Chapter 8
Passing Parameters to Subroutines via Registers

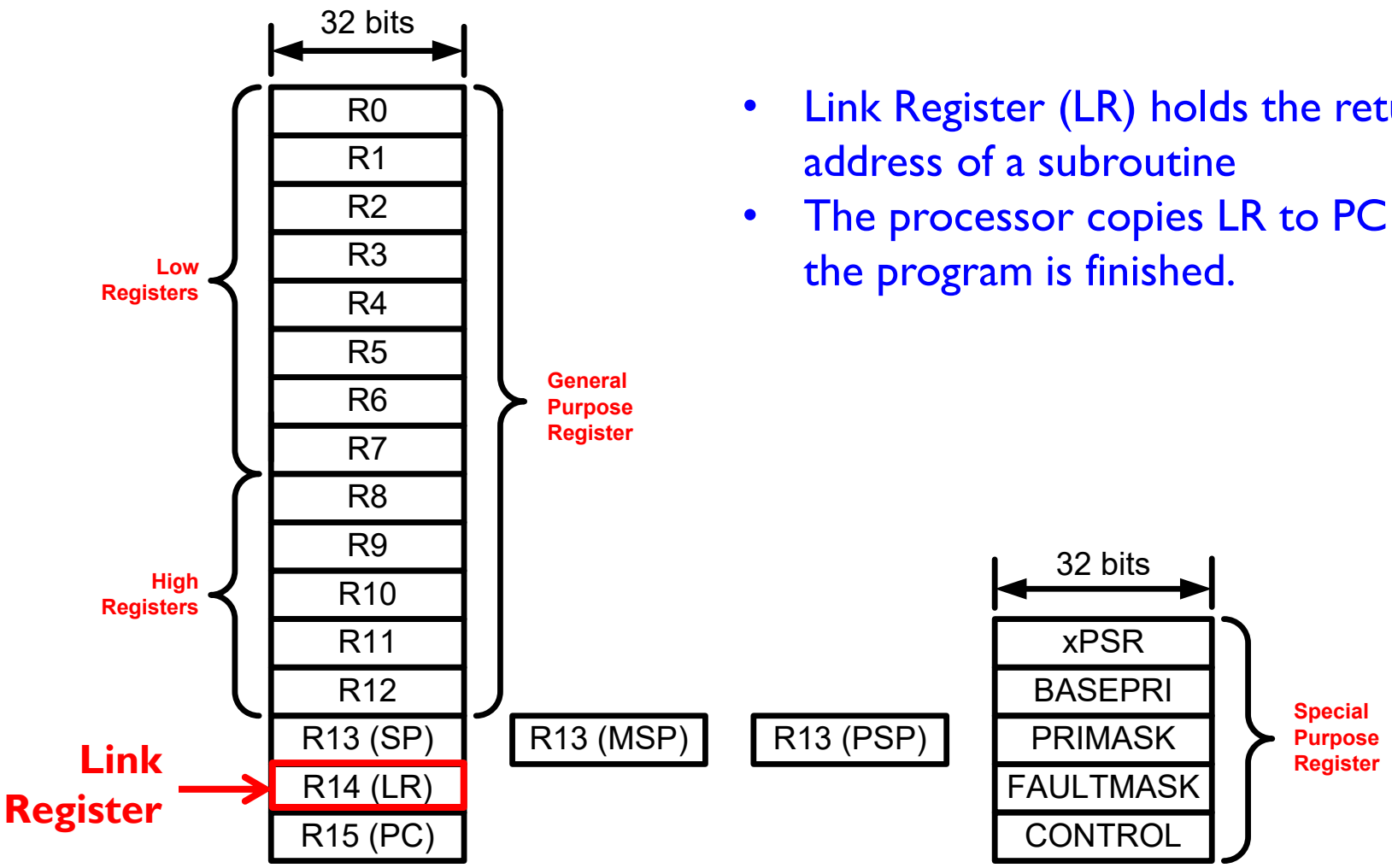
Z. Gu

Fall 2025

Overview

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

Link Register (LR)



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

Call a Subroutine (BL)

Branch with Link

BL *label*

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

- Notes:
  - *label* is name of subroutine
  - Compiler translates label to memory address
  - After call, LR holds return address (the instruction following the call)

Caller Program

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

Subroutine/Callee

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

Return from a Subroutine (BX LR)

