CSC111
Final Review

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/

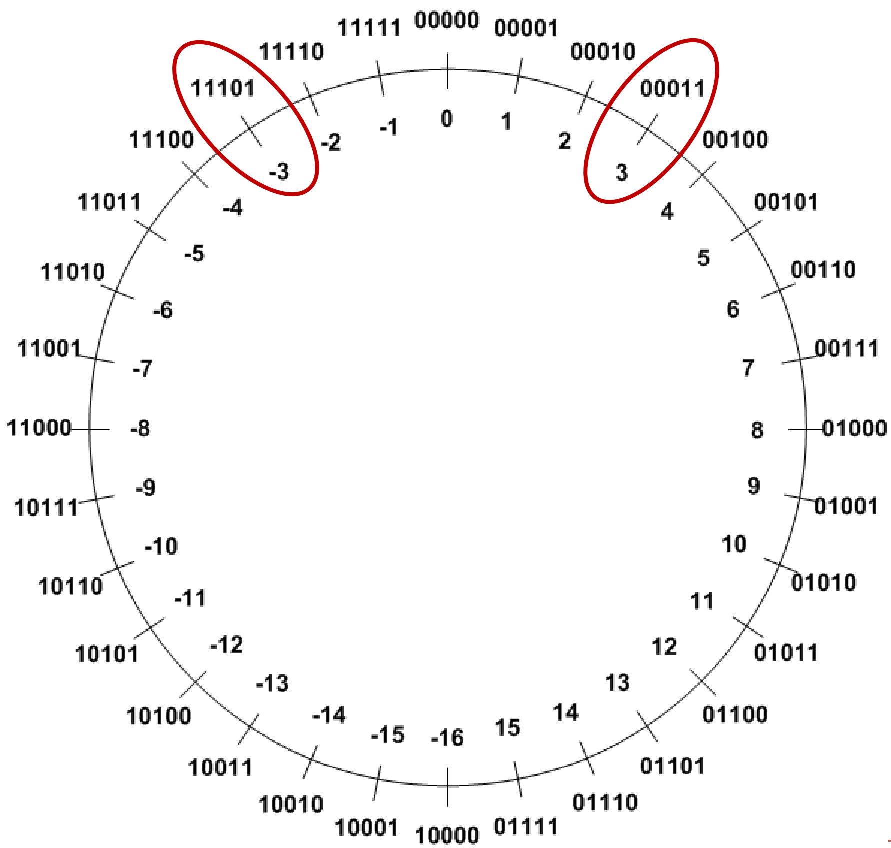
Data Representation

- Integers
  - Binary in different bases: binary, octal, hex, decimal
  - Signed Integers
    - Signed magnitude
    - One's complement
    - Two's complement
  - Arithmetic Operations
    - N,V, C, Z flags
  - Big Endian vs Little Endian
- ASCII values
  - Null-terminated string
  - Converting between numbers and ASCII
  - Upper case, lower case

Signed Integers
Method 3: Two's Complement

Two's Complement ($\overline{\alpha}$):

