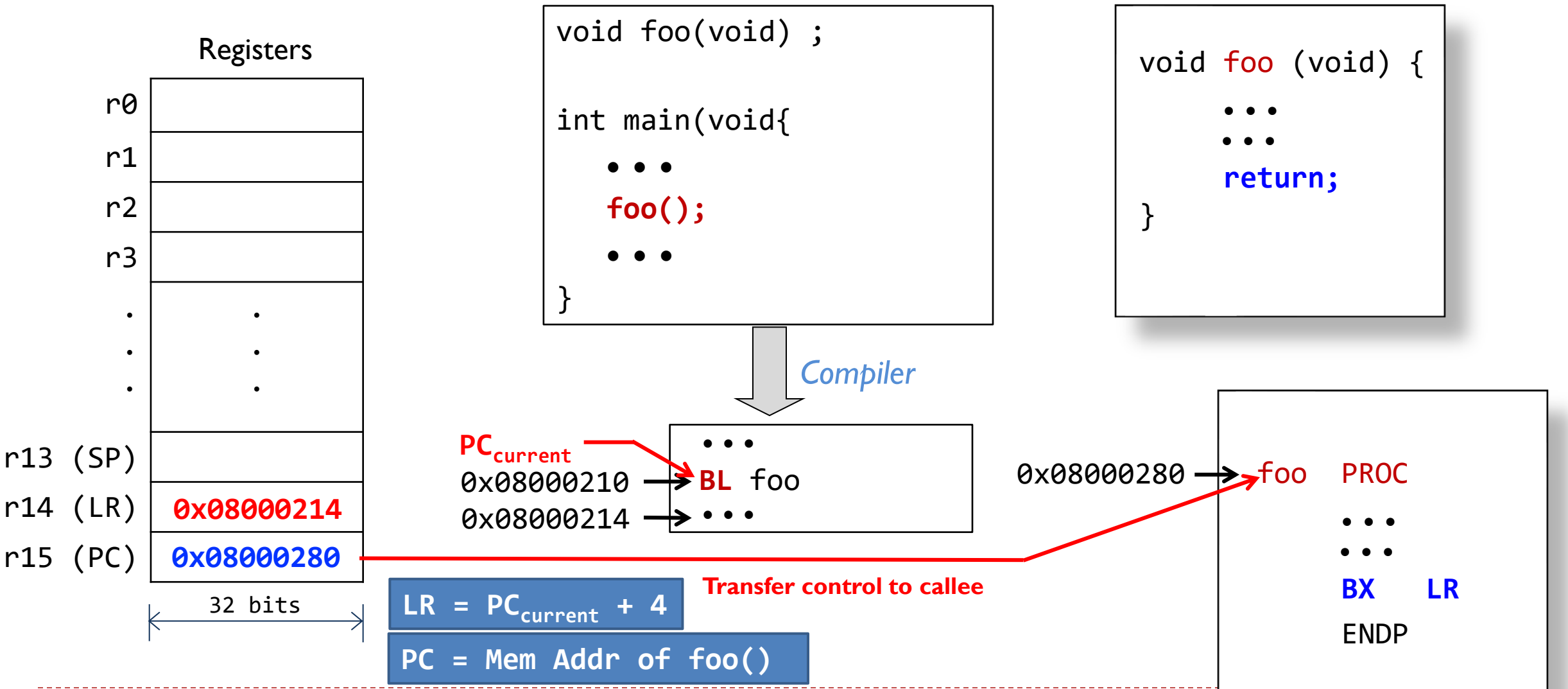


- Link Register (LR) holds the return address of a subroutine
- The processor copies LR to PC after the program is finished.

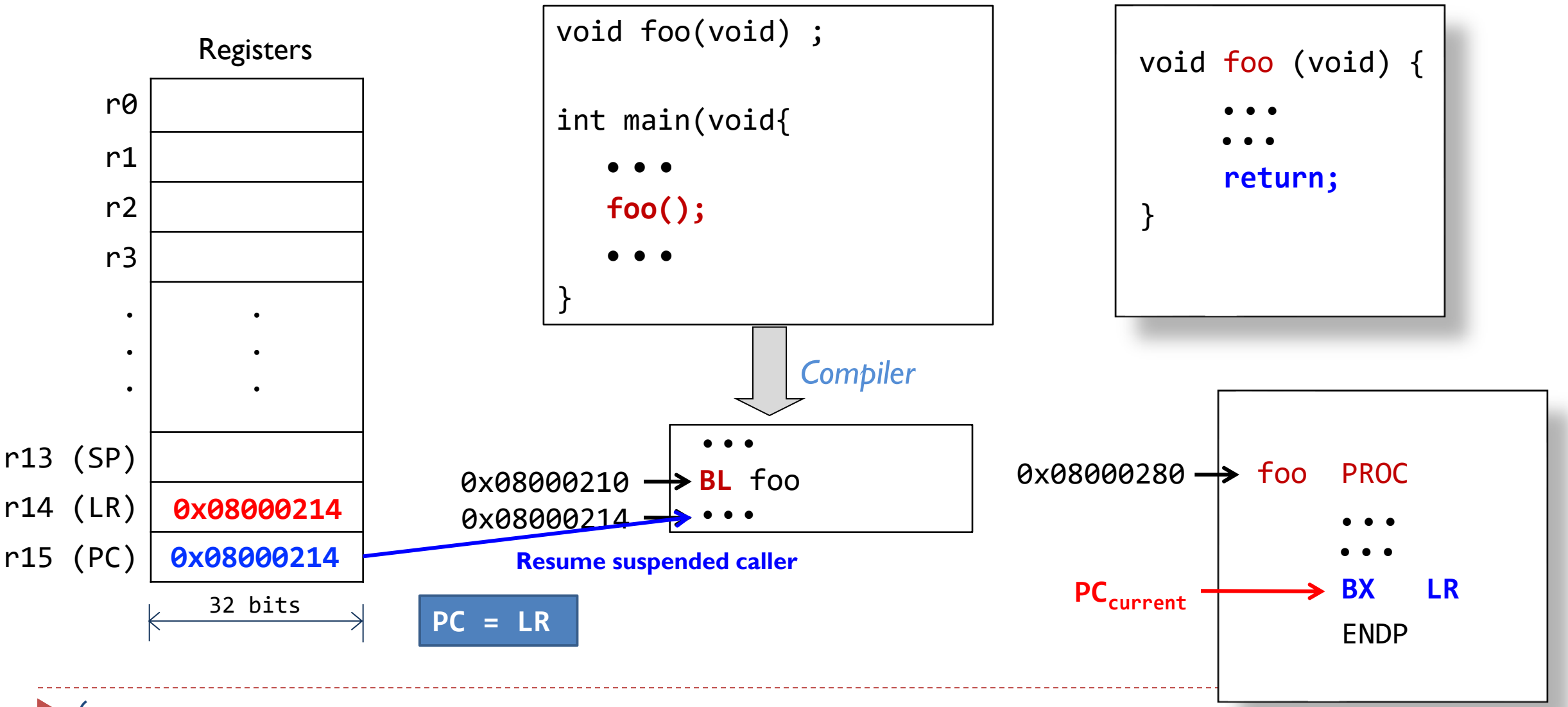
Link Register (LR)



Link Register (LR)



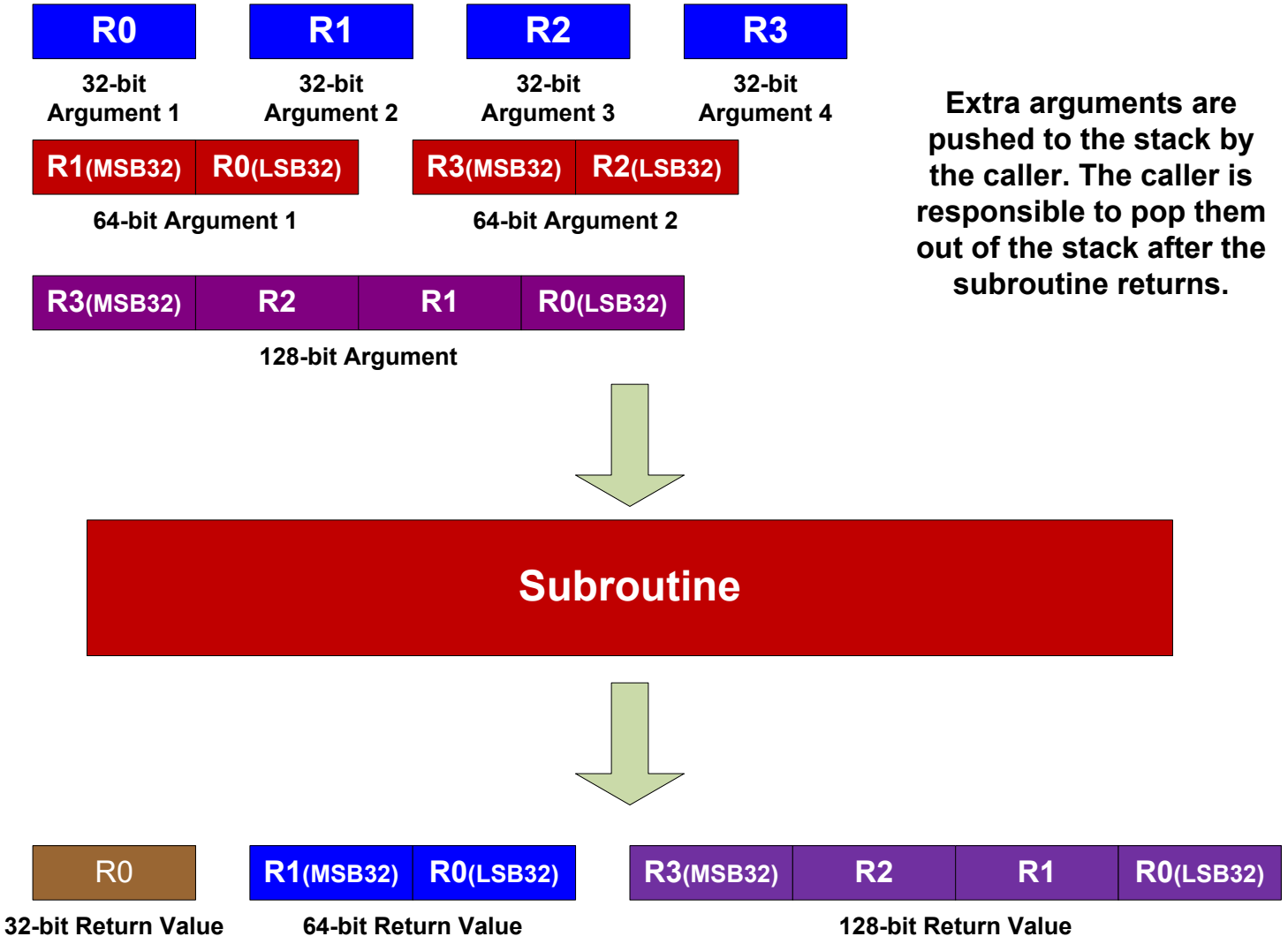
Link Register (LR)



Procedure Call Standard

- What is it?
  - Contract between a calling subroutine (caller) and a called subroutine (callee)
- Why need it?
  - Allows subroutines to be separately written, compiled, assembled but work together
  - Allows C program calls an assembly function, or vice versa
- This talk focuses on
  - How to pass arguments to a subroutine?
  - How to return a result from a subroutine?

