

Embedded Systems with ARM Cortex-M Microcontrollers in Assembly Language and C

Chapter 8 Subroutines Exercises ANS

Z. Gu

Fall 2025

Stack

PUSH {Rd} == STMDB SP!, {Rd} == STMFD SP!, {Rd}

- ▶ SUB SP, SP, #4 @ SP = SP-4 (descending stack)
- ▶ STR Rd, [SP] @ (*SP) = Rd (full stack)

Push multiple registers

PUSH {r1, r2, r3,
r7}

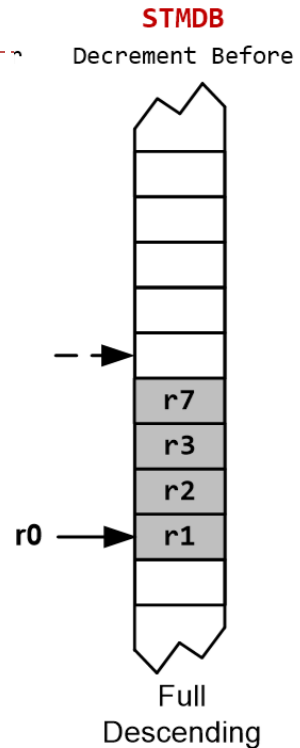


They are equivalent.

PUSH {r7, r2, r3, r1}



PUSH {r7}
PUSH {r3}
PUSH {r2}
PUSH {r1}



- SP is decremented before PUSH (pre-decrement).
- 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} == LDMIA SP!, {Rd} == LDMFD SP!, {Rd}

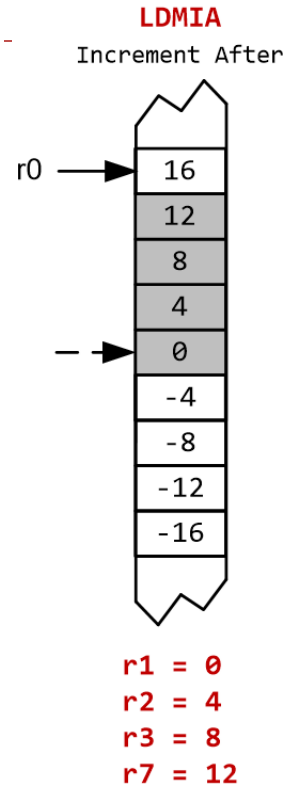
- ▶ LDR Rd, [SP] @ Rd = (*SP) (full stack)
- ▶ ADD SP, #4 @ SP = SP + 4 (Stack shrinks)

Pop multiple registers

They are equivalent.

POP {r1, r2, r3, r7} ↔ POP {r7, r3, r2, r1} ↔

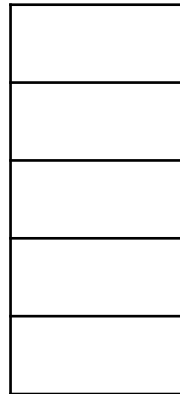
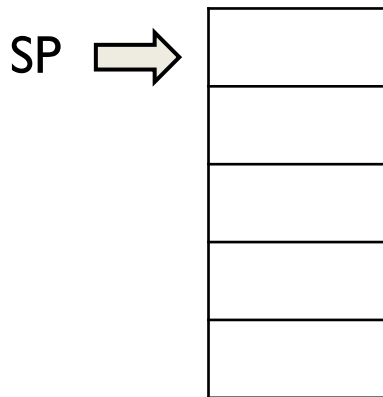
POP {r1}
POP {r2}
POP {r3}
POP {r7}



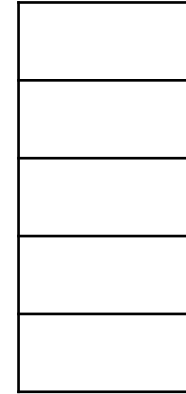
- SP is incremented after POP (post-increment).
- The order in which registers listed in the register list does not matter.
- When popping multiple registers, these registers are automatically **sorted by name** and **the lowest-numbered register** is loaded from the lowest memory address, i.e. **is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

Stack

- ▶ Initially, let $r0=0$, $r1=1$, $r2=2$.
- ▶ a) Execute `PUSH {r1,r2}`. Draw stack.
- ▶ b) Execute `POP {r0,r1}`. Draw stack.



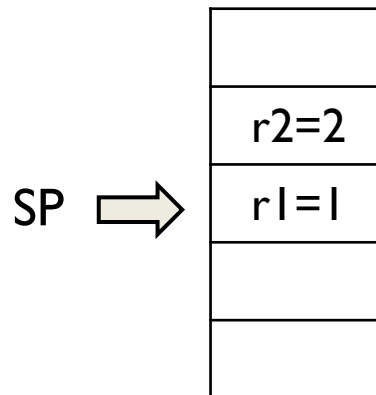
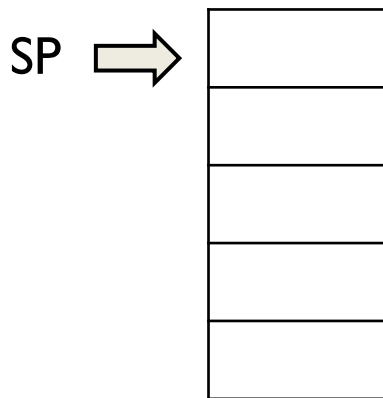
After `PUSH {r1,r2}`