Caller Program

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

Branch and Exchange

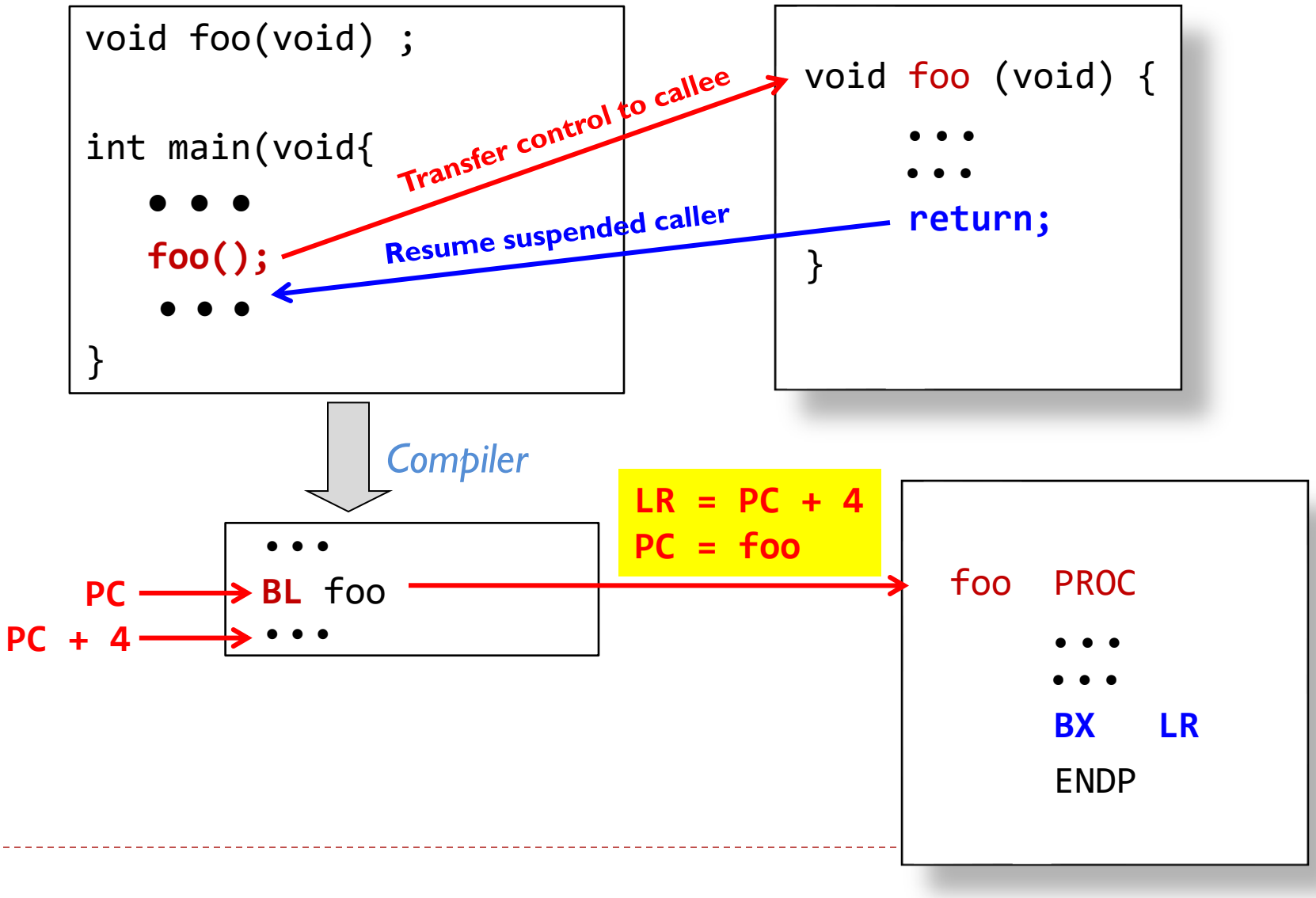
BX *LR*

- PC = LR

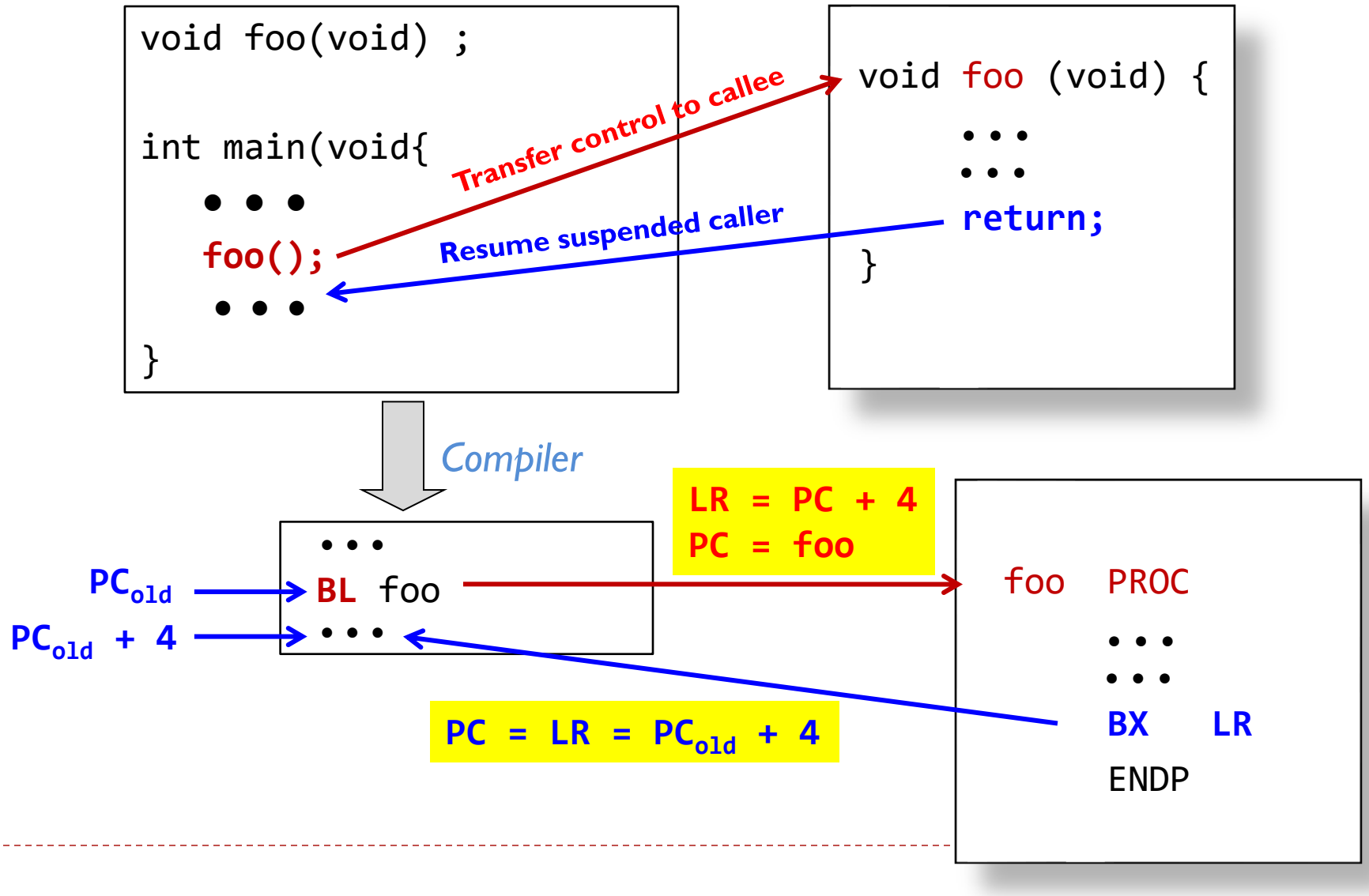
Subroutine/Callee

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

BL and BX

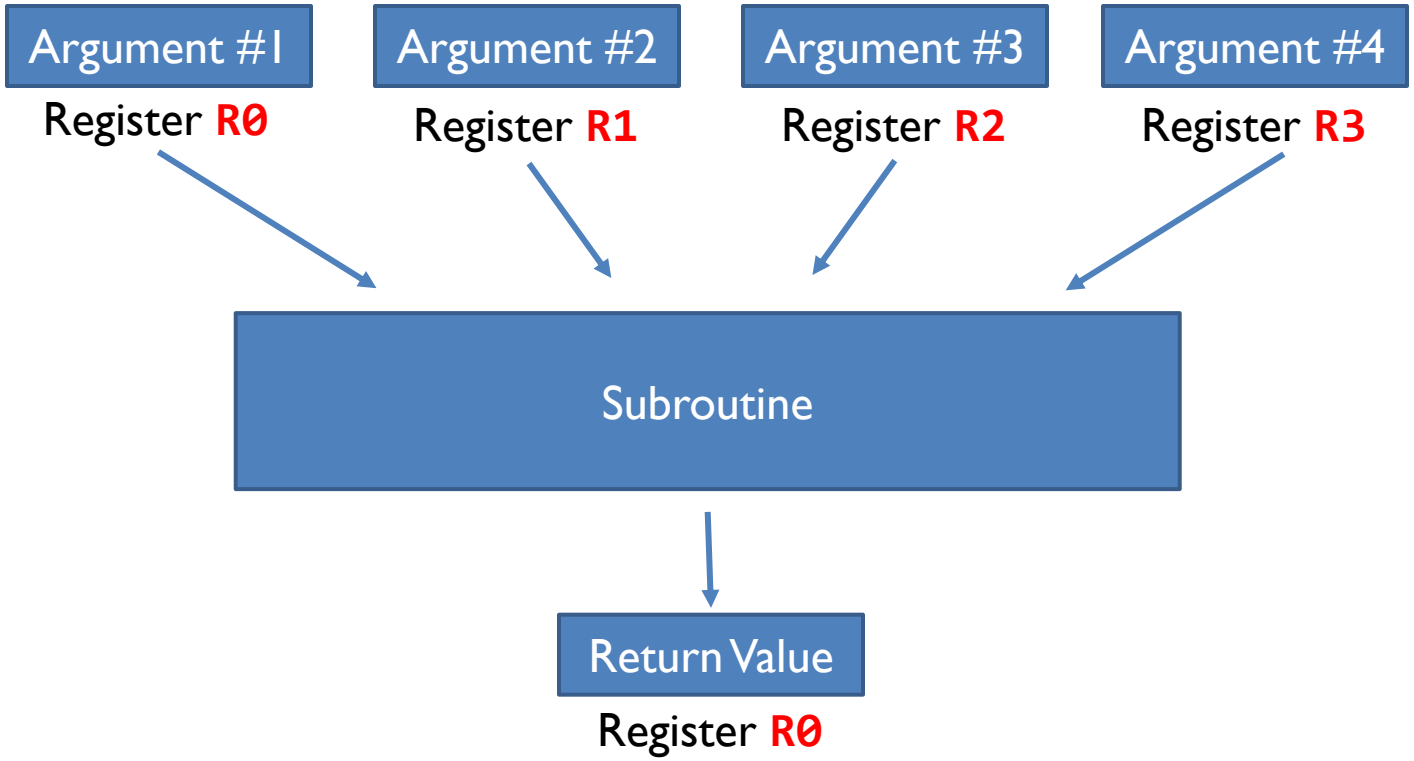

void foo(void) ;
int main(void{
• • •
foo();
• • •
}
Transfer control to callee
Resume suspended caller
void foo (void) {
• • •
• • •
return;
}
Compiler
• • •
PC
BL foo
PC + 4
• • •
LR = PC + 4
PC = foo
foo PROC
• • •
• • •
BX LR
ENDP

BL and BX

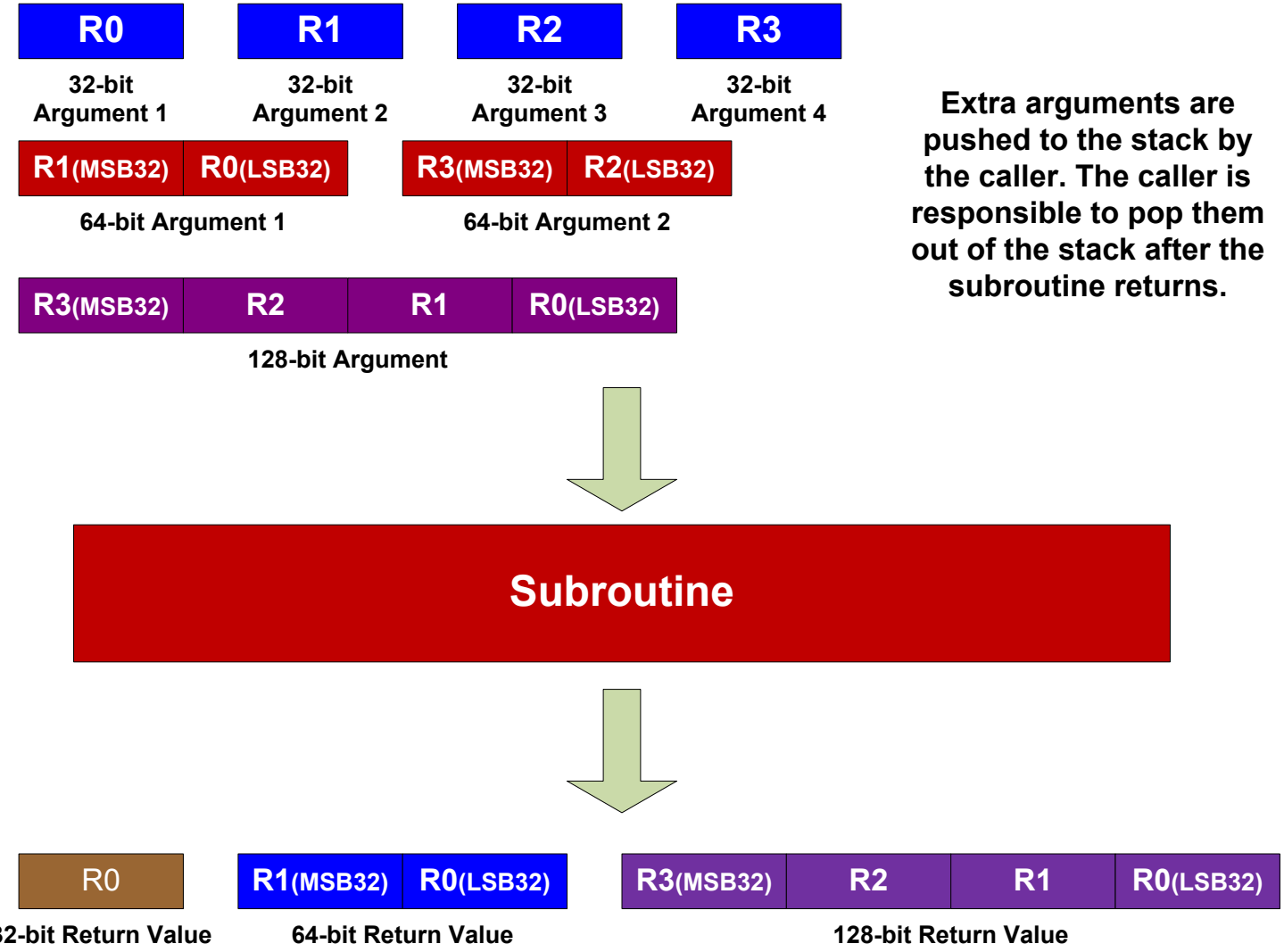

void foo(void) ;
int main(void{
foo();
}
Transfer control to callee
Resume suspended caller
void foo (void) {
return;
}
Compiler
LR = PC + 4
PC = foo
PC_old
PC_old + 4
BL foo
foo PROC
BX LR
ENDP
PC = LR = PC_old + 4