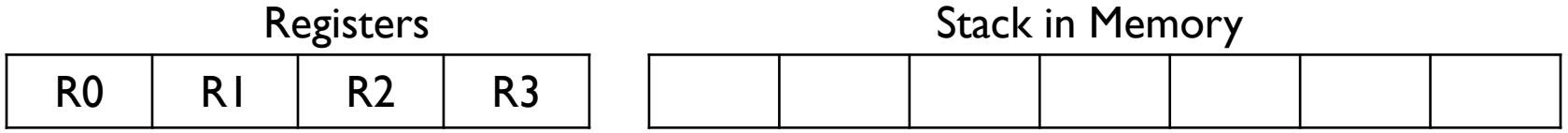
Passing Arguments and Returning Value



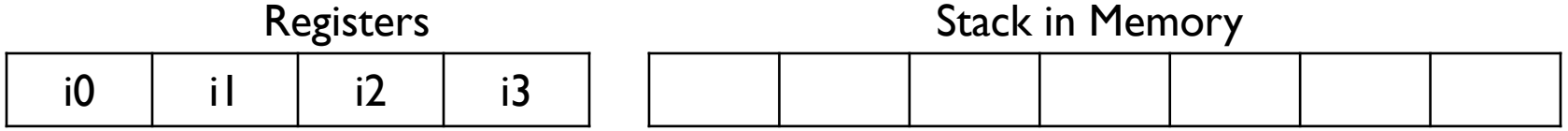
Extra arguments are pushed to the stack by the caller. The caller is responsible to pop them out of the stack after the subroutine returns.

- Each argument with size ≤ 32 bits, e.g., 8-bit char, or 16-bit short, or 32-bit int, is passed in a 32-bit register.
  - Cannot pack multiple arguments into one register.
- The subroutine can take arguments larger than 32 bits. For example, a double-word variable, such as 64-bit long, is passed in two consecutive registers (e.g. R0 and R1, or R2 and R3). A 128-bit variable is passed in four consecutive registers.
  - *int64_t add_64(int64_t a, int64_t b)*
  - R0 and R1 are used to store the variable *a*
- The return result is stored in registers (R0-R3), depending on the size of the return variable. If it is less than 32 bits, it is stored in R0. If it is a double-word sized variable, such as *long long* or *double* variables in C, it is stored in R0 and R1.
  - *int128_t multiply_64(int64_t a, int64_t b)*
  - R0, R1, R2, and R3 are used to store the result

Passing Arguments Examples

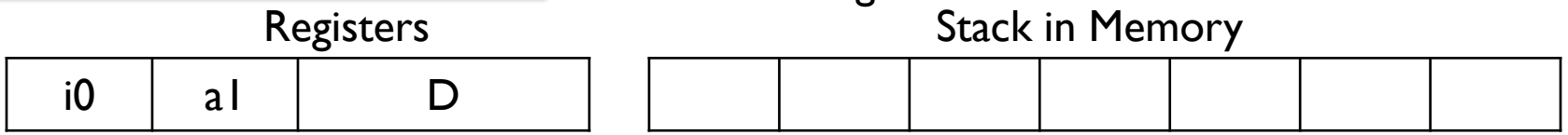


foo (int i0, int i1, int i2, int i3)

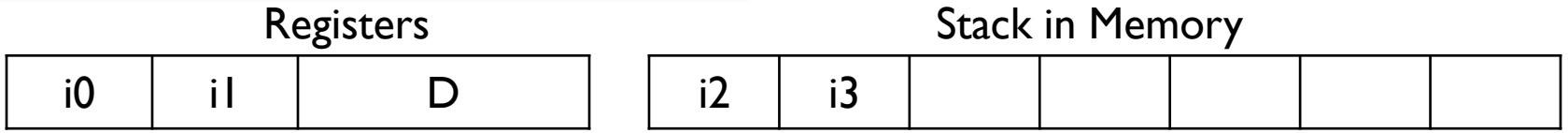


foo (int i0, char a1, double D)

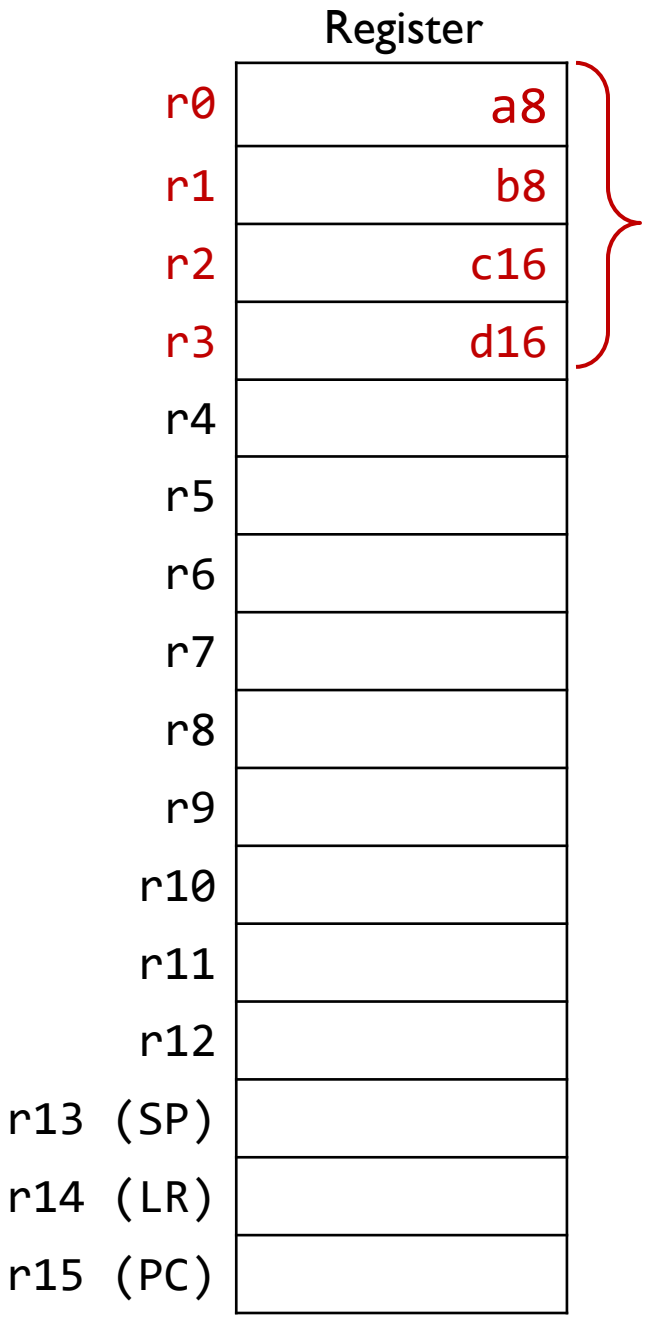
Each argument of 8-bit char, or 16-bit short, is passed in a 32-bit register



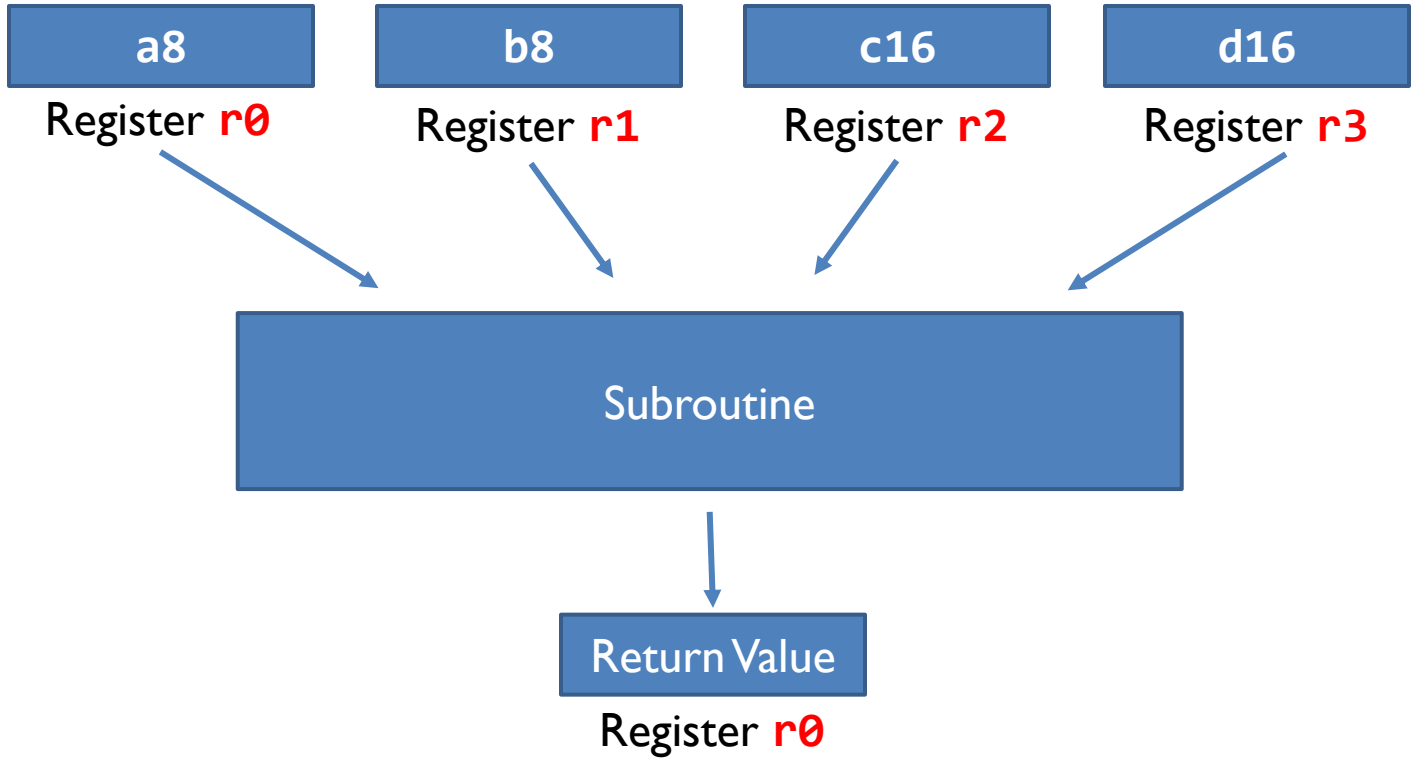
foo (int i0, int i1, double D, int i2, int i3)