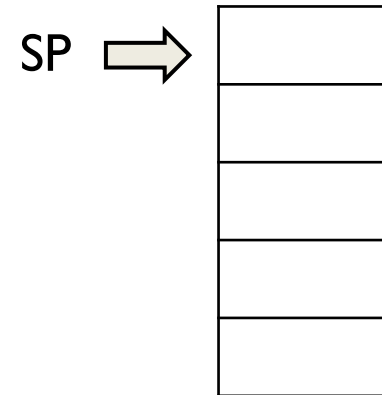
After `POP {r0,r1}`,
 $r0=?$, $r1=?$

Stack ANS

- ▶ Initially, let $r0=0$, $r1=1$, $r2=2$.
- ▶ a) Execute `PUSH {r1,r2}`. Draw stack.
- ▶ b) Execute `POP {r0,r1}`. Draw stack.



After `PUSH {r1,r2}`

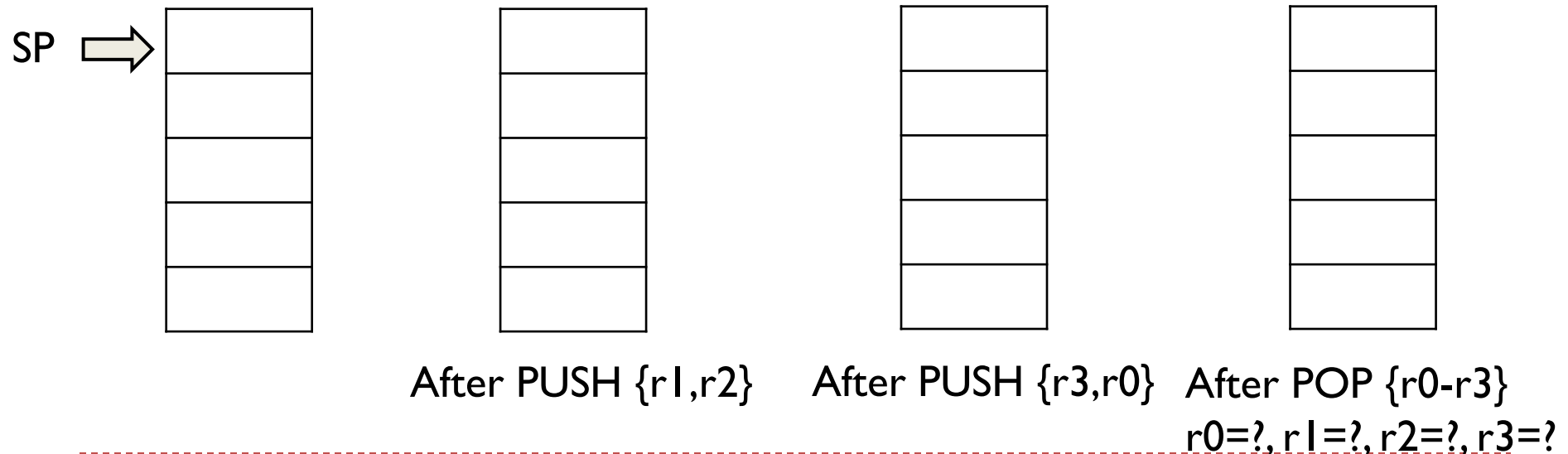


After `POP {r0,r1}`,
 $r0=1$, $r1=2$

Stack contains only values like 2, 1... I write $r2=2$, $r1=1$ in the figures for illustration purposes only.

Stack

- ▶ Initially, let $r0=0$, $r1=1$, $r2=2$, $r3=3$
- ▶ Execute
 - PUSH {r1,r2}
 - PUSH {r3,r0}
 - POP {r0-r3} (same as POP {r0, r1, r2, r3})
- ▶ Draw stack after each instruction. What is in registers after execution?



Stack ANS

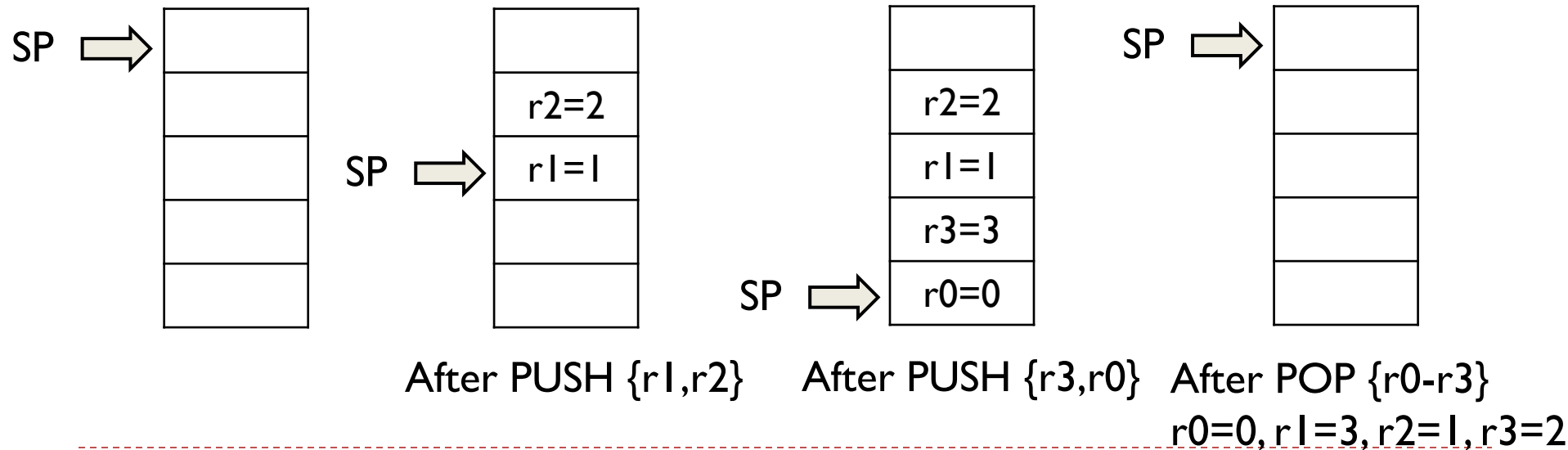
► Initially, let $r0=0, r1=1, r2=2, r3=3$