Passing arguments and Returning Value

- ARM Architecture Procedure Call Standard (AAPCS)
- First four registers are used to pass argument values into a subroutine and to return a value from a subroutine



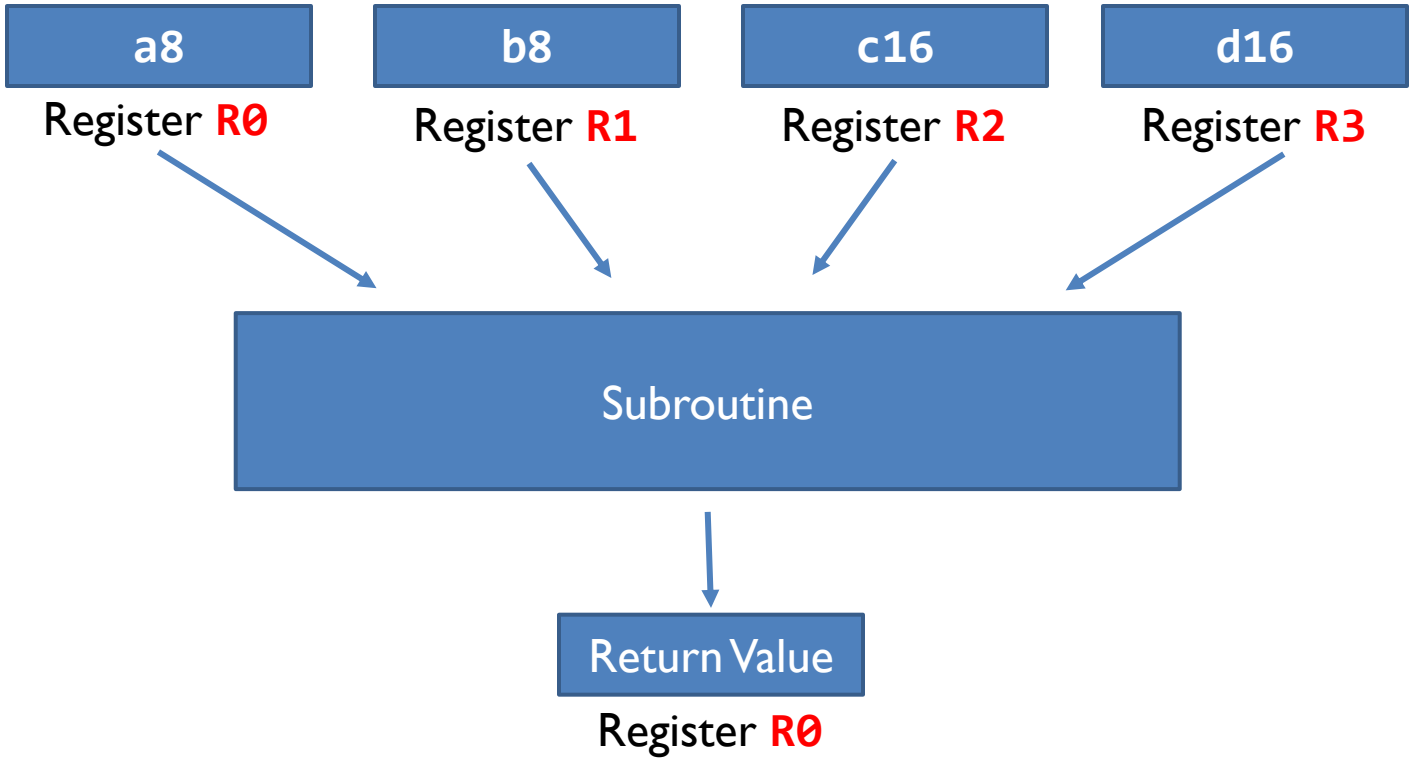
Passing arguments and Returning Value



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.

Passing arguments and Returning Value

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



Passing arguments and Returning Value

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

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

Caller

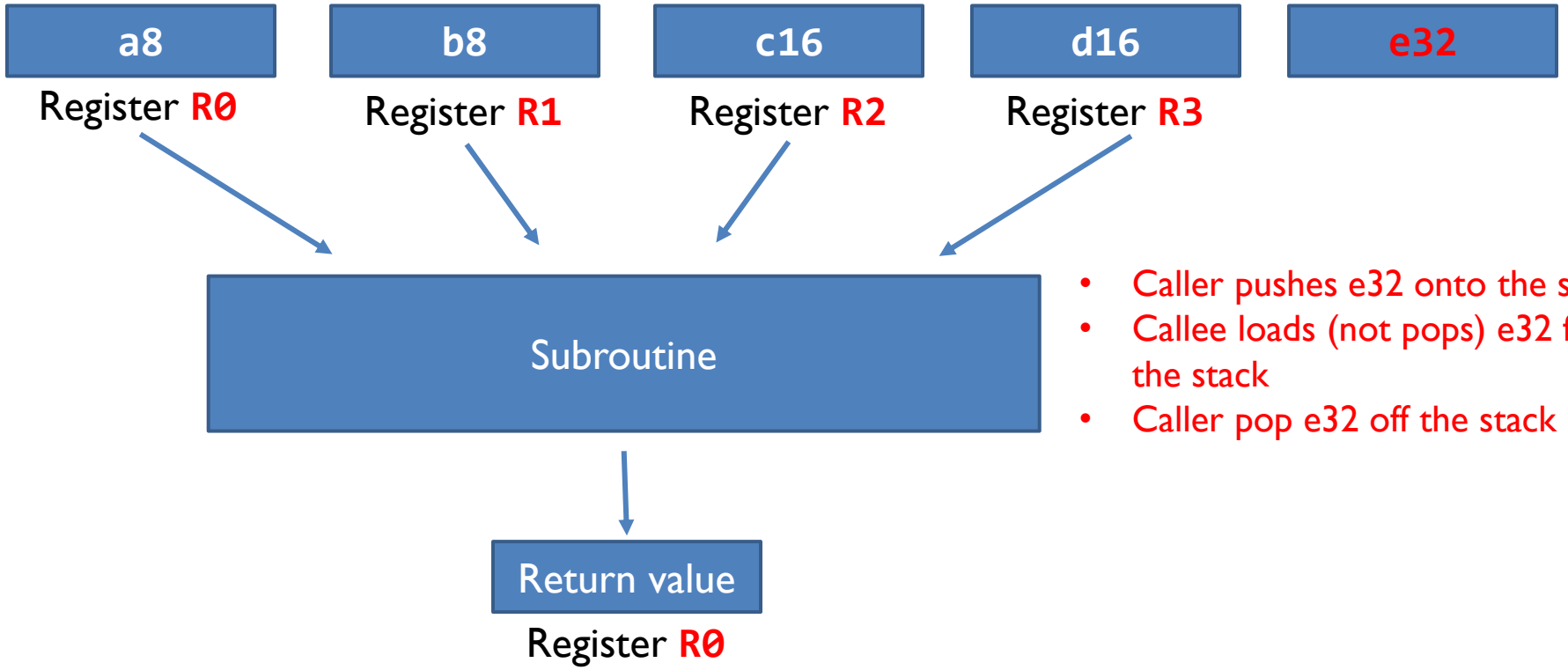
```
MOVS r0, #1   ; a8
MOVS r1, #2   ; b8
MOVS r2, #3   ; c16
MOVS r3, #4   ; d16
BL   sum
```

Callee

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

Passing arguments and Returning Value

```
uint32_t sum(uint8_t a, uint8_t b, uint16_t c, uint16_t d, uint32_t e);
```



- Caller pushes e32 onto the stack
- Callee loads (not pops) e32 from the stack
- Caller pop e32 off the stack

Passing arguments and Returning Value

```
uint32_t sum(uint8_t a8, uint8_t b8, uint16_t c16, uint16_t d16,
uint32_t e32);
s = sum(1, 2, 3, 4, 5);
```

Caller

```
MOVS r0, #5 ; e32
PUSH {r0}
MOVS r0, #1 ; a8
MOVS r1, #2 ; b8
MOVS r2, #3 ; c16
MOVS r3, #4 ; d16
BL   sum
...
POP {r0}
```