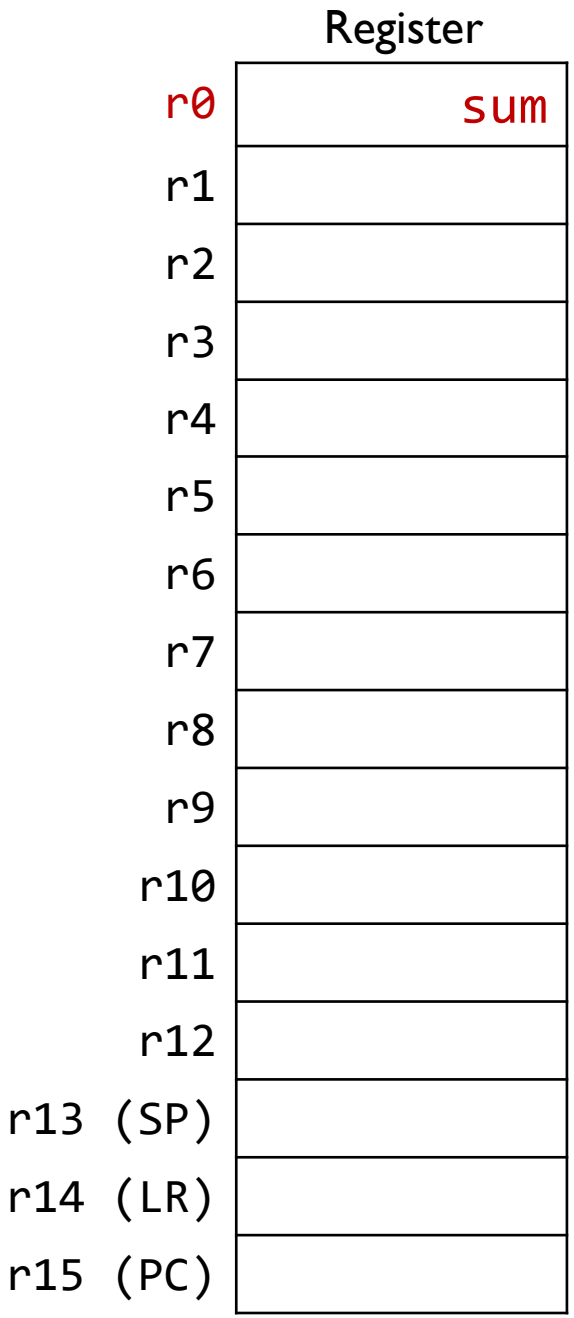


Caller passes arguments i0, i1, D in registers R0-R3 directly; pushes additional arguments i2 and i3 on the stack before subroutine call



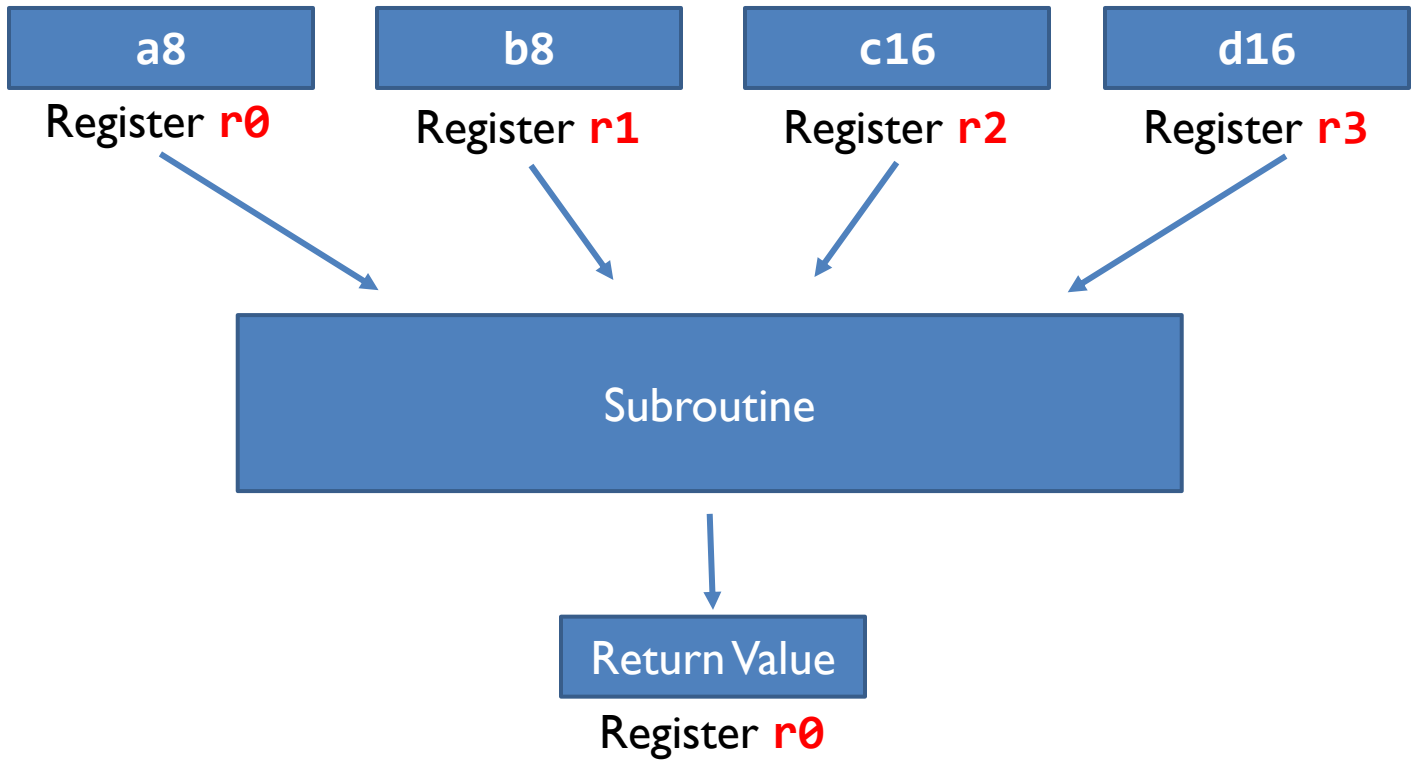

```
int32_t sum(uint8_t a8, int8_t b8, uint16_t c16, uint16_t d16);
```





Return the sum in register r0

```
int32_t sum(uint8_t a8, int8_t b8, uint16_t c16, uint16_t d16);
```



Passing Arguments and Returning Value

```
int32_t sum(uint8_t a8, int8_t b8, uint16_t c16, uint16_t d16);

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

Caller

```
MOVS r0, #1   ; a8
MOVS r1, #2   ; b8
MOVS r2, #3   ; c16
MOVS r3, #4   ; d16
BL   sum
```

Callee

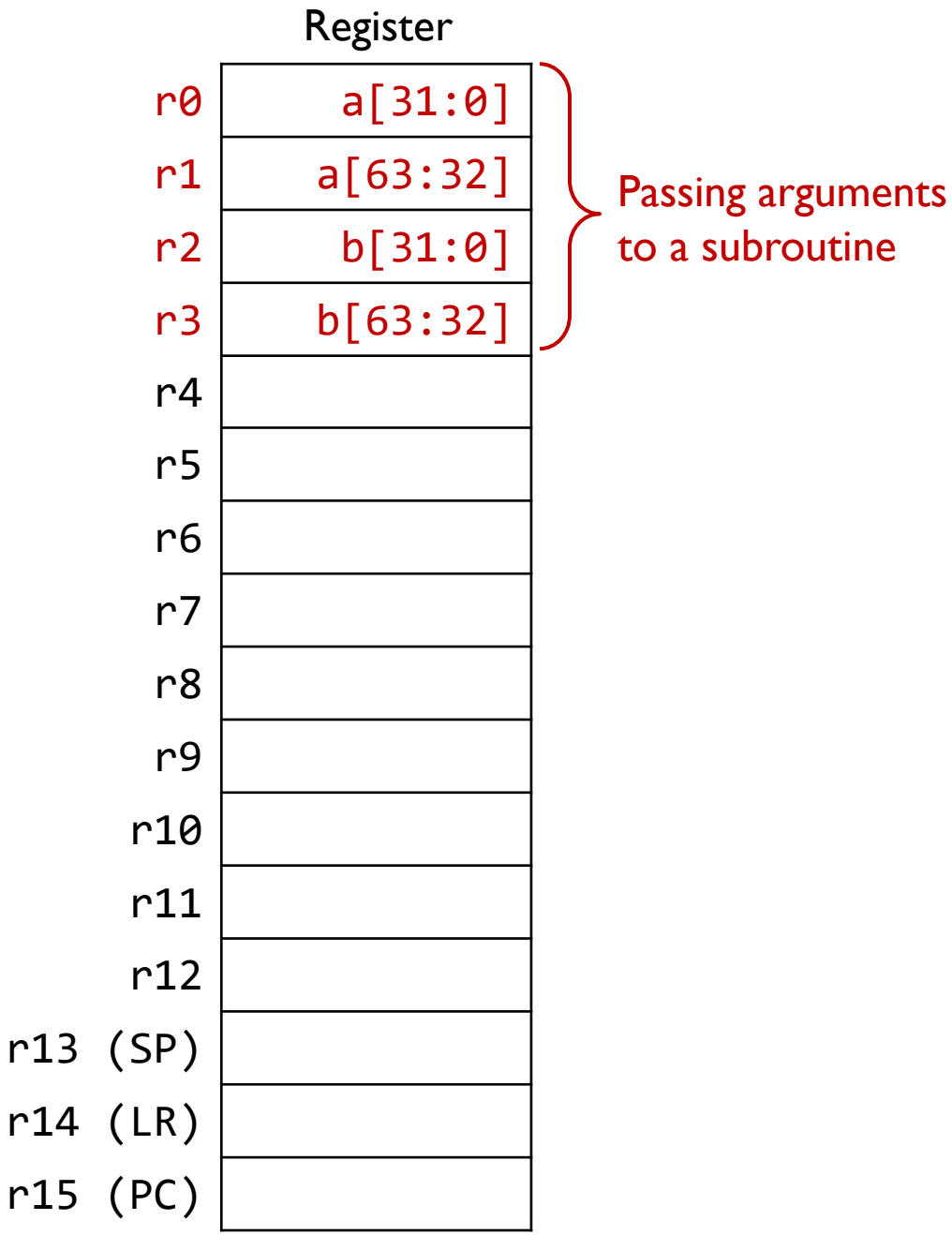
```
sum PROC
  ADD r0, r0, r1   ; a8 + b8
  ADD r0, r0, r2   ; add c16
  ADD r0, r0, r3   ; add d16
  BX  LR
  ENDP
```

Returning Value