► Execute

PUSH {r1,r2} @ Equiv. to PUSH r2 \n PUSH r1

PUSH {r3,r0} @ Equiv. to PUSH r3 \n PUSH r0

POP {r0-r3} @ Equiv. to POP r0 \n POP r1 \n POP r2 \n POP r3



Stack

Before execution

R1=0x11111111

R2=0x22222222

PUSH {R1,R2}

POP {R1}

POP {R2}

R1 0x11111111

R2 0x22222222

R13 (SP) 0x20000200

Address

xxxxxxxx

0x20000200

xxxxxxxx

0x200001FC

xxxxxxxx

0x200001F8

memory

- ▶ What is content of stack, and position of SP, after PUSH {R2,R1}?
- ▶ What are the values of R1/R2 after POP {R2}?
- ▶ What are the correct instructions for swapping R1 and R2?

Stack ANS

Before execution

R1=0x11111111

R2=0x22222222

PUSH {R1,R2}

POP {R1}

POP {R2}

R1

0x11111111

R2

0x22222222

R13 (SP)

0x20000200

Address

xxxxxxxx

0x20000200

0x22222222

0x200001FC

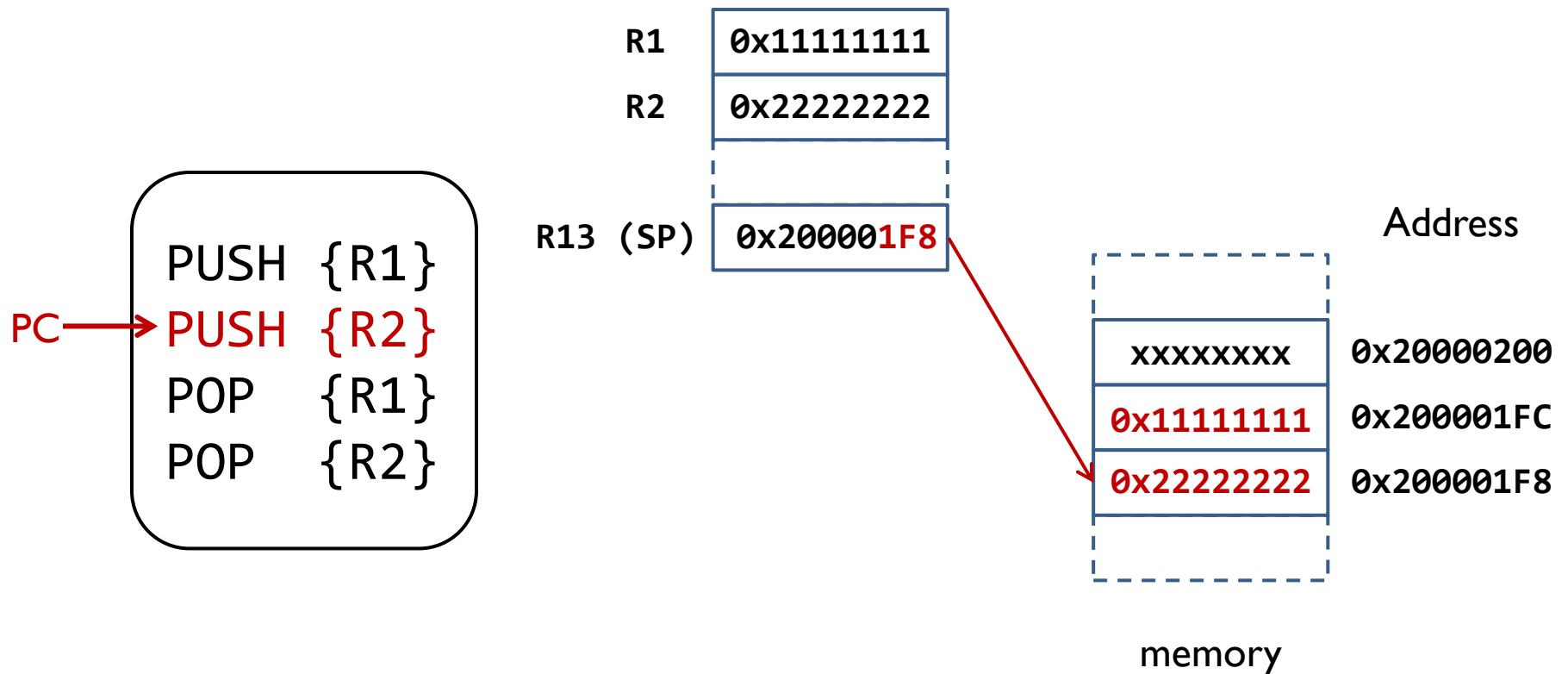
0x11111111

0x200001F8

memory

- ▶ Stack content and SP shown in figure
- ▶ After POP {R2}, R1=0x11111111, R2=0x22222222

Instructions for swapping R1 and R2



Program Understanding

- ▶ Compute register and memory values at each step of this program, given initial register values and memory contents, assuming little-endian memory ordering.