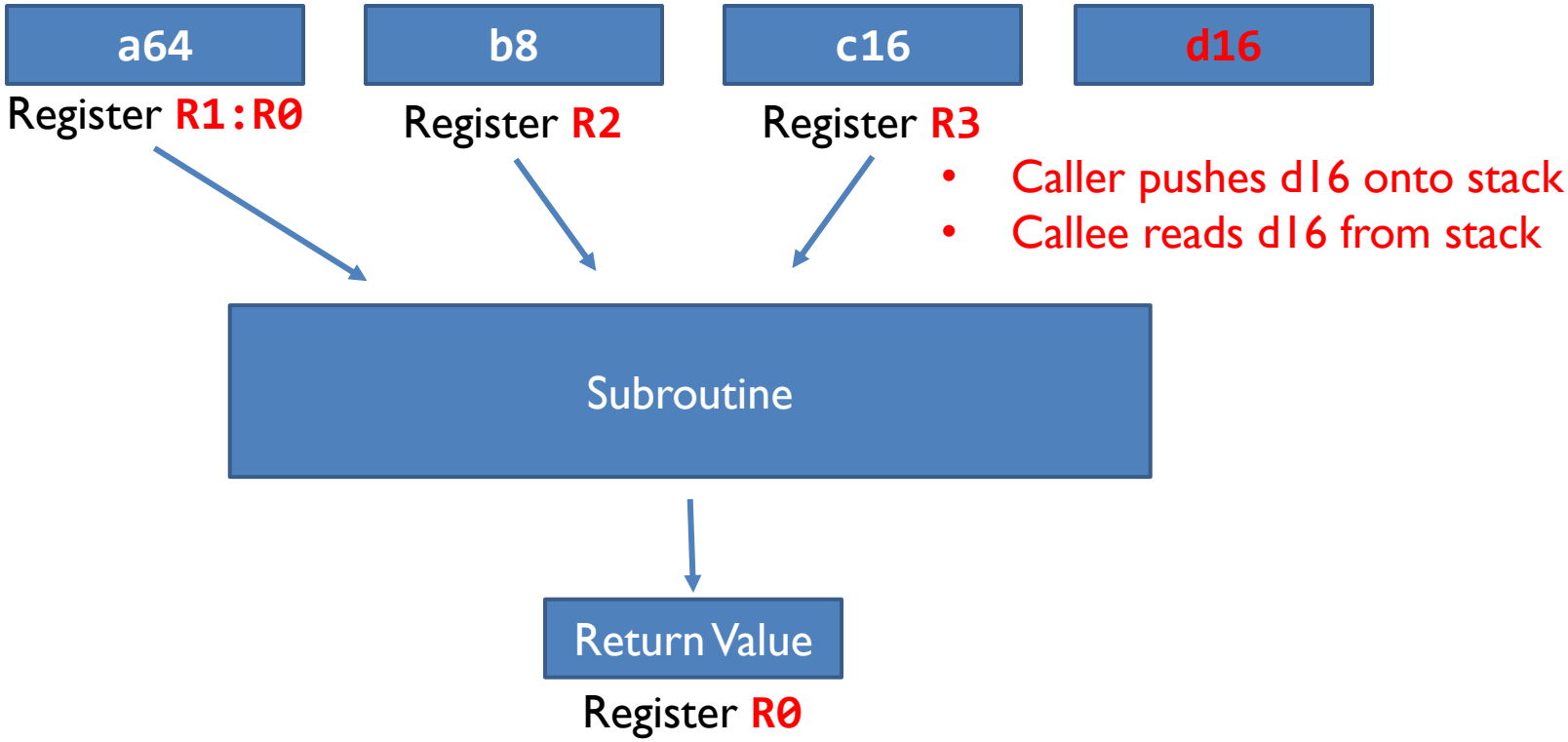
Callee

```
sum PROC
  ADD r0, r0, r1    ; a8 + b8
  ADD r0, r0, r2    ; add c16
  ADD r0, r0, r3    ; add d16
  LDR r1, [sp, #0] ; read argument e32
  ADD r0, r0, r1    ; add e32
  BX  LR
  ENDP
```

The caller is responsible to pop extra arguments out of the stack after the subroutine returns.

Passing arguments and Returning Value

```
uint64_t sum(uint64_t a64, uint8_t b8, uint16_t c16, uint16_t d16);
```



Returning Value

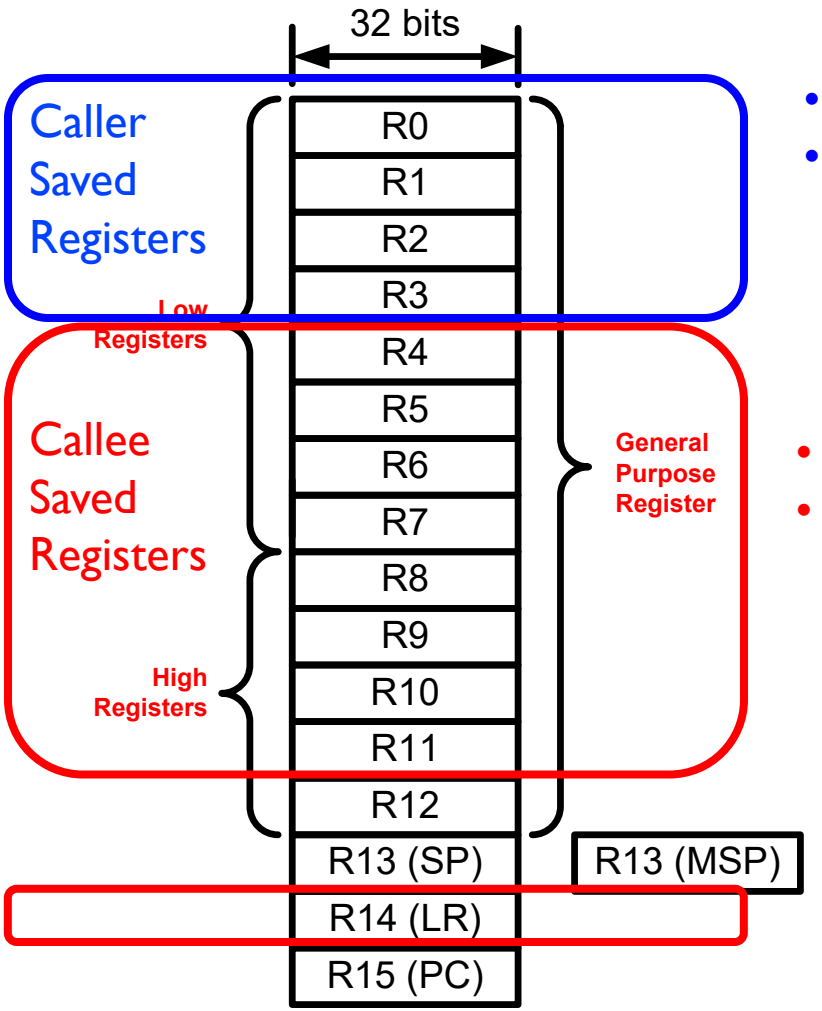
```
uint32_t s32;
uint32_t sum(uint8_t a8, uint8_t b8, uint16_t c16, uint16_t d16);

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
ADDS r0, r0, #100
LDR  r4, =s32  ; Get memory address of s32
STR  r0, [r4]  ; Save returned result to s32
```

ARM Procedure Call Standard

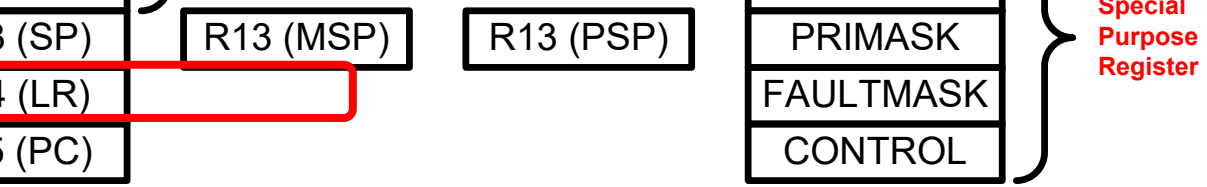
Register	Usage	Subroutine Preserved	Notes
r0	Argument 1 and return value	No	If return has 64 bits, then r0:r1 hold it. If argument 1 has 64 bits, r0:r1 hold it.
r1	Argument 2	No	
r2	Argument 3	No	If the return has 128 bits, r0-r3 hold it.
r3	Argument 4	No	If more than 4 arguments, use the stack
r4	General-purpose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



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

Example: SSQ(3, 4)