```
uint32_t s32;

uint32_t sum(uint8_t a8, uint8_t b8, uint16_t c16, uint16_t d16);

s32 = sum(1, 2, 3, 4) + 100;
```

```
MOVS r0, #1   ; 1st argument a8
MOVS r1, #2   ; 2nd argument b8
MOVS r2, #3   ; 3rd argument c16
MOVS r3, #4   ; 4th argument d16
BL   sum      ; result is returned in r0
ADD r0, r0, #100
LDR r4, =s32  ; Get memory address of s32
STR r0, [r4]  ; Save returned result to s32
```

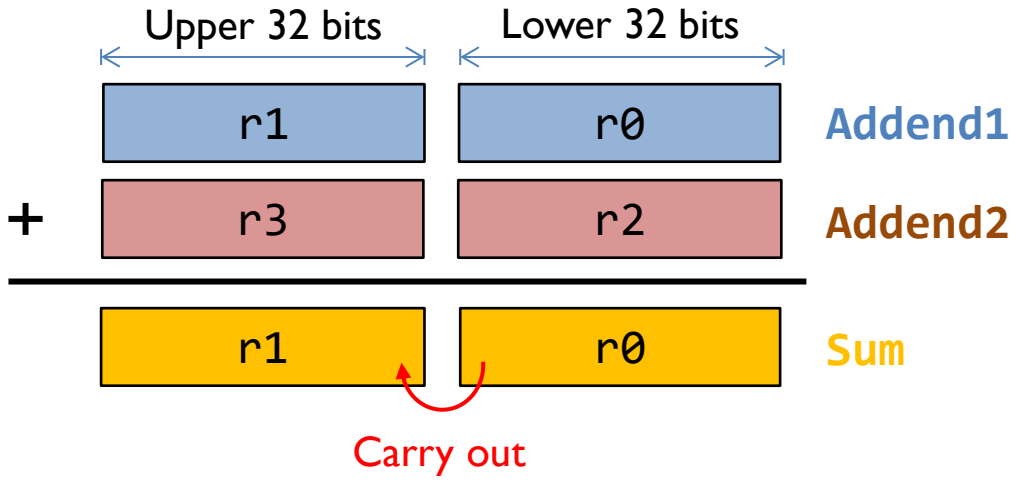
uint32_t s32 is declared as a C global variable, so the compiler/linker creates storage for it in .data or .bss depending on initialization. You can simply reference it by name in assembly as a label.

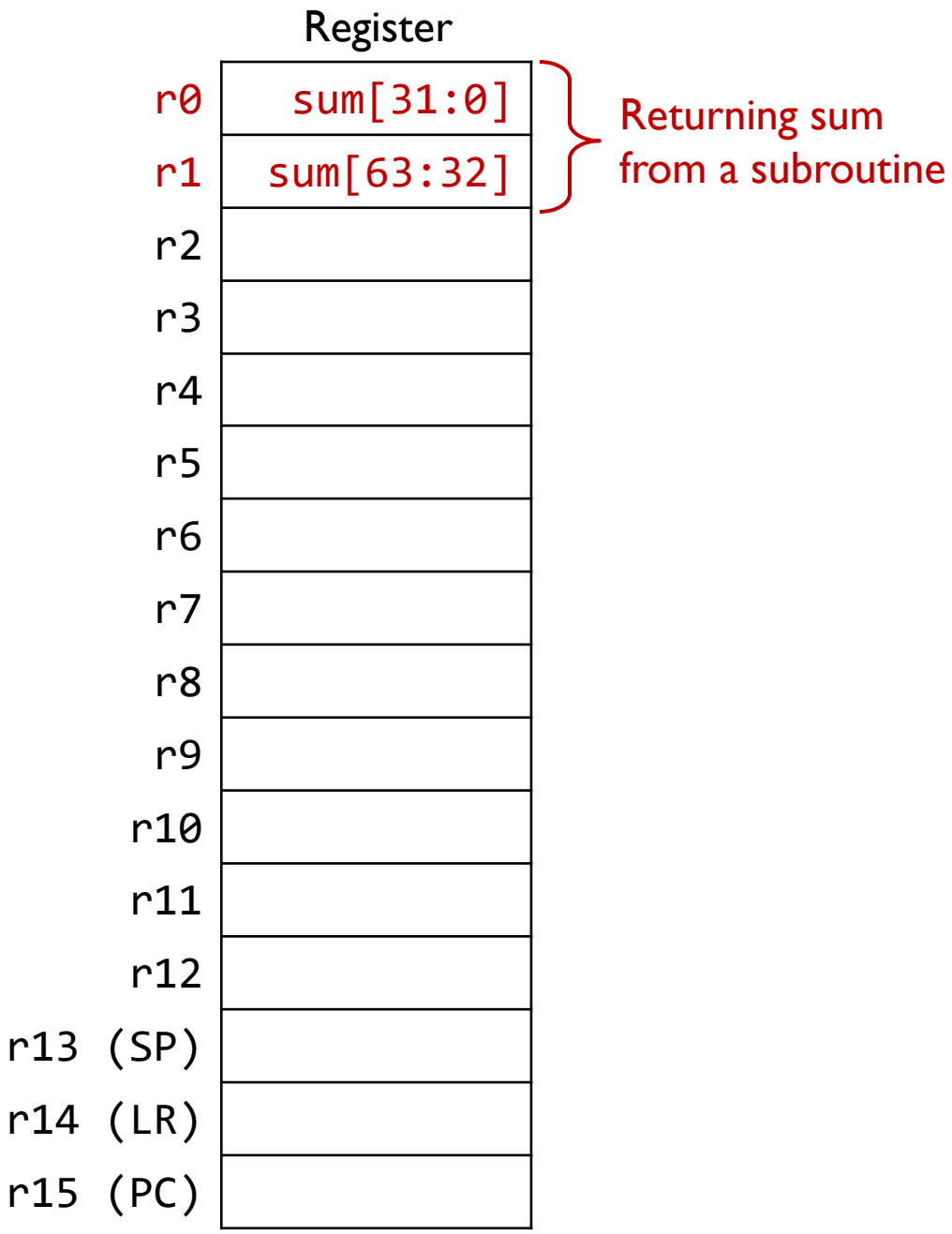



```
int64_t sum(int64_t a, int64_t b);
```

Callee

```
sum PROC
    ADDS r0, r2, r0  ; Adding lower 32 bits
    ADC  r1, r3, r1  ; Adding upper 32 bits
    BX   LR          ; Return in r1:r0
    ENDP
```

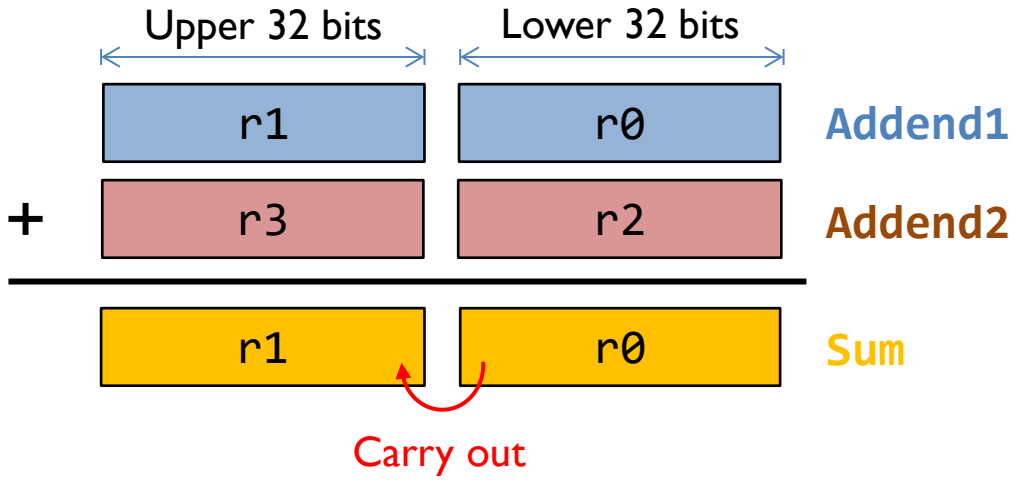


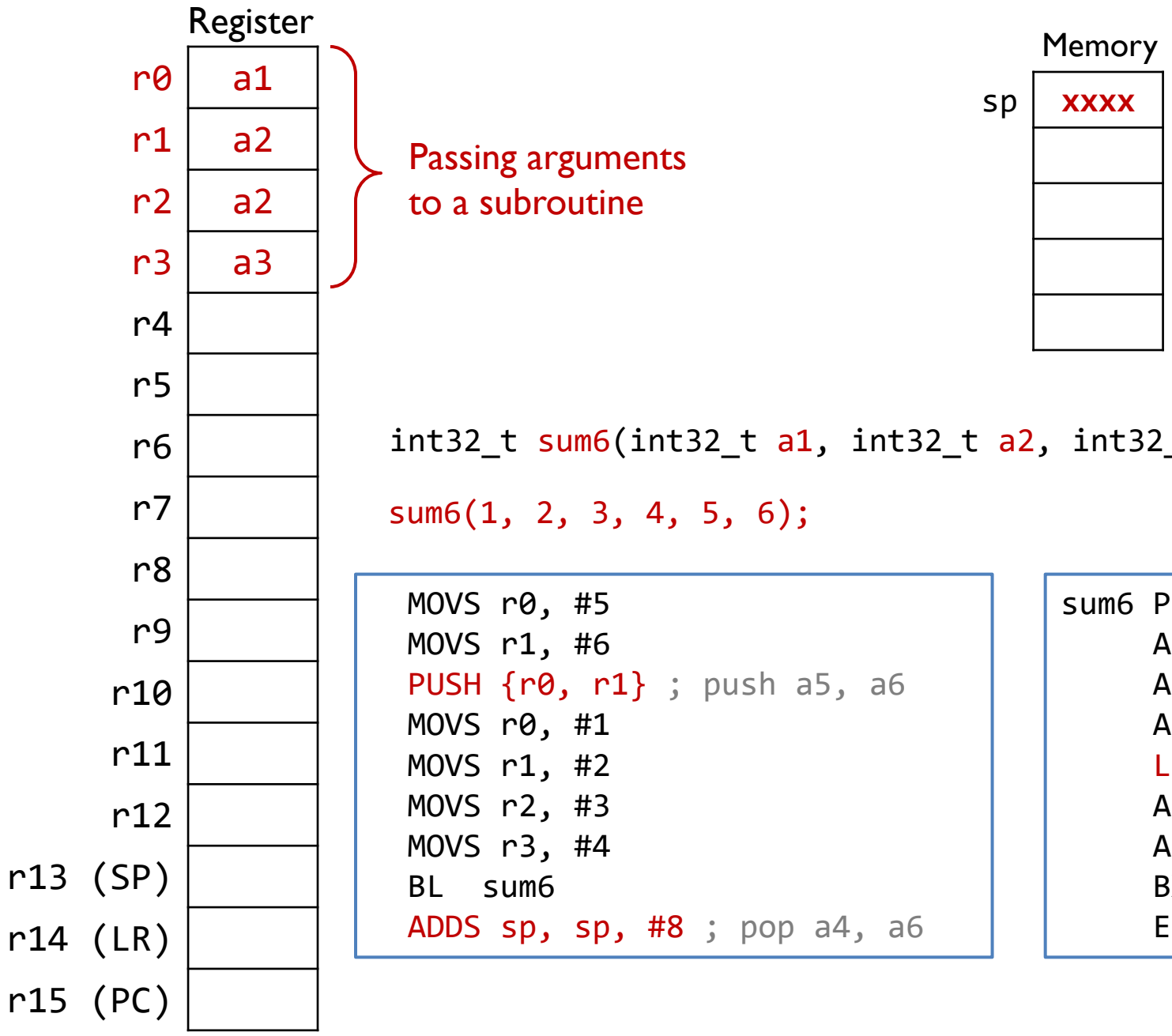