R0	0x00000000
R1	0x10000200
R2	0x0000FFFF
R3	0x18675309
R4	0x00000000
R5	0x00000000
...	
R13	0x10000200

PUSH (R1, R3)
POP (R5)

Initial Register Values

0x10000200	60	1B	11	12	EE	FF	11	22	33	44	55	66	77	88	99	92
0x100001F0	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F
0x100001E0	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F

Initial Memory Contents



Program Understanding ANS

- ▶ R13 is SP = 0x10000200. Recall:
- ▶ After two PUSHes, SP = 0x100001F8. Each PUSH is:
 - ▶ SUB SP, SP, #4 @ SP = SP - 4 (descending stack)
 - ▶ STR Rd, [SP] @ (*SP) = Rd (full stack)
- ▶ After one POP, SP = 0x100001FC:
 - ▶ LDR Rd, [SP] @ Rd = (*SP) (full stack)
 - ▶ ADD SP, #4 @ SP = SP + 4 (Stack shrinks)
- ▶ R5 = 0x10000200 (value of R1)

R0	0x00000000
R1	0x10000200
R2	0x0000FFFF
R3	0x18675309
R4	0x00000000
R5	0x10000200
...	
R13	0x10000200

PUSH (R1, R3)
POP (R5)

SP

Final Register Values

SP after 2 PUSHes

0x10000200	60	1B	11	12	EE	FF	11	22	33	44	55	66	77	88	99	92
0x100001F0	10	11	12	13	14	15	16	17	00	02	00	10	09	53	67	18
0x100001E0	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F

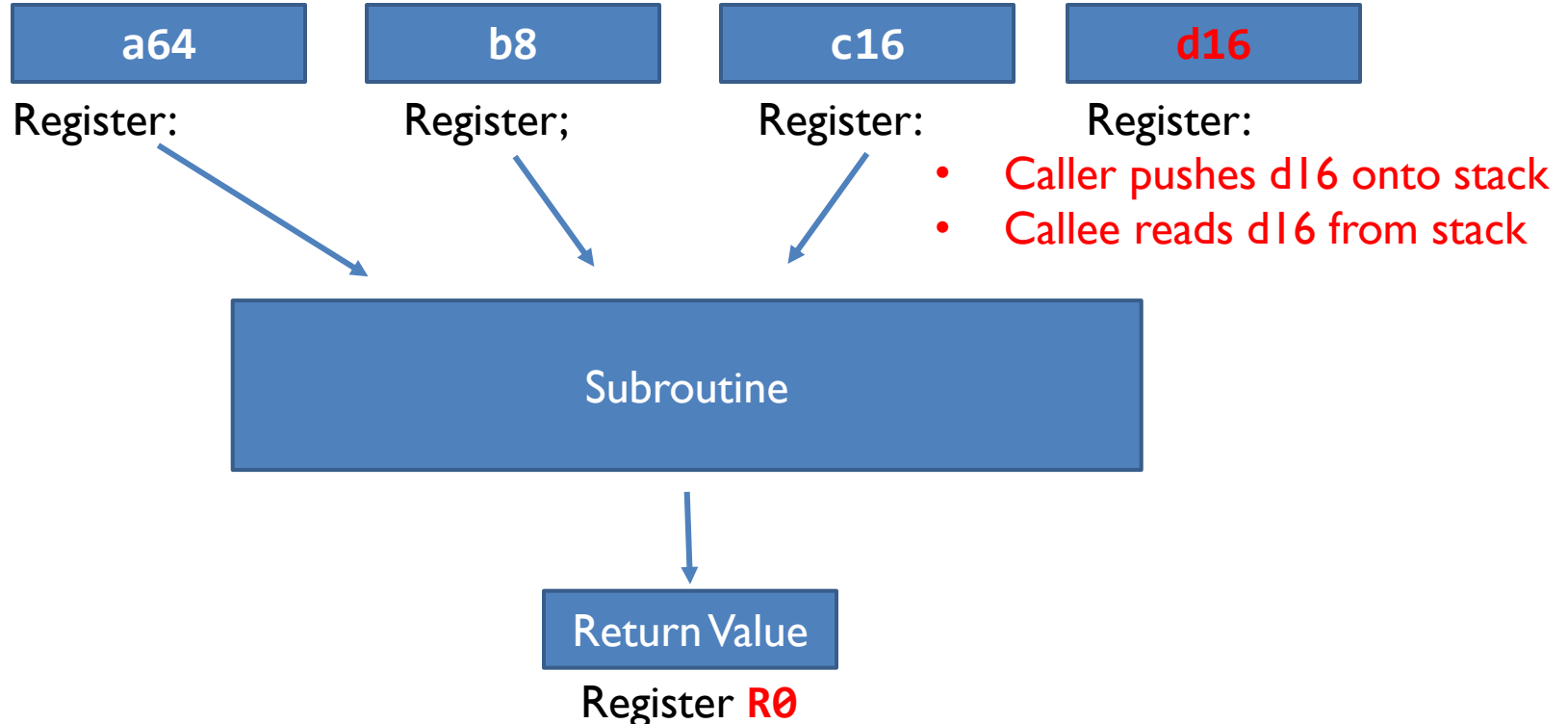
Initial Memory Contents

SP after 1 POP

Passing arguments and Returning Value

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

- Fill in register names.