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

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

R1: second argument

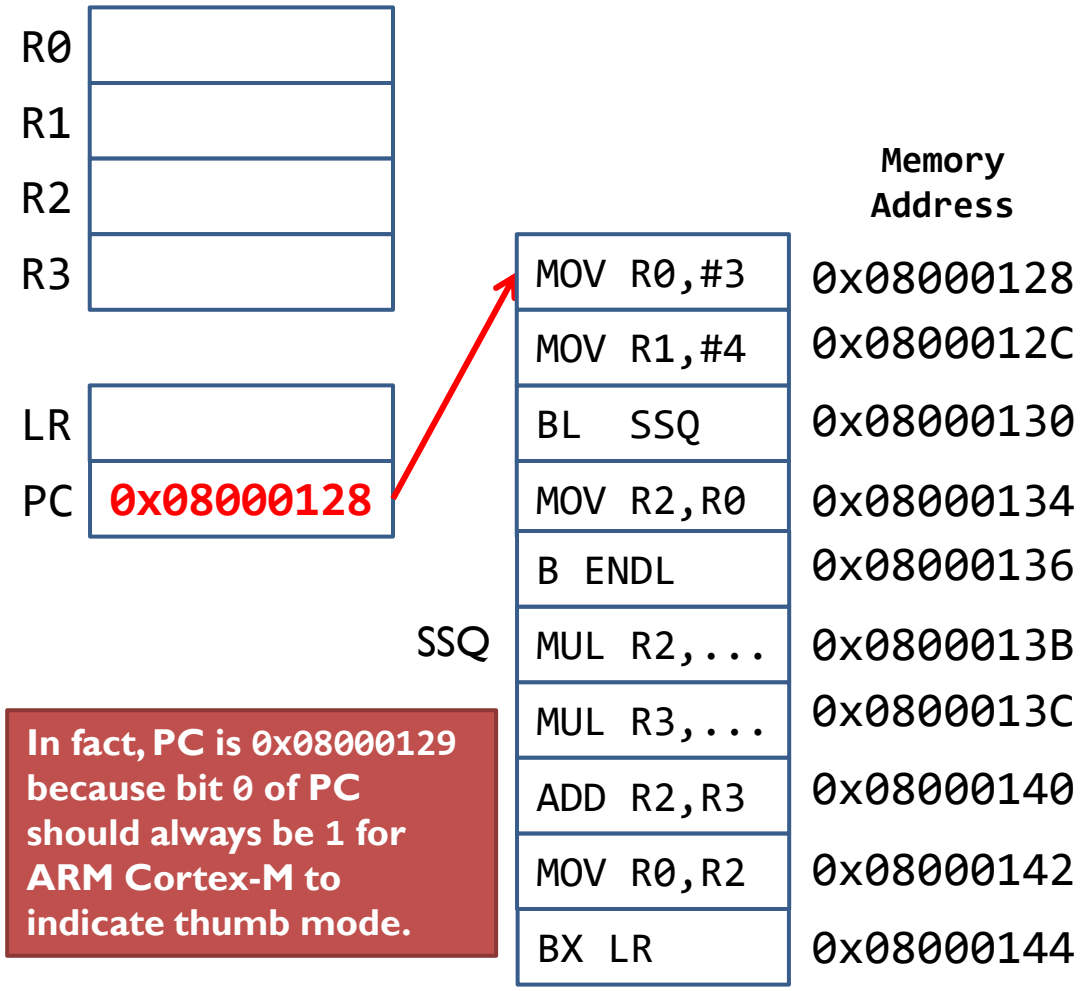
R0: first argument

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

R0: Return Value

Example: SSQ(3, 4)

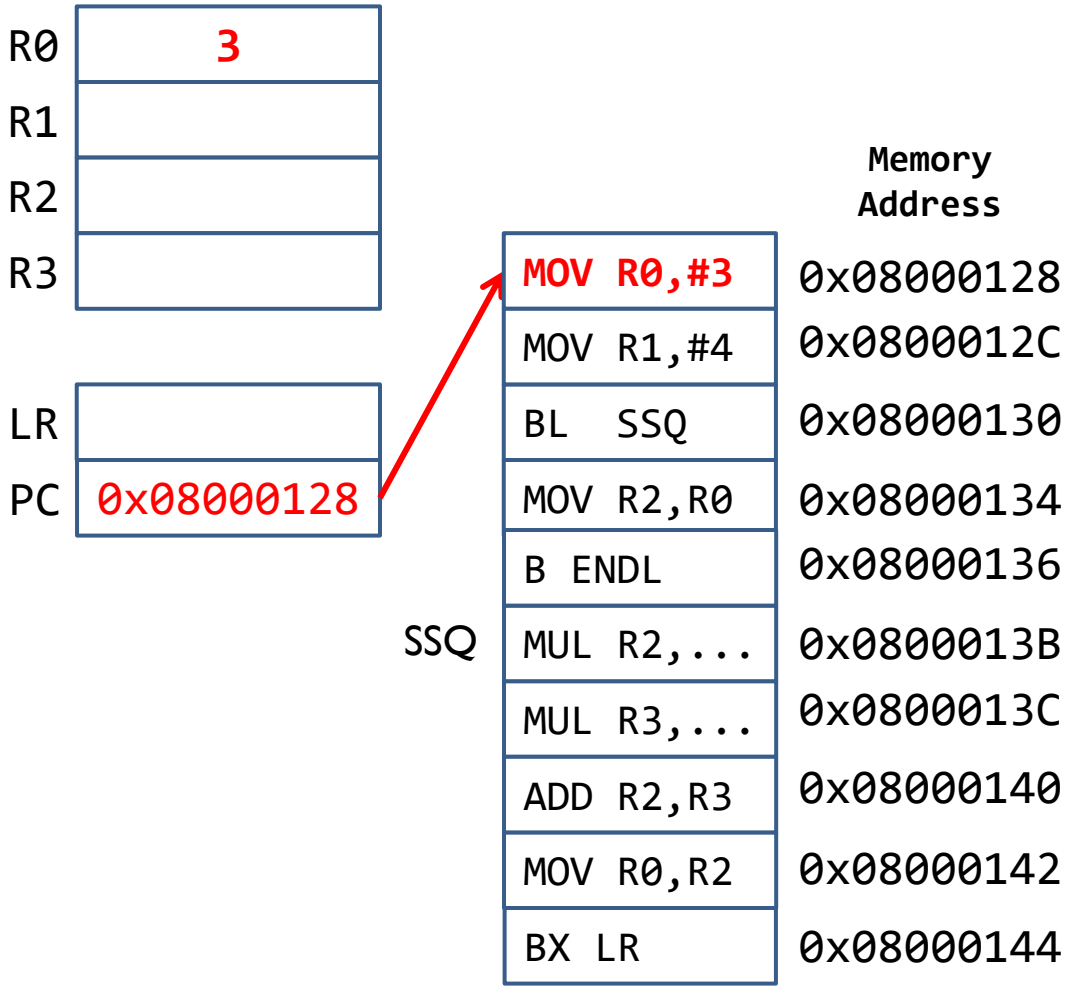
```
        MOV R0,#3
        MOV R1,#4
        BL  SSQ
        MOV R2,R0
        B ENDL
        ...
SSQ     PROC
        MUL R2,R0,R0
        MUL R3,R1,R1
        ADD R2,R2,R3
        MOV R0,R2
        BX  LR
        ENDP
ENDL    ...
```



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

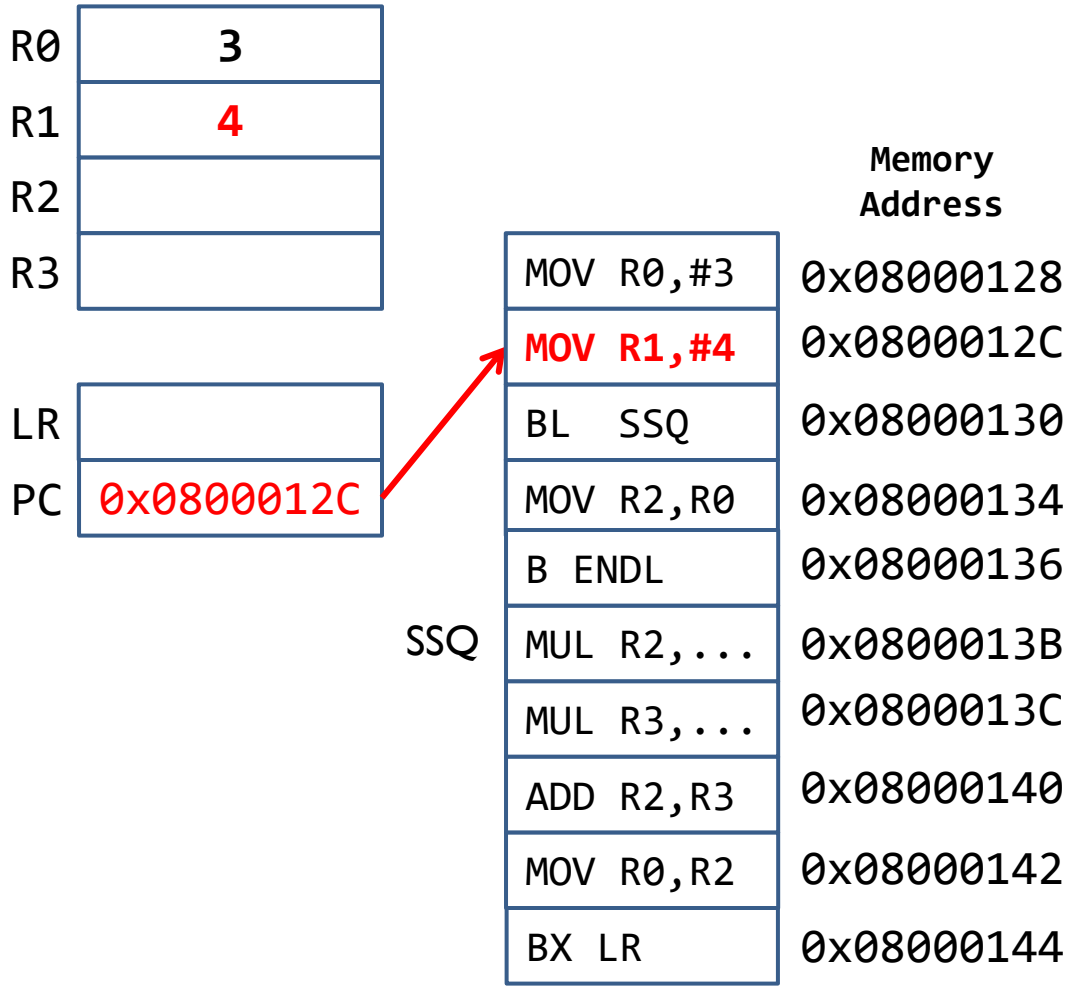
Example: SSQ(3, 4)