```
int64_t sum(int64_t a, int64_t b);
```

Callee

```
sum PROC
    ADDS r0, r2, r0  ; Adding lower 32 bits
    ADC  r1, r3, r1  ; Adding upper 32 bits
    BX   LR          ; Return in r1:r0
    ENDP
```

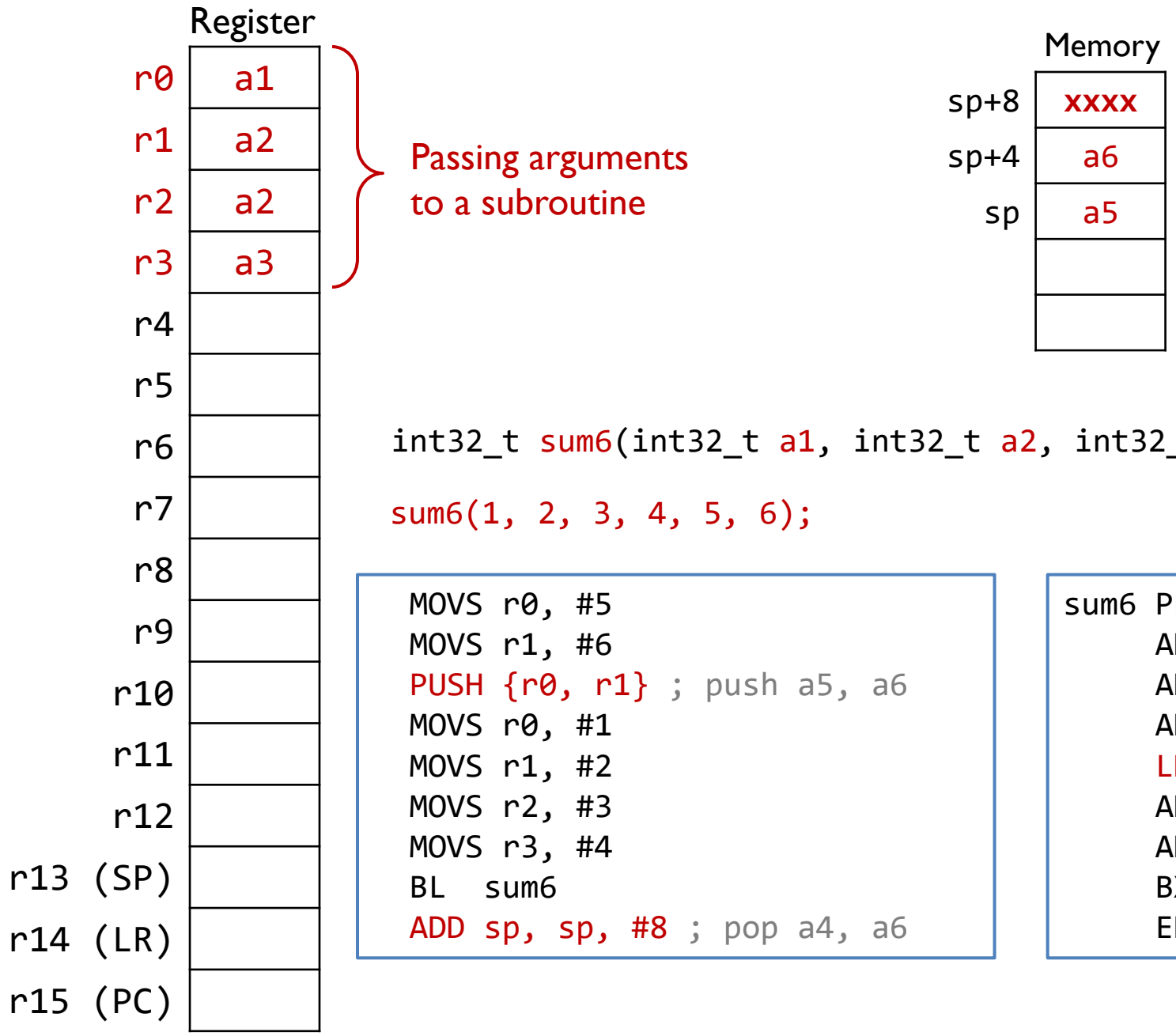





```
int32_t sum6(int32_t a1, int32_t a2, int32_t a3, int32_t a4, int32_t a5, int32_t a6)

sum6(1, 2, 3, 4, 5, 6);
```

```
MOVS r0, #5
MOVS r1, #6
PUSH {r0, r1} ; push a5, a6
MOVS r0, #1
MOVS r1, #2
MOVS r2, #3
MOVS r3, #4
BL   sum6
ADDS sp, sp, #8 ; pop a4, a6
```

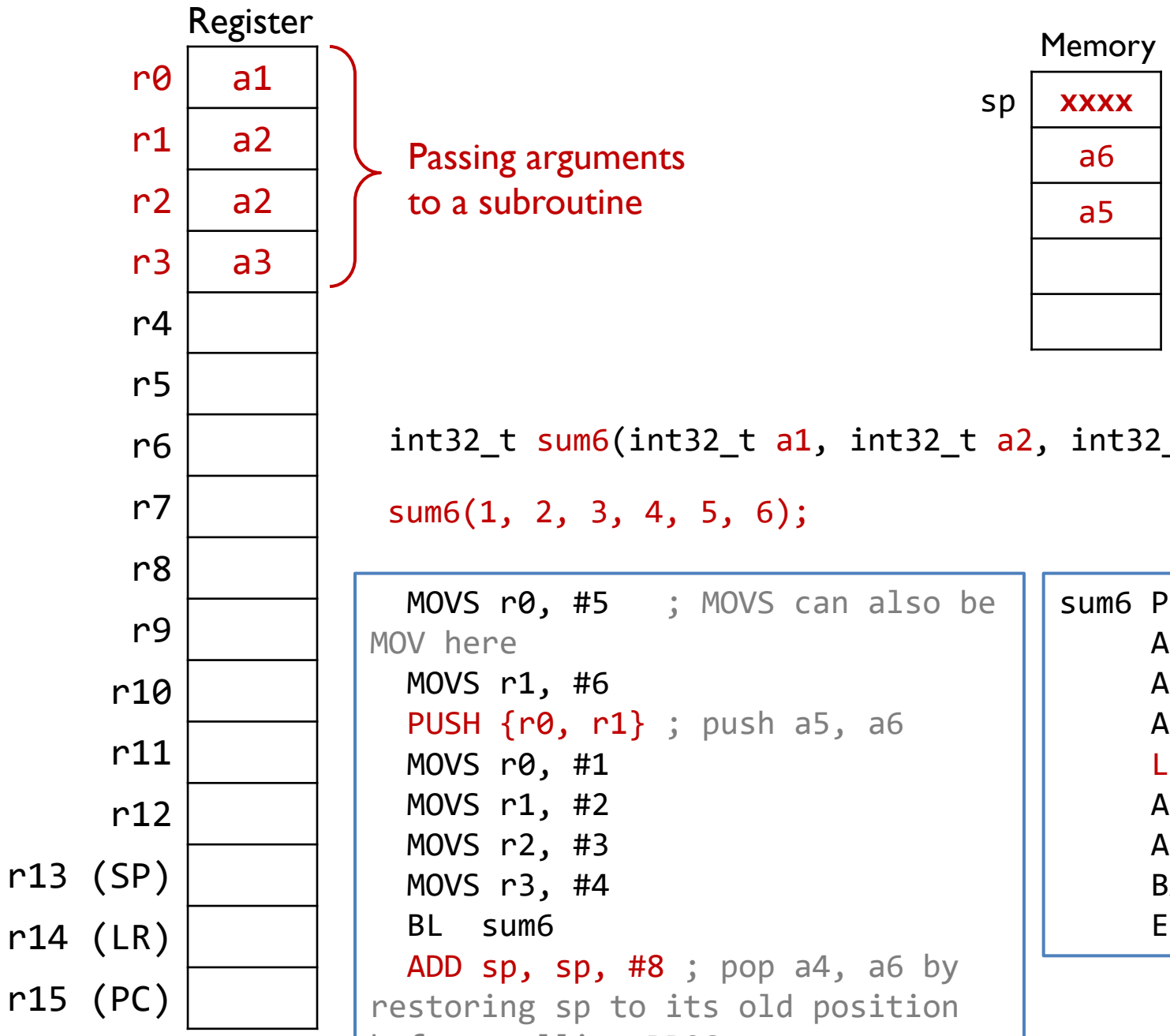
```
sum6 PROC
     ADD  r0, r0, r1    ; sum = a1 + a2
     ADD  r0, r0, r2    ; sum += a3
     ADD  r0, r0, r3    ; sum += a4
     LDRD r1, r2, [sp] ; load a5, a6
     ADD  r0, r0, r1    ; sum += a5
     ADD  r0, r0, r2    ; sum += a6
     BX   LR            ; return in r0
     ENDP
```




```
int32_t sum6(int32_t a1, int32_t a2, int32_t a3, int32_t a4, int32_t a5, int32_t a6)

sum6(1, 2, 3, 4, 5, 6);
```

```
MOVS r0, #5
MOVS r1, #6
PUSH {r0, r1} ; push a5, a6
MOVS r0, #1
MOVS r1, #2
MOVS r2, #3
MOVS r3, #4
BL   sum6
ADD sp, sp, #8 ; pop a4, a6
```

```
sum6 PROC
     ADD   r0, r0, r1    ; sum = a1 + a2
     ADD   r0, r0, r2    ; sum += a3
     ADD   r0, r0, r3    ; sum += a4
     LDRD r1, r2, [sp] ; load a5, a6
     ADD   r0, r0, r1    ; sum += a5
     ADD   r0, r0, r2    ; sum += a6
     BX    LR            ; return in r0
     ENDP
```




```
int32_t sum6(int32_t a1, int32_t a2, int32_t a3, int32_t a4, int32_t a5, int32_t a6)

sum6(1, 2, 3, 4, 5, 6);
```

```
  MOVS r0, #5    ; MOVS can also be
MOV here
  MOVS r1, #6
  PUSH {r0, r1} ; push a5, a6
  MOVS r0, #1
  MOVS r1, #2
  MOVS r2, #3
  MOVS r3, #4
  BL   sum6
  ADD sp, sp, #8 ; pop a4, a6 by
restoring sp to its old position
before calling PROC
```

```
sum6 PROC
     ADD  r0, r0, r1    ; sum = a1 + a2
     ADD  r0, r0, r2    ; sum += a3
     ADD  r0, r0, r3    ; sum += a4
     LDRD r1, r2, [sp] ; load a5, a6
     ADD  r0, r0, r1    ; sum += a5
     ADD  r0, r0, r2    ; sum += a6
     BX   LR            ; return in r0
     ENDP
```

Explanations

Step	r0	r1	r2	r3	Stack (top first)	Description
Before call	5	6	—	—	—	a5, a6 prepared
After PUSH	—	—	—	—	a5=5, a6=6	pushed to stack
Set r0–r3	1	2	3	4	a5=5, a6=6	a1–a4 in regs
Inside sum6	1	2	3	4	a5=5, a6=6	entry point
LDRD loads	—	5	6	—	a5=5, a6=6	from stack
Return	21	—	—	—	a5=5, a6=6	1+2+3+4+5+6 = 21

- First 4 parameters → registers r0–r3.Extra parameters → pushed onto the stack.
- LDRD (Load Register Double) fetches two words (a5, a6) efficiently.
- r0 always holds the return value.
- Caller cleans up the stack after the function (ADDS sp, sp, #8).

Quiz

- Can I use POP to replace ADD?

```
MOVS r0, #5
MOVS r1, #6
PUSH {r0, r1} ; push a5, a6
MOVS r0, #1
MOVS r1, #2
MOVS r2, #3
MOVS r3, #4
BL  sum6
POP {r0, r1}
; ADD sp, sp, #8
```

Quiz ANS

- POP {r0, r1} is equivalent to LDMIA sp!, {r0, r1}
  - Load [sp] into r0
  - Load [sp + 4] into r1
  - Then increment sp by 8
- This is wrong, because it would overwrite registers r0 and r1 with garbage (the old arguments you pushed), whereas r0 should contain the return value
  - After returning from sum6, you do not need those values (a5, a6) anymore. You just want to discard them, so ADD moves stack pointer up by 8 bytes and discards data.