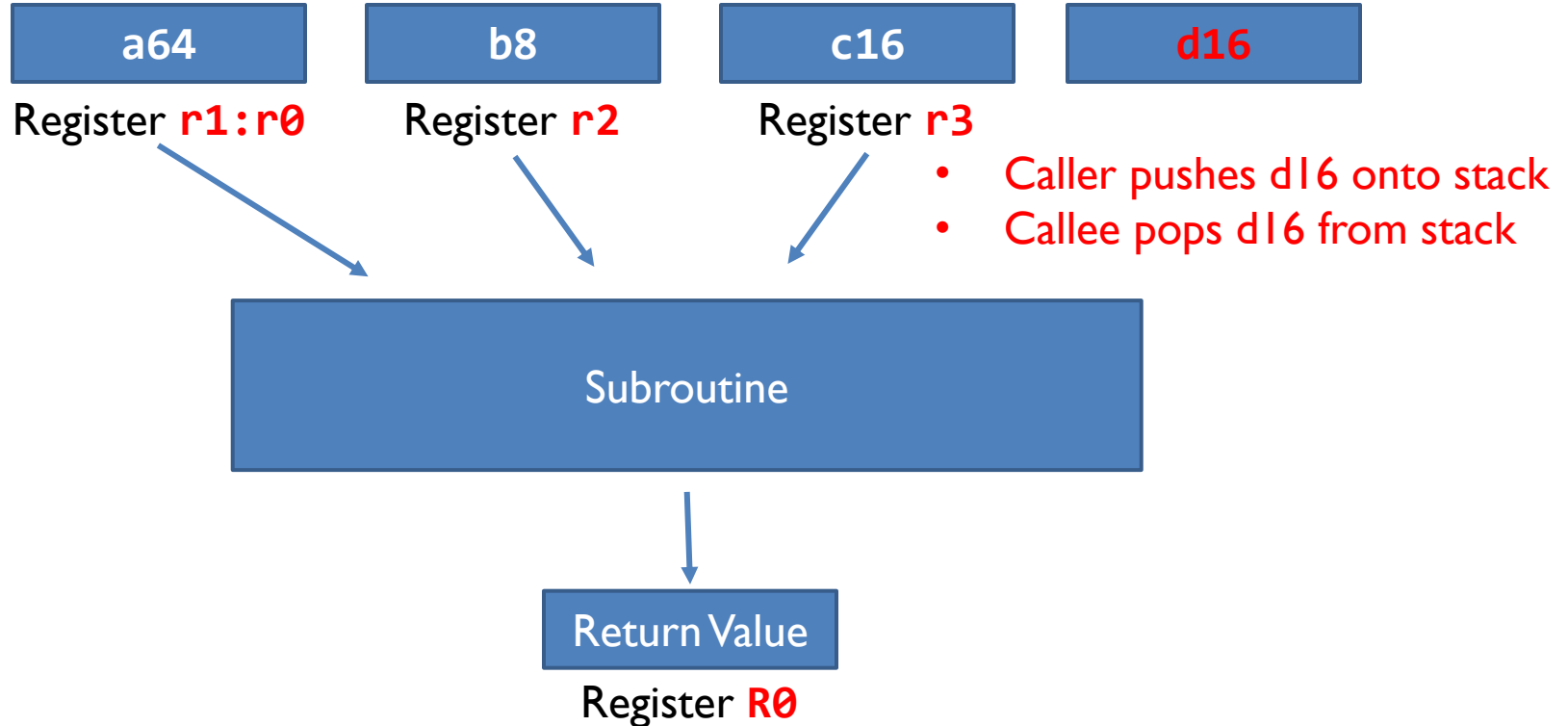


Passing arguments and Returning Value

ANS

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

- Fill in register names.



Argument Passing

- ▶ Which registers are used to pass the arguments and return the result?

long fun (short a1, char a2, double a3, int a4, char a5)

Registers

R0	R1	R2	R3
----	----	----	----

Stack in Memory

--

Argument Passing ANS

- ▶ Which registers are used to pass the arguments and return the result?

```
long fun (short a1, char a2, double a3, int a4, char a5)
```

Registers			Stack in Memory		
a1	a2	a3	a4	a5	

- ▶ Each argument of 8-bit char, or 16-bit short, is passed in one 32-bit register; cannot use 1 register to pass more than 1 arguments
 - ▶ a1 (short) in R0.
 - ▶ a2 (char) in R1.
 - ▶ a3 (double) in R2 and R3.
 - ▶ a4 (int) on the stack.
 - ▶ a5 (char) on the stack.

What's wrong? 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

```
MOV r0, #5 ; e32  
MOV r0, #1 ; a8  
MOV r1, #2 ; b8  
MOV r2, #3 ; c16  
MOV 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  
    ADD r0, r0, r1 ; add e32  
    BX LR  
ENDP
```

What's wrong? Passing arguments and Returning Value ANS

```
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

```
MOV r0, #5 ; e32  
PUSH {r0}  
MOV r0, #1 ; a8  
MOV r1, #2 ; b8  
MOV r2, #3 ; c16  
MOV 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.