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



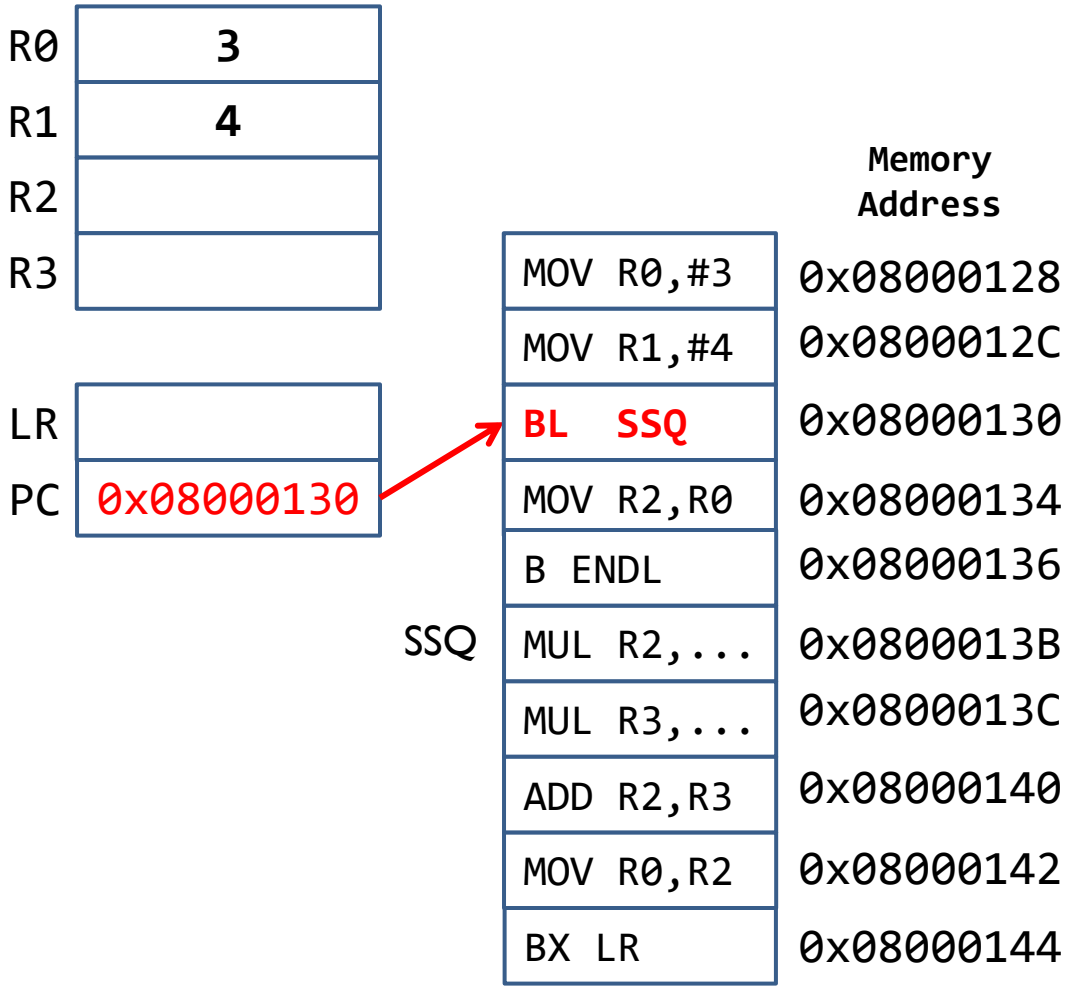
Example: SSQ(3, 4)

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



Example: SSQ(3, 4)

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

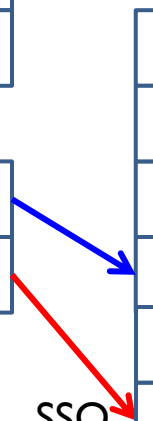


Example: SSQ(3, 4)

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

R0	3
R1	4
R2	
R3	

LR	0x08000134
PC	0x0800013B



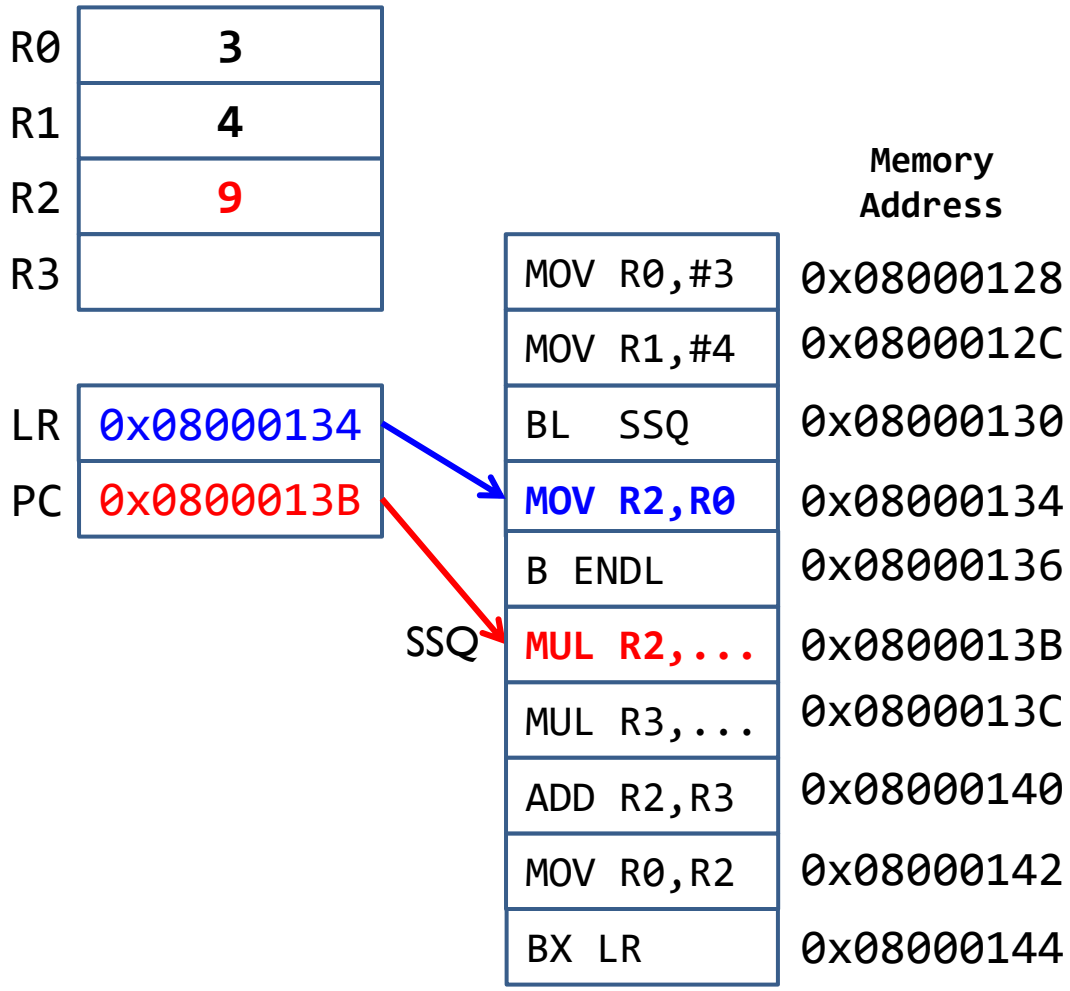
Memory Address	
0x08000128	MOV R0,#3
0x0800012C	MOV R1,#4
0x08000130	BL SSQ
0x08000134	MOV R2,R0
0x08000136	B ENDL
0x0800013B	MUL R2,...
0x0800013C	MUL R3,...
0x08000140	ADD R2,R3
0x08000142	MOV R0,R2
0x08000144	BX LR

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.

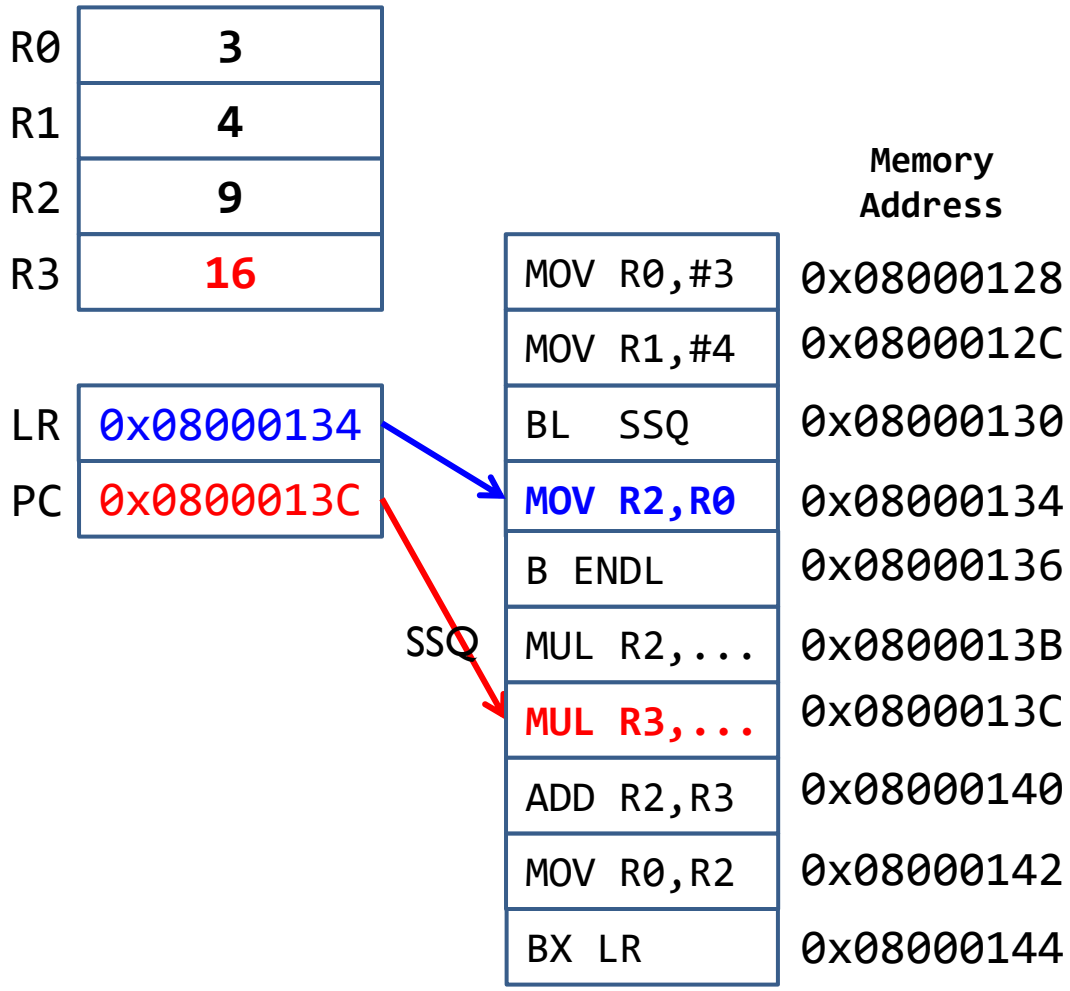
Example: SSQ(3, 4)