```
MOVS r0, #5
MOVS r1, #6
PUSH {r0, r1} ; push a5, a6
MOVS r0, #1
MOVS r1, #2
MOVS r2, #3
MOVS r3, #4
BL  sum6
POP {r0, r1}
; ADD sp, sp, #8
```

Calling Assembly Subroutine in C

- If your assembly code follows the procedure call standard, a C code can call an assembly subroutine, and vice versa.

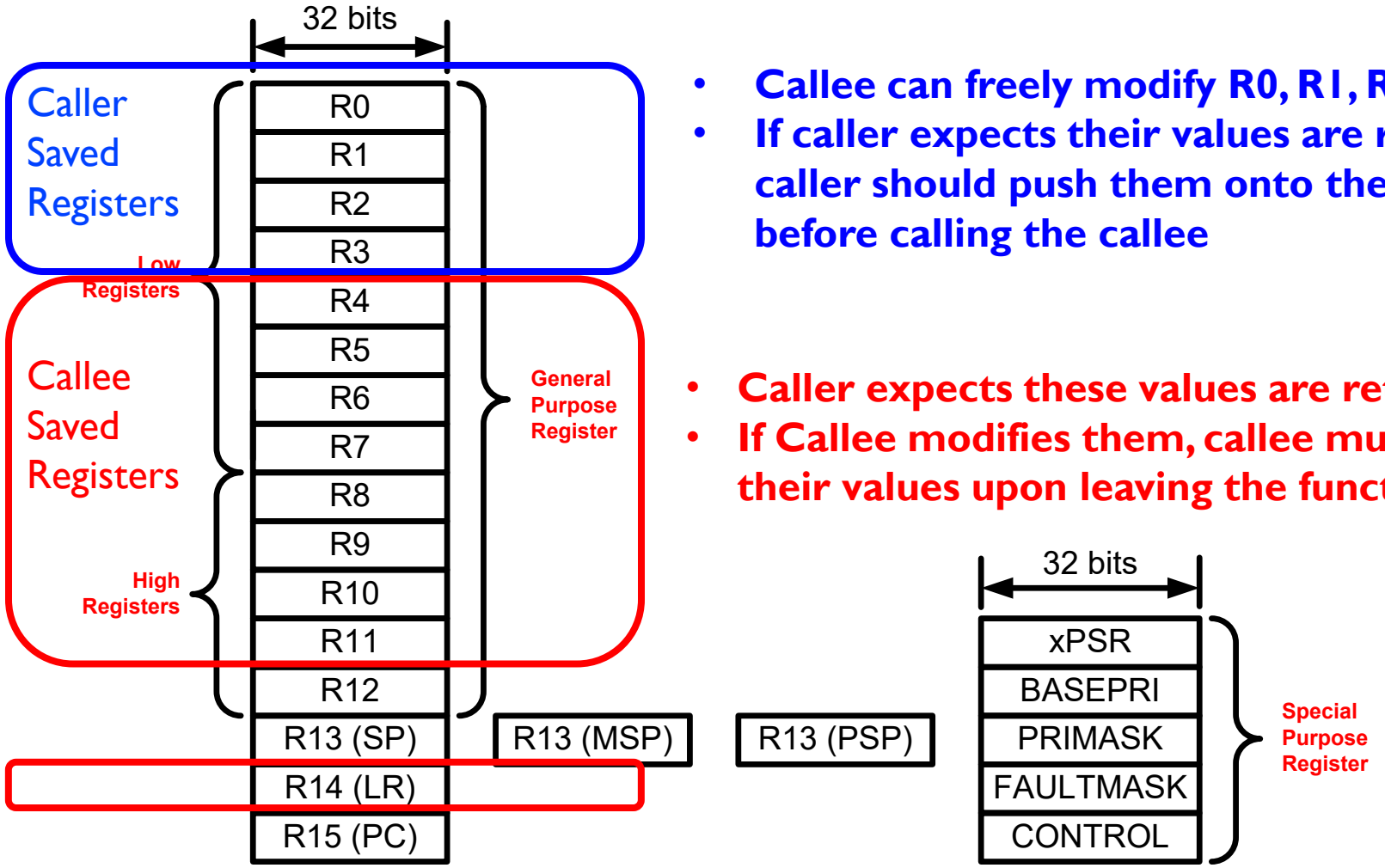
```
extern int32_t sum3(int32_t a1, int32_t a2, int32_t a3);


int main(void){
  int32_t s

  ...
  s = sum3(-1, -2, -3) + sum3(4, 5, 6);
  ...