What is Wrong?

Caller Program

```
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);
```

```
...
```

Callee Program

```
sum3 PROC
```

```
EXPORT sum3
```

```
; r3 = sum
```

```
ADD r3, r0, r1 ; sum = a1 + a2
```

```
ADD r3, r0, r2 ; sum += a3
```

```
MOV r1, r3
```

```
BX pc
```

```
ENDP
```



What is Wrong? ANS

- ▶ Return result should be put into r0, not r1
- ▶ BX lr returns to caller
- ▶ BX pc is not a return and will re-execute at the PC value

Caller Program

```
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);
    ...
}
```

Callee Program

```
sum3 PROC
EXPORT sum3
; r3 = sum
ADD r3, r0, r1 ; sum = a1 + a2
ADD r3, r0, r2 ; sum += a3
MOV r0, r3
BX lr
ENDP
```

Program Understanding

- ▶ Write out the sequence of values of r0 and r7 after running this program.

```
start:
    mov     r0, #1

main:
    add     r0, r0, #1
    cmp     r0, #5
    bne     skip
    bl      call

skip:
    b       main

call:
    add     r7, r7, #255
    mov     r0, #1
    bx      lr
```

Program Understanding ANS

- ▶ It starts with $r0 = 1$. It loops, incrementing $r0$ each time. When $r0$ reaches 5, it calls subroutine call, which adds 255 to $r7$ and resets $r0$ to 1. Control returns to main, and the cycle repeats. So the pattern goes like this:
- ▶ $r0: 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow (\text{call resets to } 1)$
- ▶ $r7: +255$ each time $r0$ reaches 5

```
start:
    mov     r0, #1        @ start: r0 = 1

main:
    add     r0, r0, #1    @ increment r0
    cmp     r0, #5        @ compare r0 to 5
    bne     skip          @ if not equal, go to skip
    bl      call           @ if equal, branch-and-link to call

skip:
    b       main          @ unconditional branch back to main

call:
    add     r7, r7, #255   @ add 255 to r7 (side-effect)
    mov     r0, #1        @ reset r0 to 1
    bx      lr            @ return (branch to link register)
```

toLower

Caller Program

```
#include <stdio.h>

extern int mystery(int); /* mystery assembler routine */

int main(void)
{
    static const char str[] = "Hello, World!";

    const int len = sizeof(str)/sizeof(str[0]);
    char      newstr[len];
    int       i;

    for (i = 0; i < len; i++)
        newstr[i] = toLower (str[i]);

    printf("%s\n", newstr);

    return 0;
}
```

- ▶ Consider the following C program that converts all ASCII letters to lower case. Write the toLower function in ARMv7 assembly code.

Callee Program

```
int toLower (int c)
{
    if (c >= 'A' && c <= 'Z')
        c += 32; /* 'a' - 'A' = 32 */

    return c;
}
```

Callee Program Assembly

```
.text
.global toLower
toLower:
```



toLower ANS

Callee Program

```
int toLower (int c)
{
    if (c >= 'A' && c <= 'Z')
        c += 32;

    return c;
}
```

Callee Program Assembly

```
.text
.global toLower
toLower:
    cmp     r0, #'A'           @ if c < 'A' -> skip_adjust
    blt     skip_adjust
    cmp     r0, #'Z'           @ if c > 'Z' -> skip_adjust
    bgt     skip_adjust
    add     r0, r0, #32         @ c += 32 = 'a' - 'A'

skip_adjust:
    bx      lr
```


Separator Page

- ▶ The following slides will not be covered in the exam

If Then Else

- Translate the following program into ARMv7 assembly.

C Program	Assembly Program
<pre>int foo(int x, int y) { if (x+y < 0) return 0; else return 1; }</pre>	<pre>@ int foo(int x, int y) - returns 0 if (x+y) < 0, else 1 @ x in r0, y in r1, return in r0 foo: ... BX lr</pre>

If Then Else ANS

Straight-line with conditional execution

```
foo:
    ADD    r2,r0,r1    ; r2 = x + y
    CMP    r2,#0       ; sets N,Z,V,C for signed
                        ; compare to 0
    MOVLT  r0,#0       ; if (x+y) < 0 -> r0 = 0
    MOVGE  r0,#1       ; else r0 = 1
    BX     lr
```

Conditional branch to label w/ CMP

```
foo:
    ADD    r2,r0,r1
    CMP    r2,#0
    BLT    .Lneg
    MOV    r0,#1
    BX     lr
.Lneg:
    MOV    r0,#0
    BX     lr
```

Combine add and compare with ADDS

```
foo:
    ADDS   r2,r0,r1    ; r2 = x + y, sets flags from
                        ; the sum
    MOVMI  r0,#0       ; MI means N=1 (negative)
    MOVPL  r0,#1       ; PL means N=0 (non-
                        ; negative)
    BX     lr
```

No temp register r2, reuse r0

```
foo:
    ADDS   r0,r0,r1    ; r0 = x + y, flags set
    MOVMI  r0,#0
    MOVPL  r0,#1
    BX     lr
```

Conditional branch to label w/ ADDS

```
foo:
    ADDS   r2,r0,r1
    BMI    .Lneg       ; if negative
    MOV    r0,#1
    BX     lr
.Lneg:
    MOV    r0,#0
    BX     lr
```



Factorial

- Fill in the blanks (marked TODO) for the assembly programs for calculating the factorial of a number, corresponding to the following C programs. One iterative version, one recursive version.

```
//Iterative version
#include <stdint.h>

uint32_t fact_iter(uint32_t n) {
    uint32_t acc = 1;
    if (n <= 1) {
        return 1;
    }
    while (n > 1) {
        acc *= n;
        n -= 1;
    }
    return acc;
}
```

```
//Recursive version
#include <stdint.h>

uint32_t fact_rec(uint32_t n) {
    if (n <= 1) {
        return 1;
    }
    return n * fact_rec(n - 1);
}
```

Factorial

% uint32_t fact_iter(uint32_t n); iterative version
% r0 = n, returns r0 = n!