$$\alpha + \overline{\alpha} = 2^n$$

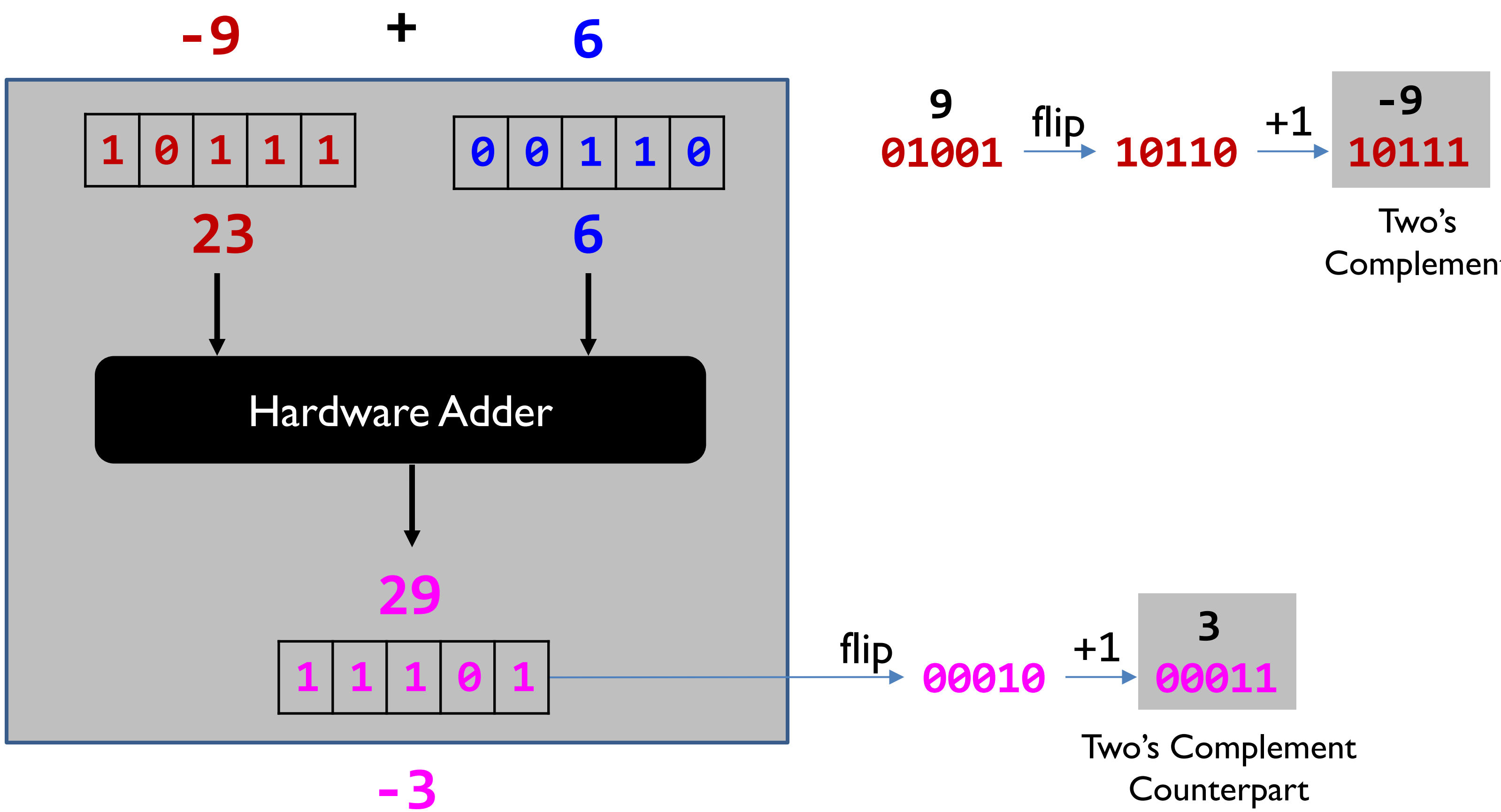


TC of a number can be obtained by its bitwise NOT plus one.

Example **1**: TC(3)

	Binary	Decimal
Original number	00011	3
Step 1: Invert every bit	11100	
Step 2: Add 1	+ 00001	
Two's complement	11101	-3

Adding two integers



- Same bit patterns, different interpretation.
  - Unsigned addition: 23+6=29
  - Signed addition: -9+6=-3
- This example shows that the hardware adder for adding unsigned numbers, also works correctly for adding signed numbers.

Basic Assembly Programming

- Load-modify-store sequence
- Accessing memory
  - Memory addressing mode
    - Pre-index
    - Post-index
    - Pre-index with update
- Data processing
  - Arithmetic, Logic, Comparison, Data Movement
  - Barrel Shifter:
    - ADD r1, r0, r0, LSL 2
  - Bit operations
    - Set a bit, Reset a bit, Toggle a bit, Check a bit
  - LSL, LSR, ASR, ROR, RRX
- Flow control: if, if-then-else, for loop, while loop
  - Unconditional Branch: B
  - Conditional Branch: CMP, TEQ, TST, BEQ, BNE, BMI, BLS, BHI, etc.
  - Conditional Execution: MOVEQ, MOVNE

Barrel Shifter: Explanations

- LSL (logical shift left): shifts left, fills zeros on the right; C gets the last bit shifted out of bit 31. This is multiply by 2^n for non-overflowing values.
- LSR (logical shift right): shifts right, fills zeros on the left; C gets the last bit shifted out of bit 0. This is unsigned division by 2^n.
- ASR (arithmetic shift right): shifts right, fills the sign bit on the left to preserving the sign; C gets the last bit shifted out of bit 0. This is signed division by 2^n with sign extension
- ROR (rotate right): rotates bits right with wraparound; bits leaving bit 0 re-enter at bit 31, and C receives the bit that wrapped. This is a pure rotation without data loss.
- RRX (rotate right extended): rotates right by one through the carry flag, treating C as a 33rd bit; new bit 31 comes from old C, and C receives old bit 0.