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



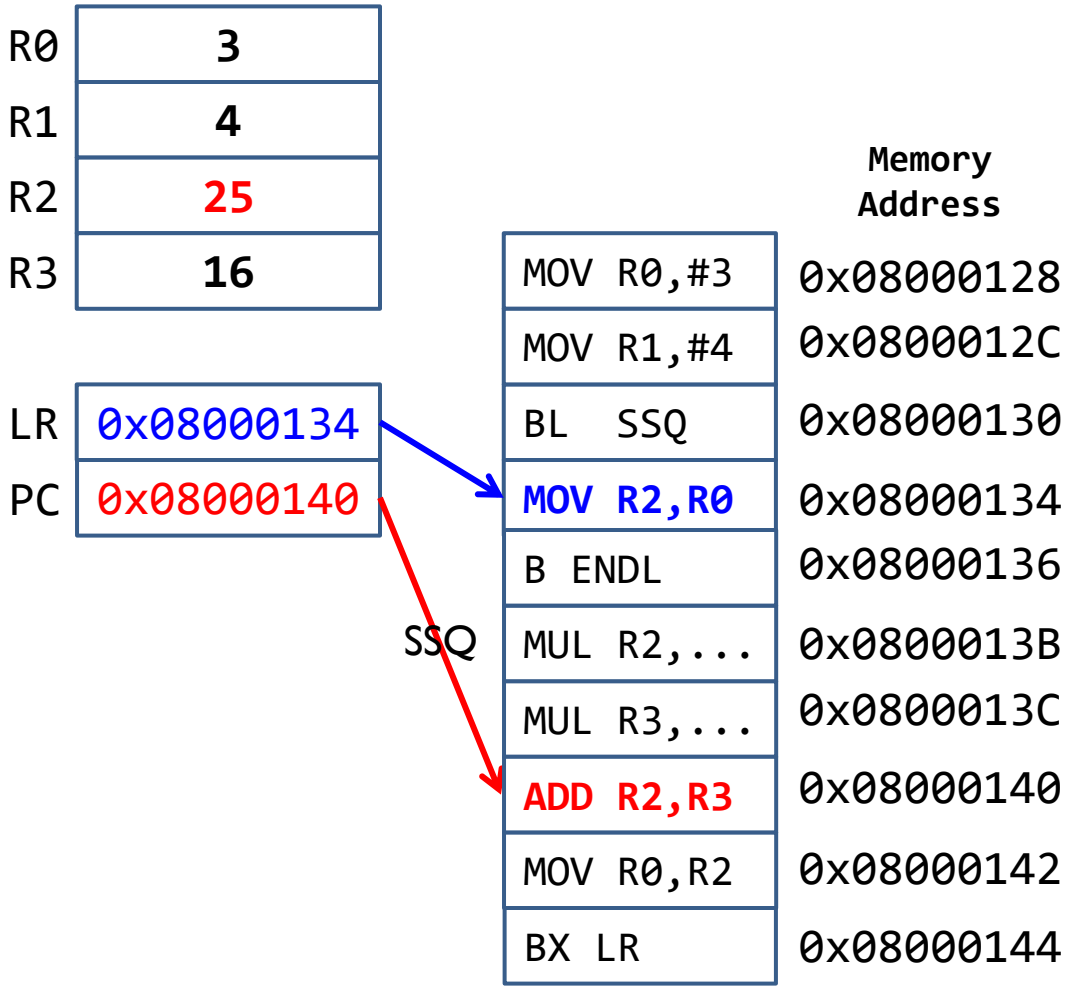
Example: SSQ(3, 4)

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



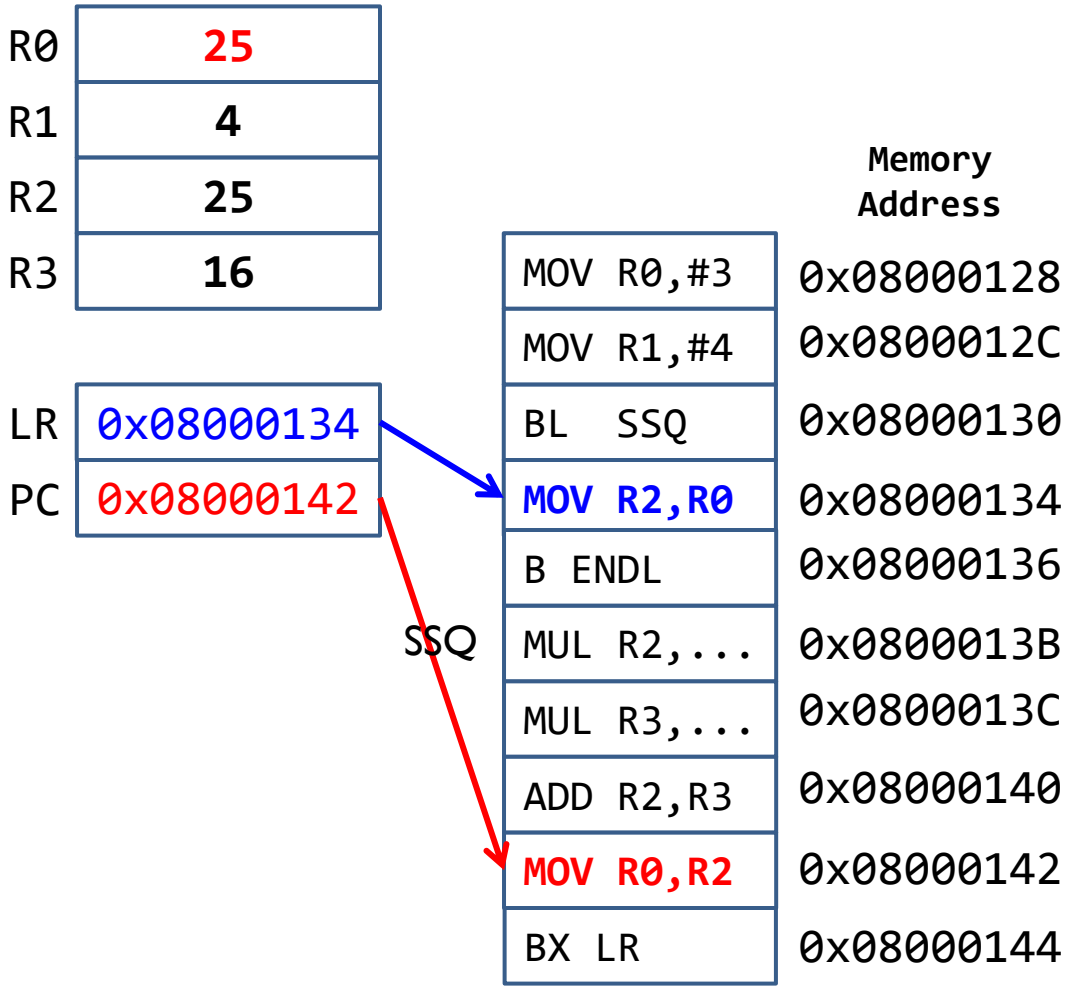
Example: SSQ(3, 4)

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



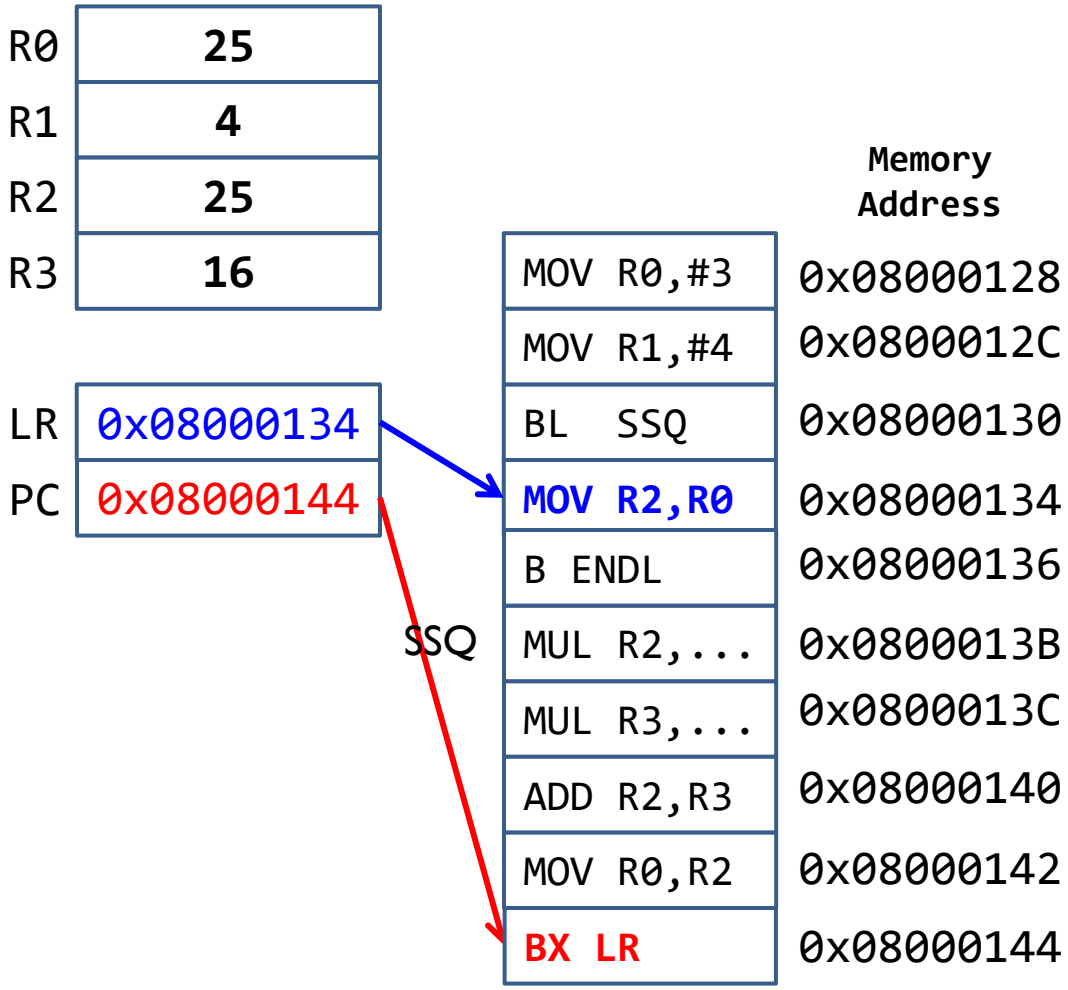
Example: SSQ(3, 4)