.global fact_iter

fact_iter:

PUSH {r4, lr} % save callee-saved we'll
use and return addr

MOV r1, r0 % r1 = n (loop counter)

MOV r0, #1 % r0 = acc = 1

CMP r1, #1

BLS .Ldone_iter % if n <= 1, return 1
%(unsigned compare)

.Lloop_iter:

% TODO

.Ldone_iter:

POP {r4, lr}

BX lr

% uint32_t factorial(uint32_t n); Recursive version

% r0: n

% returns r0: n!

factorial:

CMP r0, #1 % if (n <= 1) ...

BLS base_case % ... return 1

PUSH {lr} % save return address for this frame

PUSH {r0} % save current n on stack (we'll need it after the
recursive call)

SUB r0, r0, #1 % r0 = n - 1 (argument for recursive call)

BL factorial % r0 = factorial(n - 1)

POP {r1} % r1 = saved n (restore caller's n)

MUL r0, r0, r1 % r0 = factorial(n - 1) * n

POP {lr} % restore return address

BX lr % return with result in r0

base_case:

% TODO



Factorial ANS

```
% uint32_t fact_iter(uint32_t n); iterative version
% r0 = n, returns r0 = n!
```

```
.global fact_iter
```

```
fact_iter:
```

```
    PUSH    {r4, lr}      % save callee-saved we'll
use and return addr
```

```
    MOV     r1, r0        % r1 = n (loop counter)
```

```
    MOV     r0, #1        % r0 = acc = 1
```

```
    CMP     r1, #1
```

```
    BLS     .Ldone_iter   % if n <= 1, return 1
                        % (unsigned compare)
```

```
.Lloop_iter:
```

```
    MUL     r0, r0, r1    % acc *= i
```

```
    SUBS    r1, r1, #1    % i--
```

```
    BHI     .Lloop_iter   % continue while i > 1
                        % (unsigned compare)
```

```
.Ldone_iter:
```

```
    POP     {r4, lr}
```

```
    BX      lr
```

POP is used here because the function previously PUSHed registers that must be **restored** before returning; POP both restores those registers and fixes the stack pointer in one instruction.

```
% uint32_t factorial(uint32_t n); Recursive version
```

```
% r0: n
```

```
% returns r0: n!
```

```
factorial:
```

```
    CMP     r0, #1        % if (n <= 1) ...
```

```
    BLS     base_case     % ... return 1
```

```
    PUSH    {lr}          % save return address for this frame
```

```
    PUSH    {r0}          % save current n on stack (we'll need it after the
recursive call)
```

```
    SUB     r0, r0, #1    % r0 = n - 1 (argument for recursive call)
```

```
    BL      factorial     % r0 = factorial(n - 1)
```

```
    POP     {r1}          % r1 = saved n (restore caller's n)
```

```
    MUL     r0, r0, r1    % r0 = factorial(n - 1) * n
```

```
    POP     {lr}          % restore return address
```

```
    BX      lr            % return with result in r0
```

```
base_case:
```

```
    MOV     r0, #1        % factorial(n) = 1 for n <= 1
```

```
    BX      lr            % return
```

What is wrong?

```
int16_t sum_of_array(int16_t *pArray){
    uint32_t i;
    int32_t sum = 0;
    for(i=0; i<64; i++) // array size = 64
        sum += pArray[i];
    return (int16_t) sum;
}
```

```
sum_of_array PROC
    MOV     r2, #0    ; loop index
    MOV     r3, #0    ; sum
    B       check
loop  LDRSH  r1, [r0, r2, LSL #1]
    ADD     r3, r3, r1 ; sum += pArray[i]
    ADD     r2, r2, #1 ; i++
check CMP    r2, #64
    BLO     loop      ; branch if unsigned LOwer
    MOV     r0, r3    ; return result in r0
    BX      lr
ENDP
```

What is wrong? ANS

```
int16_t sum_of_array(int16_t *pArray){
    uint32_t i;
    int32_t sum = 0;
    for(i=0; i<64; i++) // array size = 64
        sum += pArray[i];
    return (int16_t) sum;
}
```

```
int32_t x = 0x12345678;
int16_t y = (int16_t) x;
```

0x12345678 → 0x00005678

0x1234ABCD → 0xFFFFABCD

```
sum_of_array PROC
    MOV    r2, #0    ; loop index
    MOV    r3, #0    ; sum
    B      check
loop  LDRSH r1, [r0, r2, LSL #1]
    ADD    r3, r3, r1 ; sum += pArray[i]
    ADD    r2, r2, #1 ; i++
check CMP    r2, #64
    BLO    loop      ; branch if unsigned LOwer
    MOV    r0, r3, LSL #16
    MOV    r0, r0, ASR #16 ; r0 = (short) r3
    BX     lr
ENDP
```