Load/Store a Byte, Halfword, Word

```
LDRxxx R0, [R1]
; Load data from memory into a 32-bit register
```

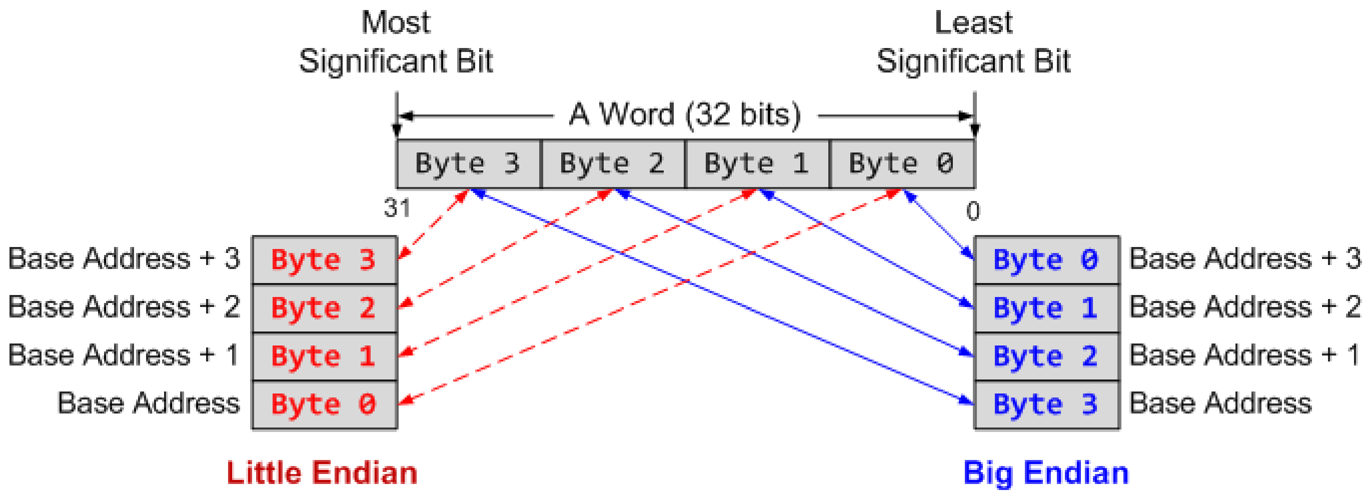
LDR	Load Word	uint32_t/int32_t	unsigned or signed int
LDRB	Load **B**yte	uint8_t	unsigned char
LDRH	Load **H**alfword	uint16_t	unsigned short int
LDRSB	Load **S**igned **B**yte	int8_t	signed char
LDRSH	Load **S**igned **H**alfword	int16_t	signed short int

```
STRxxx R0, [R1]
; Store data extracted from a 32-bit register into memory
```

STR	Store Word	uint32_t/int32_t	unsigned or signed int
STRB	Store Lower **B**yte	uint8_t/int8_t	unsigned or signed char
STRH	Store Lower **H**alfword	uint16_t/int16_t	unsigned or signed short

ARM Load Store Summary

- Memory address is always in terms of bytes.
- How data is organized in memory?



- How data is addressed?

Addressing Format	Example	Equivalent
Pre-index	`LDR r1, [r0, #4]`	r1 ← memory[r0 + 4], r0 is unchanged
Pre-index with update	`LDR r1, [r0, #4]!`	r1 ← memory[r0 + 4] r0 ← r0 + 4
Post-Index	`LDR r1, [r0], #4`	r1 ← memory[r0] r0 ← r0 + 4

Character String

char str[13] = “ARM Assembly”;

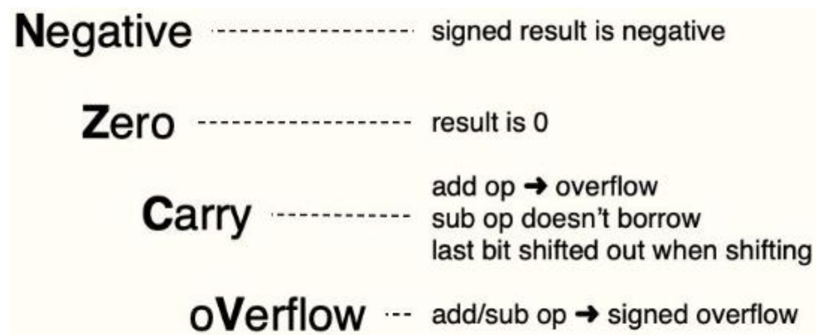
Memory Address	Memory Content	Letter
str + 12→	0x00	\0
str + 11→	0x79	y
str + 10→	0x6C	l
str + 9→	0x62	b
str + 8→	0x6D	m
str + 7→	0x65	e
str + 6→	0x73	s
str + 5→	0x73	s
str + 4→	0x41	A
str + 3→	0x20	space
str + 2→	0x4D	M
str + 1→	0x52	R
str →	0x41	A

This diagram does not indicate big-endian or little-endian. Endianness is irrelevant for single-byte char arrays.
If you want to detect endianness, you must inspect a **multi-byte** value in memory, e.g.:
int x = 0x12345678;
That will reveal the byte order.