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



Example: SSQ(3, 4)

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



Example: SSQ(3, 4)

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

R0	25
R1	4
R2	25
R3	16

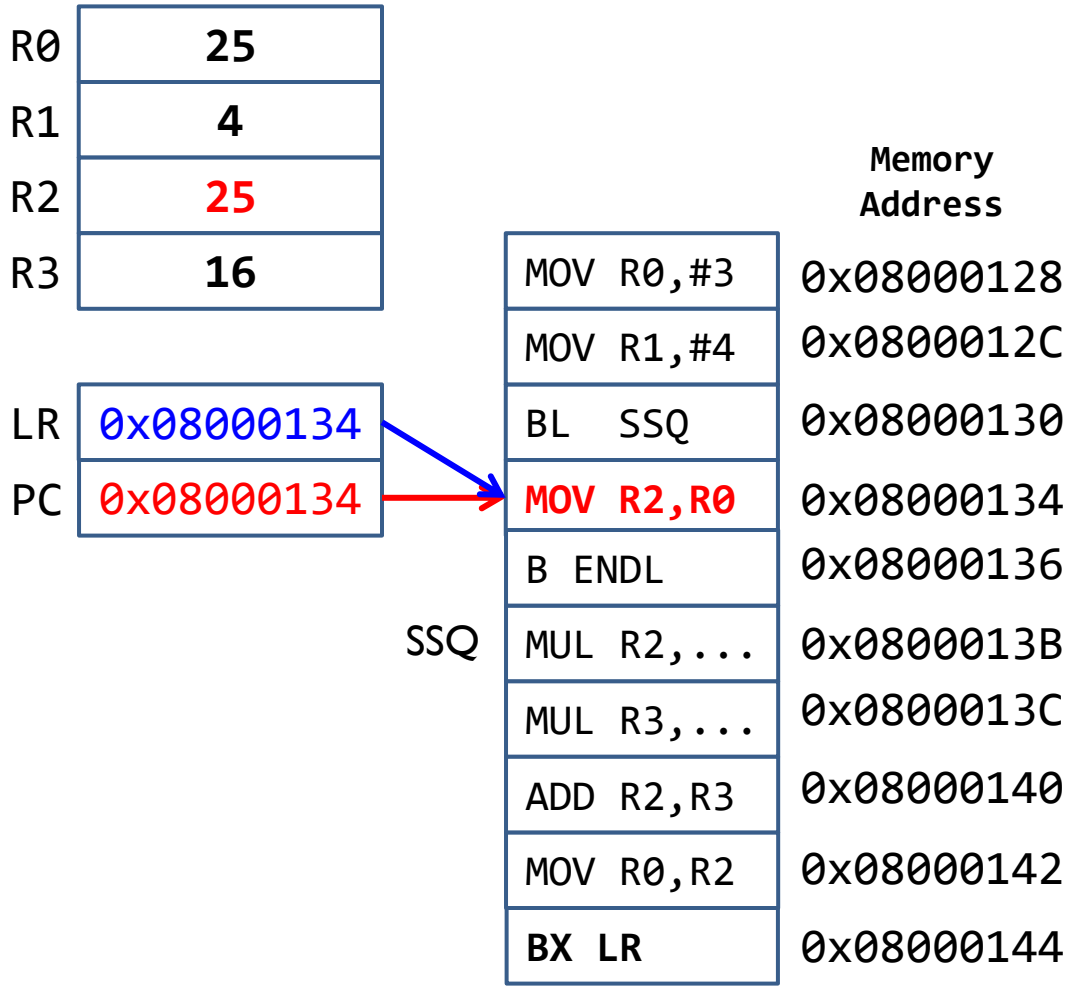
LR	0x08000134
PC	0x08000134

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

Copy LR to PC when returning from a subroutine!

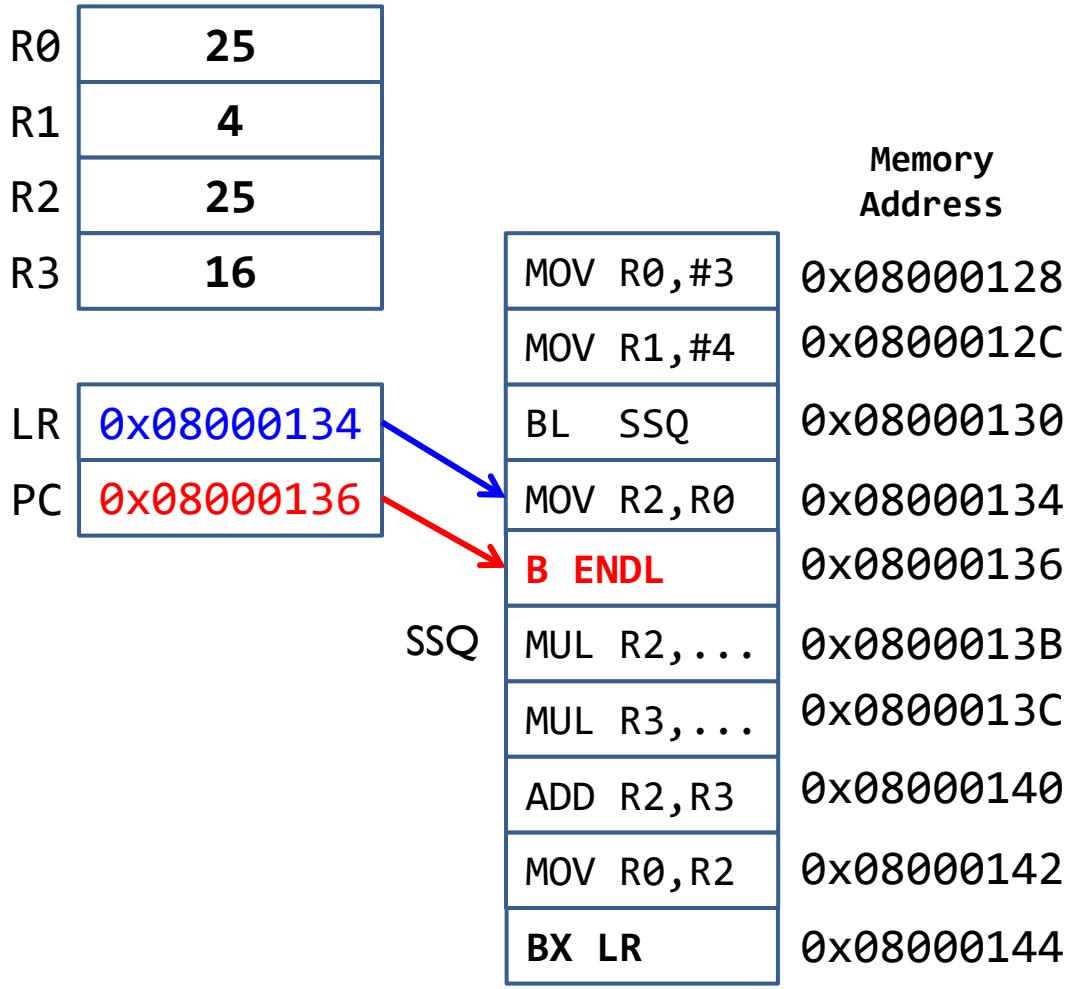
Example: SSQ(3, 4)

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



Example: SSQ(3, 4)

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



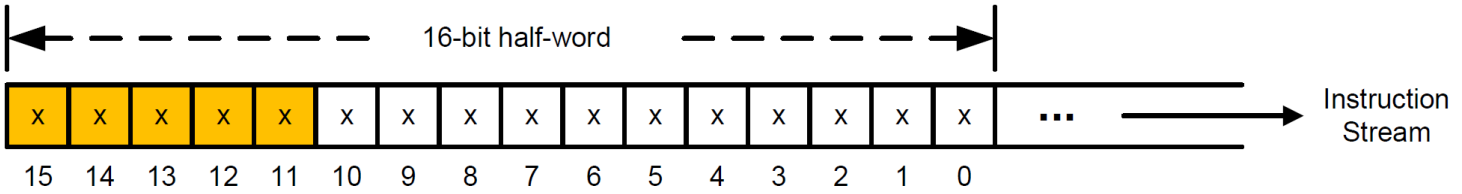
Realities

- 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)