- ▶ C function returns `int16_t`, but the assembly code puts the return in `r0` as a 32-bit word, so that 16-bit value in `r3` must be sign-extended to 32 bits before returning
 - ▶ While `r3` is 32 bits, the meaningful 16-bit result is in `r3[15:0]`
 - ▶ `r3[31:16]` may contain carry-over bits and must be replaced by copies of the sign bit before returning.
 - ▶ Casting an `int32_t` to `int16_t` keeps only the low 16 bits and produces 32-bit sign-extended result in `r0` (e.g. with `SXTH`)

Explanations

- ▶ These two lines sign-extend the lower 16 bits of r3 into a full 32-bit value in r0.
- ▶ `MOV r0, r3, LSL #16`
 - ▶ Shifts the value in r3 left by 16 bits.
 - ▶ The lower 16 bits of r3 (the part we care about) move up into the upper 16 bits of r0.
 - ▶ The bottom 16 bits of r0 become 0.
- ▶ `MOV r0, r0, ASR #16`
 - ▶ Shifts the value right by 16 bits, but preserves the sign (arithmetic shift).
 - ▶ That means if bit 31 (the sign bit after the first shift) is 1, it fills with 1s; if 0, it fills with 0s.
 - ▶ The result is a sign-extended 16-bit value from the original lower half of r3.
- ▶ Examples:
- ▶ `r3 = 0x00001234`
 - ▶ `LSL #16` → `0x12340000`
 - ▶ `ASR #16` → `0x00001234`
- ▶ `r3 = 0x0000ABCD`
 - ▶ `LSL #16` → `0xABCD0000`
 - ▶ `ASR #16` → `0xFFFFABCD`
- ▶ Can be replaced with a single instruction:
 - ▶ `SXTH r0, r3 ; r0 = (int16_t) r3, sign-extend`
 - ▶ Take the low 16 bits of r3, sign-extend them to 32 bits, and store the result in r0.

Program Understanding

- Given the following code, fill in the blanks listed next to each instruction with the values that would be in each register or status bit **after each instruction executes**. Instruction addresses are given to the left of each assembly instruction (not part of the code). Assume that $SP = 0x10000200$ at the start of the program.

```
        AREA mycode, CODE, READONLY
        EXPORT __main

__main  PROC
0x250   MOV R0, #5
0x254   MOV R1, #10
0x258   CMP R0, R1           ; NZCV = _____
0x25A   BGE loop             ; PC = _____(after this instruction)
0x25C   BL foo               ; LR = _____, PC = _____(after this inst.)
0x260   loop B loop
        ENDP

foo     PROC
0x262   PUSH{R0, R1}         ; SP = _____
0x264   POP{R1}
0x266   POP{R0}              ; R0 = _____, SP = _____
0x268   BX LR                ; LR = _____, PC = _____
        ENDP
```

Program Understanding ANS

- Given the following code, fill in the blanks listed next to each instruction with the values that would be in each register or status bit after the instruction executes. Instruction addresses are given to the left of each assembly instruction (not part of the code). Assume that $SP = 0x10000200$ at the start of the program.

```
AREA mycode, CODE, READONLY
EXPORT __main
__main PROC
0x250    MOV R0, #5
0x254    MOV R1, #10
0x258    CMP R0, R1                ; NZCV = 1000
0x25A    BGE loop                  ; PC = 0x25C (after this instruction)
0x25C    BL foo                    ; LR = 0x260, PC = 0x262 (after this inst.)
0x260    loop B loop
        ENDP

foo      PROC
0x262    PUSH{R0, R1}              ; SP = 0x100001F8
0x264    POP{R1}                   ; R1 = 5
0x266    POP{R0}                   ; R0 = 10, SP = 0x10000200
0x268    BX LR                     ; LR = 0x260, PC = 0x260 (after this inst.)
        ENDP
```



Detailed Explanations

- ▶ 1. CMP R0, R1 (0x858/0x258)
 - ▶ Performs: $8 - 10$
 - ▶ NZCV = 1000 (Negative=1, Zero=0, Carry=0, Overflow=0)
- ▶ 2. BGE loop (0x25A)
 - ▶ BGE checks if $N == V$ (Greater or Equal in signed comparison)
 - ▶ Since $N=1$ and $V=0$, condition is FALSE
- ▶ Branch NOT taken, continues to next instruction
 - ▶ PC = 0x25C
- ▶ 3. BL foo (0x25C)
 - ▶ Saves return address in LR and branches to foo
 - ▶ LR = 0x260 (next instruction after BL)
 - ▶ PC = 0x262 (address of foo)
- ▶ 4. PUSH {R0, R1} (0x262)
 - ▶ Stack is descending and full, so SP decrements by 8 bytes (2 registers \times 4 bytes)
 - ▶ $SP = 0x10000200 - 8 = 0x100001F8$
- ▶ 5. POP {R0} (0x266)
 - ▶ After popping both registers, R0 and R1 are swapped
 - ▶ R0 = 10 (original R1 value)
 - ▶ $SP = 0x10000200$ (back to original)
- ▶ 6. BX LR (0x268)
 - ▶ Returns to address stored in LR
 - ▶ LR = 0x260 (unchanged)
 - ▶ PC = 0x260 (returns to caller)