}
```

```
sum3 PROC
     EXPORT sum3
     ADD   r0, r0, r1    ; sum = a1 + a2
     ADD   r0, r0, r2    ; sum += a3
     BX    LR            ; return in r0
     ENDP
```

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

Callee Saved Registers *vs* Caller Saved Registers



- **Callee can freely modify R0, R1, R2, and R3**
- **If caller expects their values are retained, caller should push them onto the stack before calling the callee**

- **Caller expects these values are retained .**
- **If Callee modifies them, callee must restore their values upon leaving the function.**

Example: SSQ(3, 4)

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

Sum of Square: $x^2 + y^2$

R1: second argument

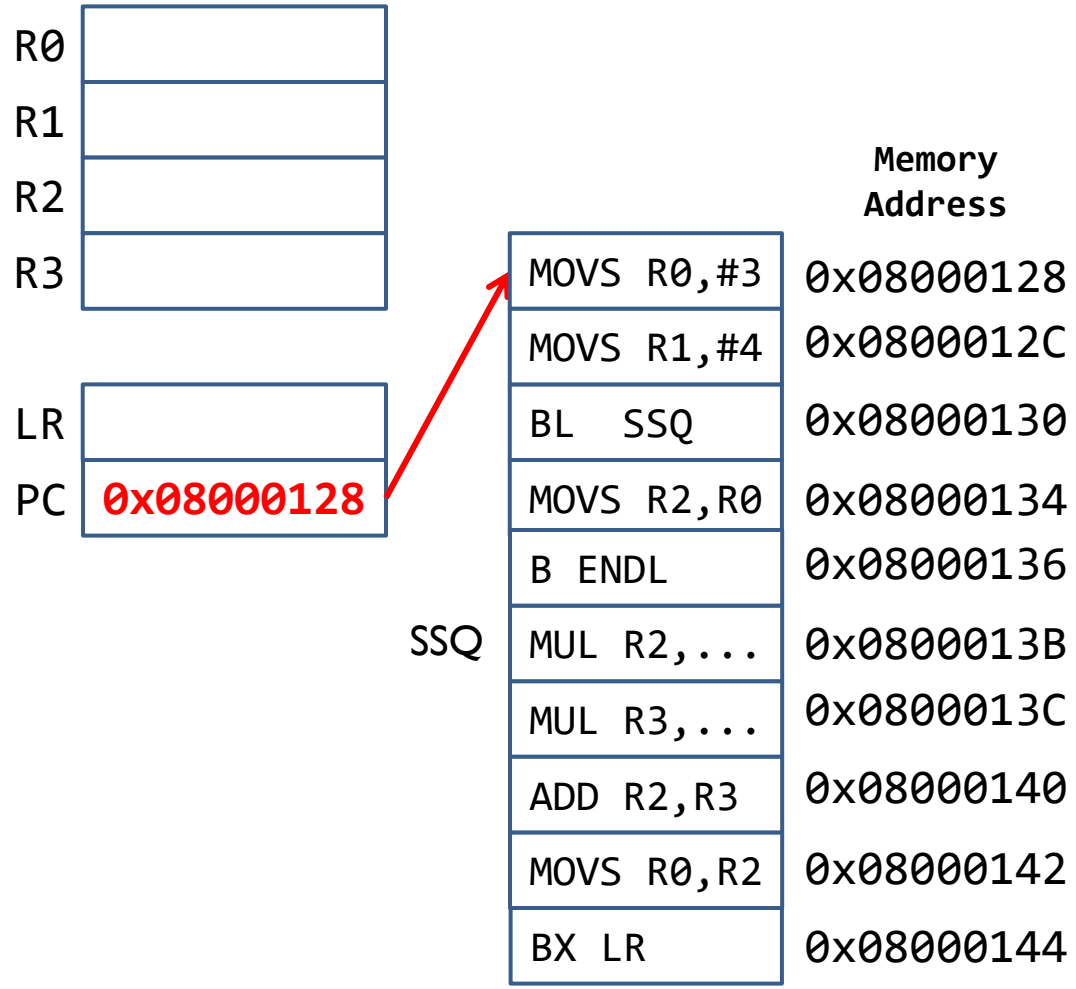
R0: first argument

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

R0: Return Value

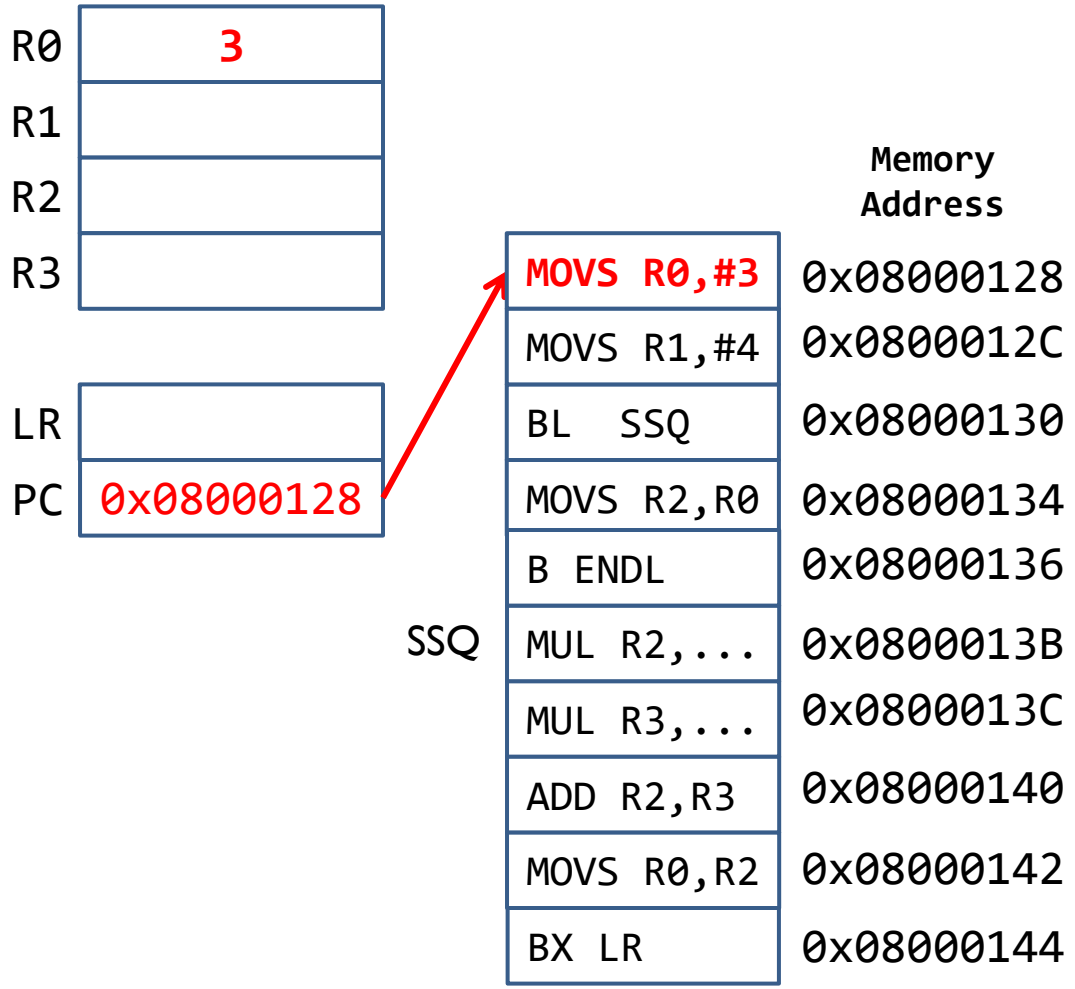
Example: SSQ(3, 4)

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



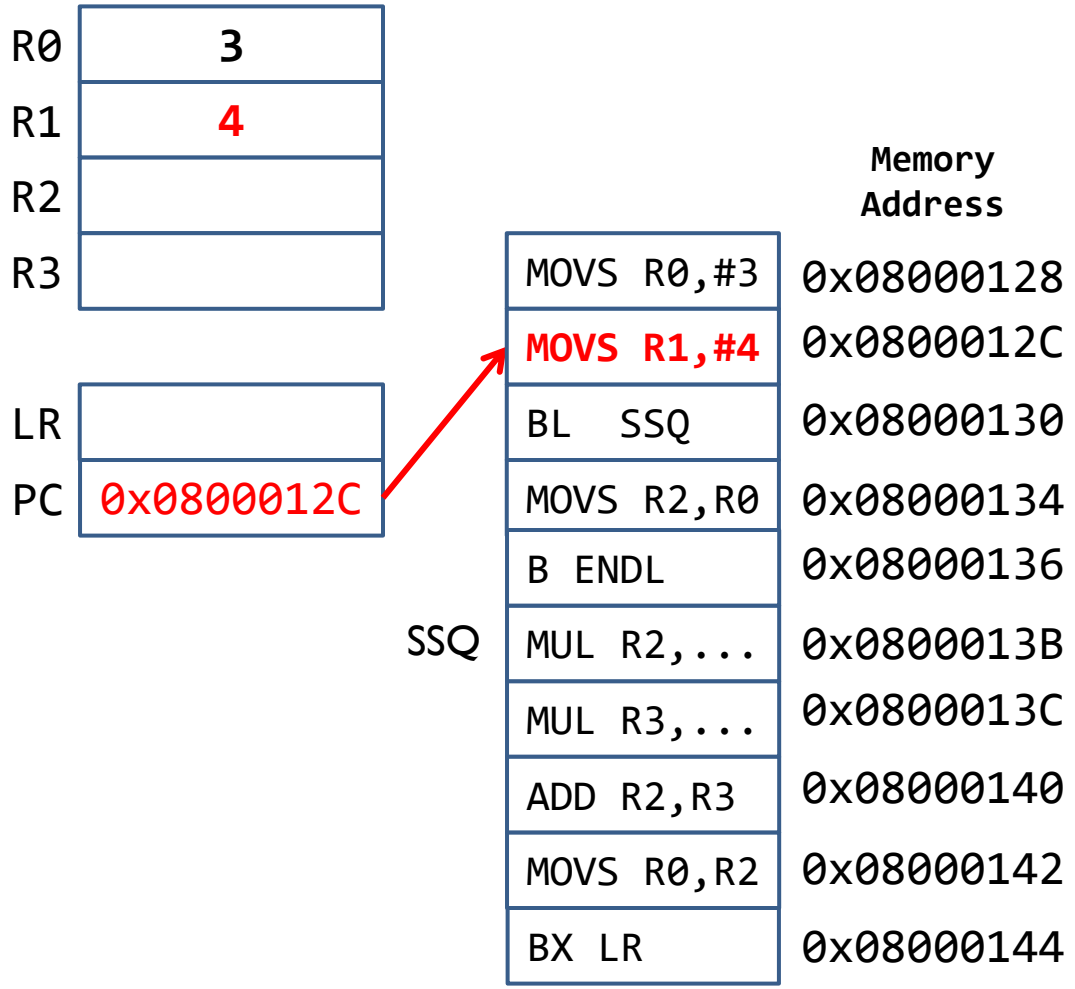
Example: SSQ(3, 4)

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



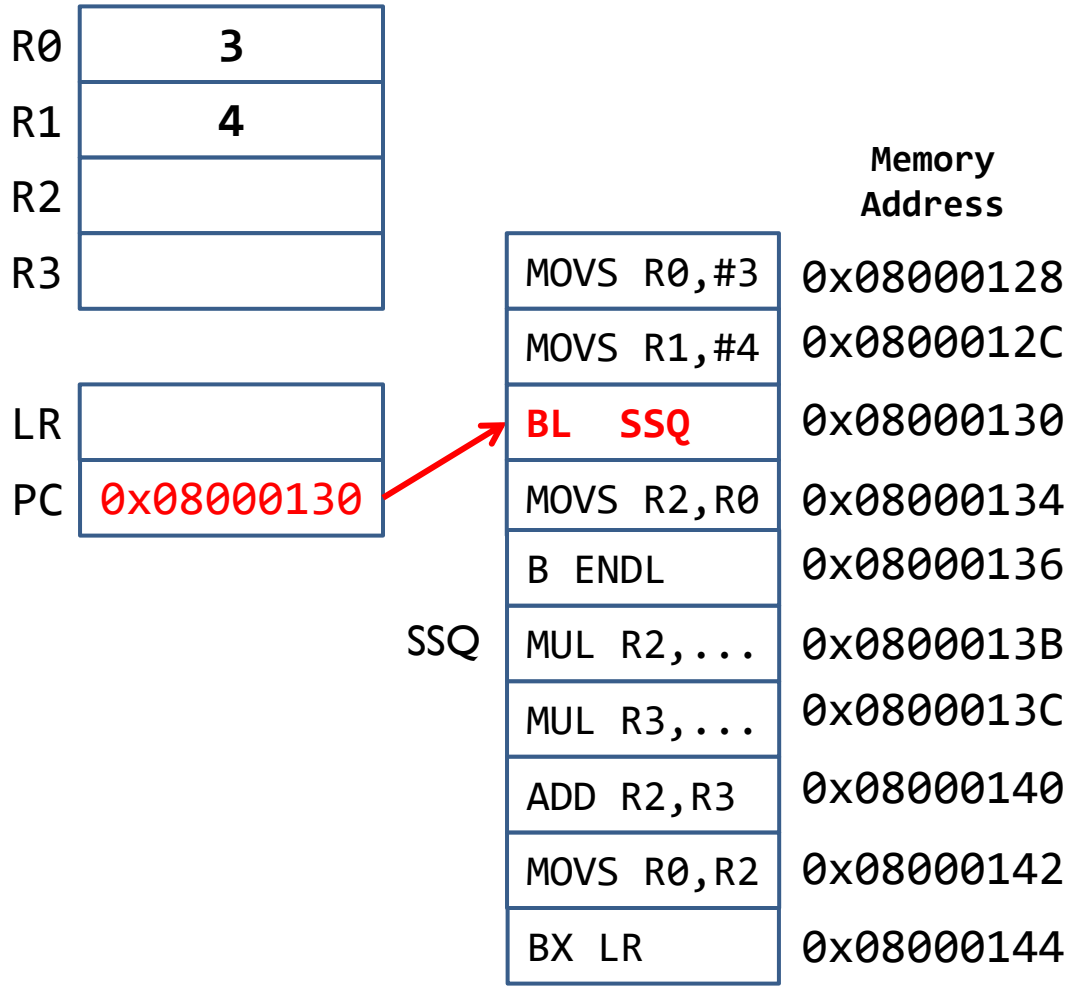
Example: SSQ(3, 4)

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



Example: SSQ(3, 4)

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



Example: SSQ(3, 4)

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

R0	3
R1	4
R2	
R3	

LR	0x08000134
PC	0x0800013B

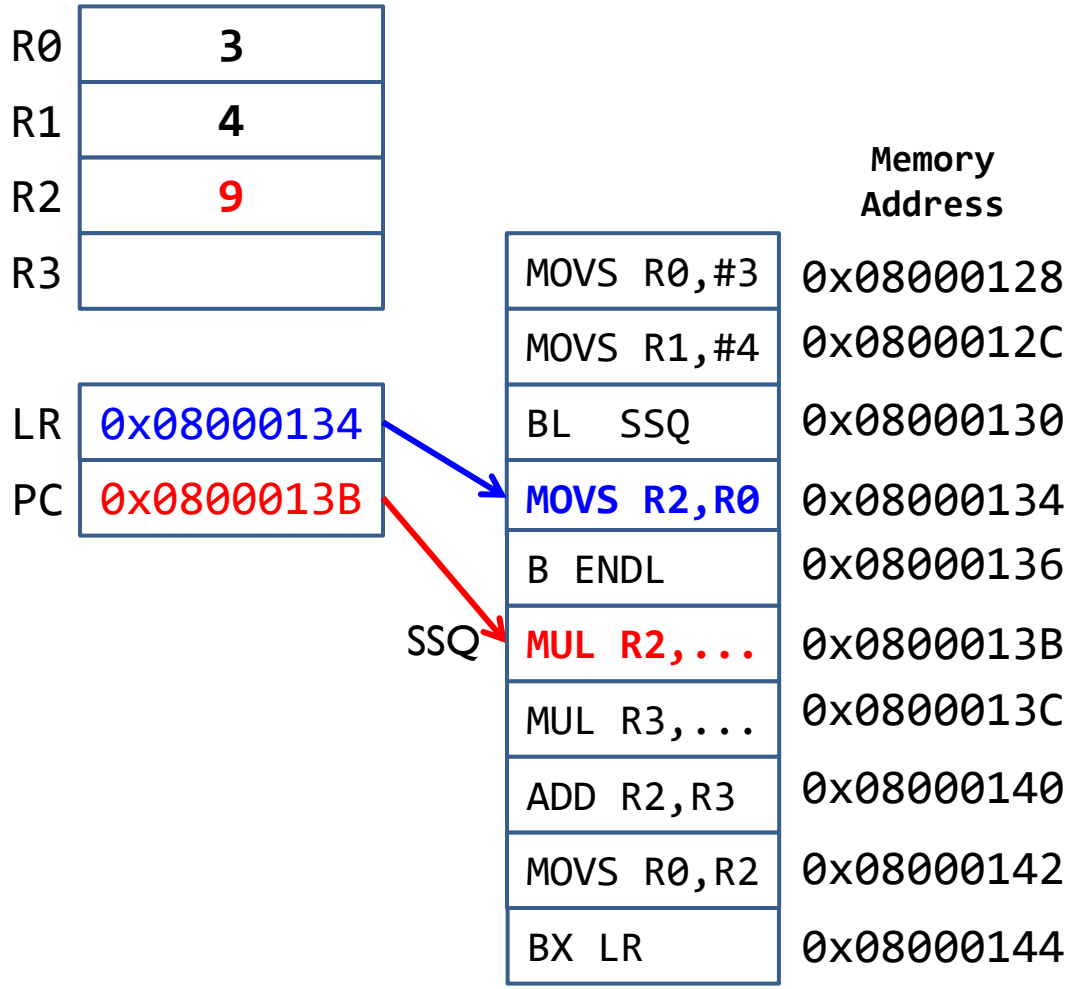
Memory Address	
0x08000128	MOVS R0,#3
0x0800012C	MOVS R1,#4
0x08000130	BL SSQ
0x08000134	MOVS R2,R0
0x08000136	B ENDL
0x0800013B	SSQ MUL R2,...
0x0800013C	MUL R3,...
0x08000140	ADD R2,R3
0x08000142	MOVS R0,R2
0x08000144	BX LR

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 LR.

Example: SSQ(3, 4)

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

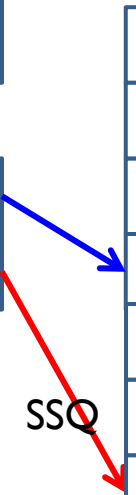


Example: SSQ(3, 4)

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

R0	3
R1	4
R2	9
R3	16

LR	0x08000134
PC	0x0800013C



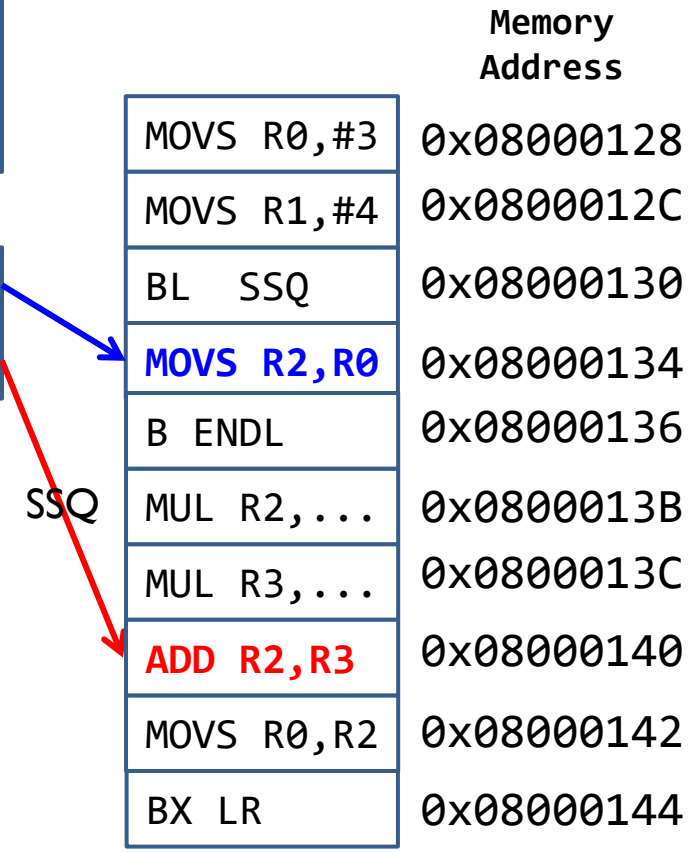
	Memory Address
MOVS R0,#3	0x08000128
MOVS R1,#4	0x0800012C
BL SSQ	0x08000130
MOVS R2,R0	0x08000134
B ENDL	0x08000136
MUL R2,...	0x0800013B
MUL R3,...	0x0800013C
ADD R2,R3	0x08000140
MOVS R0,R2	0x08000142
BX LR	0x08000144

Example: SSQ(3, 4)

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

R0	3
R1	4
R2	25
R3	16

LR	0x08000134
PC	0x08000140



Example: SSQ(3, 4)

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

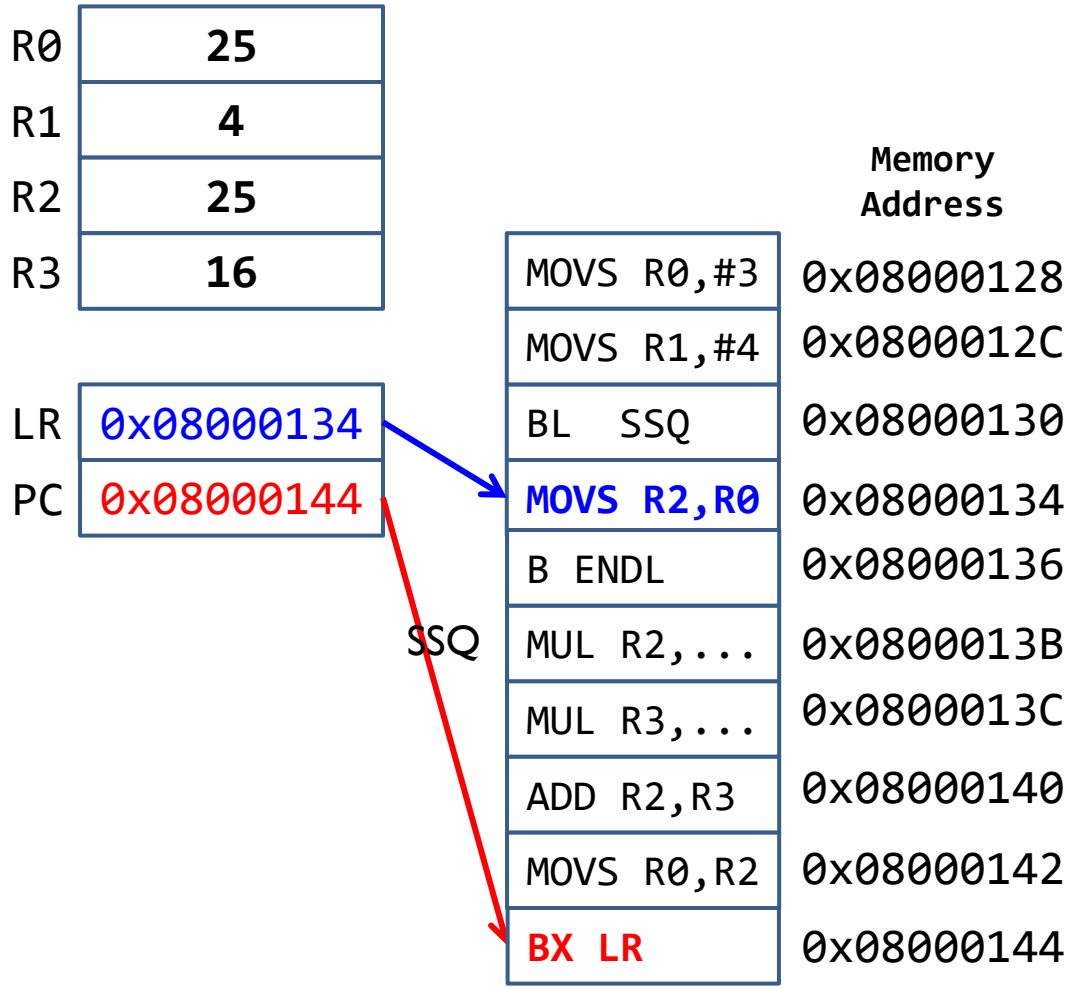
R0	25
R1	4
R2	25
R3	16

LR	0x08000134
PC	0x08000142

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

Example: SSQ(3, 4)

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



Example: SSQ(3, 4)

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

R0	25
R1	4
R2	25
R3	16

LR	0x08000134
PC	0x08000134

Copy LR to PC when returning from a subroutine!

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

Example: SSQ(3, 4)

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

R0	25
R1	4
R2	25
R3	16

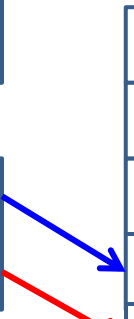
LR	0x08000134
PC	0x08000134

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

Example: SSQ(3, 4)

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

R0	25
R1	4
R2	25
R3	16
LR	0x08000134
PC	0x08000136



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

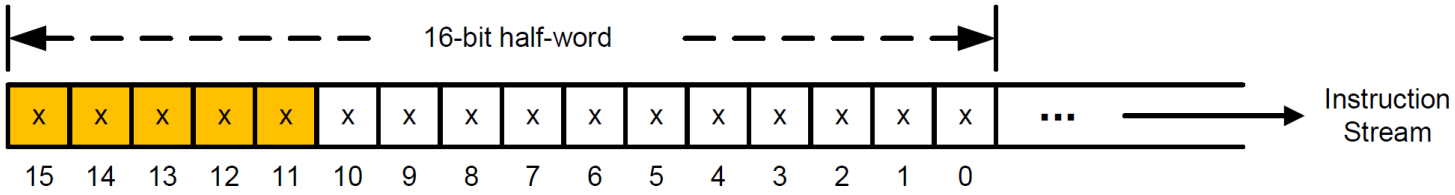
Realities

- In the previous example,
  - PC is incremented by 2 or 4.
  - The least significant bit of LR is always 0.

Well, I lied!

Realities

- 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.

- The least significant bit of LR is always **1** for ARM Cortex-M
  - This bit is used to control the processor mode:
    - `0` = ARM, 1 = THUMB
  - Cortex-M only supports THUMB.

Summary

- How to call a subroutine?
  - Branch with link: **`BL subroutine`**
- How to return the control back to the caller?
  - Branch and exchange: **`BX LR`**
- 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
- How to return a value in a subroutine?
  - Value is returned in `r0`
- How to preserve the running environment for the caller?
  - (to be covered)

Common Coding Patterns

- Callee returns a constant in r0.
  - mov r0,#17 @ r0 is return value register
  - bx lr @ return from function
- Callee saves some registers, does some arithmetic, and returns the result in r0.
  - push {r4-r7,lr}
  - mov r4, #10
  - mov r5, #100
  - add r0,r4,r5
  - pop {r4-r7,pc} @ pop saved lr value into PC to return from function
- Callee calls another function (nested function calls)
  - push {lr} @ must save LR if we call our own function
  - mov r0, #123 @ r0 is first function parameter
  - bl print_int @ call function print_int(123)
  - pop {pc} @ pop saved lr into PC to return from function
- Callee return: restore previously-pushed LR, then jump to LR (POP {lr}; BX lr), or equivalently, pop previously-pushed LR to PC
  - POP {pc} ≡ POP {lr}; BX lr

Common Coding Patterns

- Memory access: first put memory address into register, then load memory content at that address
  - adr r2, mydata @Compute address of label mydata using a PC-relative add and put that address in r2
  - ldr r0, [r2] @Dereference that address, loading the 32-bit word stored at mydata into r0
  - bx lr
  - mydata:
  - .word 123
- Or
  - ldr r2,=mydata @ pseudo-instruction that loads absolute address of mydata from a nearby literal pool into r2
  - ldr r0,[r2]
  - bx lr
  - mydata:
  - .word 123
- adr vs. ldr
  - If mydata is in range for adr (c.f.,), both forms will leave r2 holding the same address at run time.
  - Out-of-range labels: adr may fail; ldr =mydata still works, but it incurs one memory access
- Note: pseudo-instruction LDR Rd, =X loads a value if X is a constant, and an address if X is a symbol,
  - e.g., "LDR r2, =0x55555555" loads the value 0x55555555 from memory

References

- Lecture 29. Calling a subroutine
  - https://www.youtube.com/watch?v=xt2Q9n1Udb4&list=PLRJhV4hUhIymmp5CCeIFPyxbknsdcXCc8&index=29
- Lecture 30. Passing Arguments to a Subroutine
  - https://www.youtube.com/watch?v=DGKjFKjxAYs&list=PLRJhV4hUhIymmp5CCeIFPyxbknsdcXCc8&index=31