Condition Flags

Program Status Register (PSR)

N	Z	C	V	Q	ICI/IT	T	Reserved	GE	Reserved	ICI/IT		ISR number



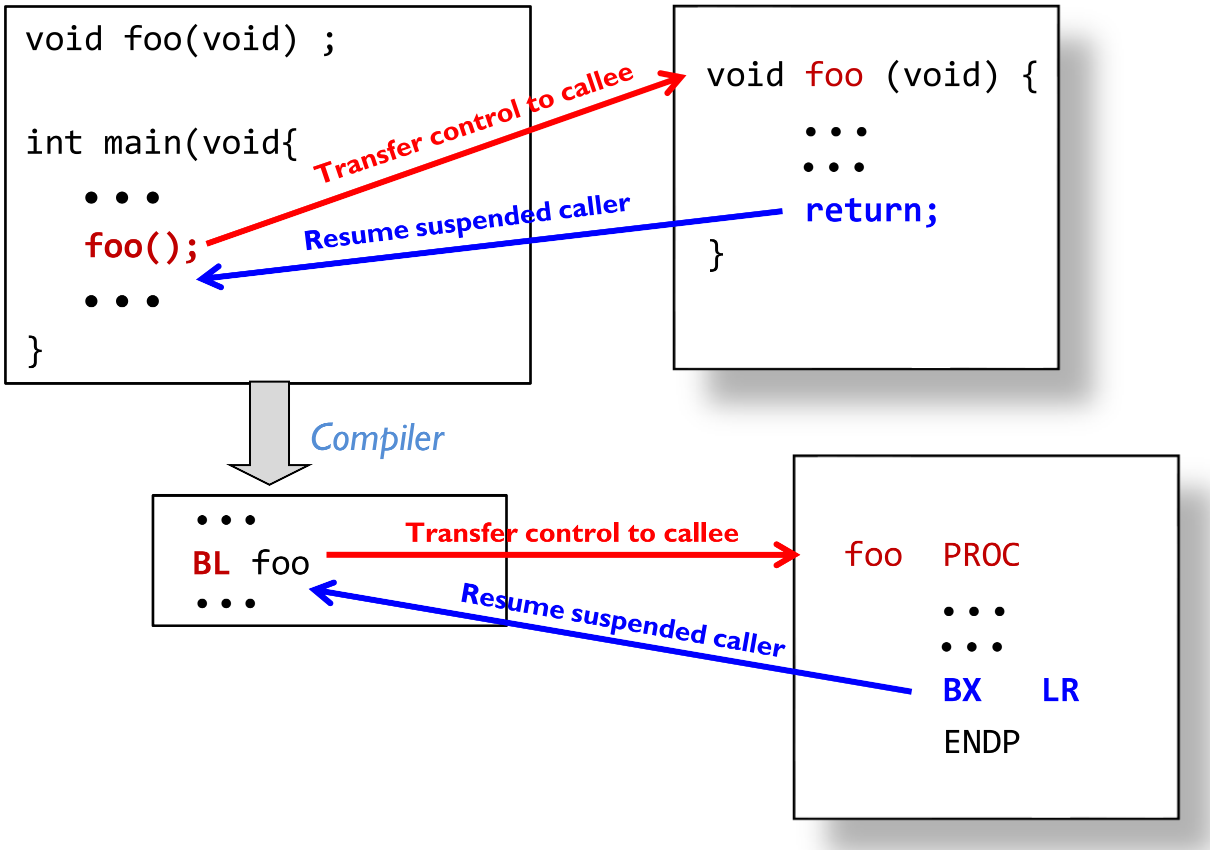
- **Negative** bit
  - `N = 1` if most significant bit of result is 1
- **Zero** bit
  - `Z = 1` if all bits of result are `0`
- **Carry** bit
  - For unsigned addition, `C = 1` if carry takes place
  - For unsigned subtraction, `C = 0` (carry = not borrow) if borrow takes place
  - For shift/rotation, C = last bit shifted out
- **oVerflow** bit
  - `V = 1` if adding 2 same-signed numbers produces a result with the opposite sign
    - Positive + Positive = Negative, or
    - Negative + negative = Positive
  - Non-arithmetic operations does not touch V bit, such as `MOV`, `AND`, `LSL`, `MUL`

Carry and Overflow Flags w/ Arithmetic Instructions

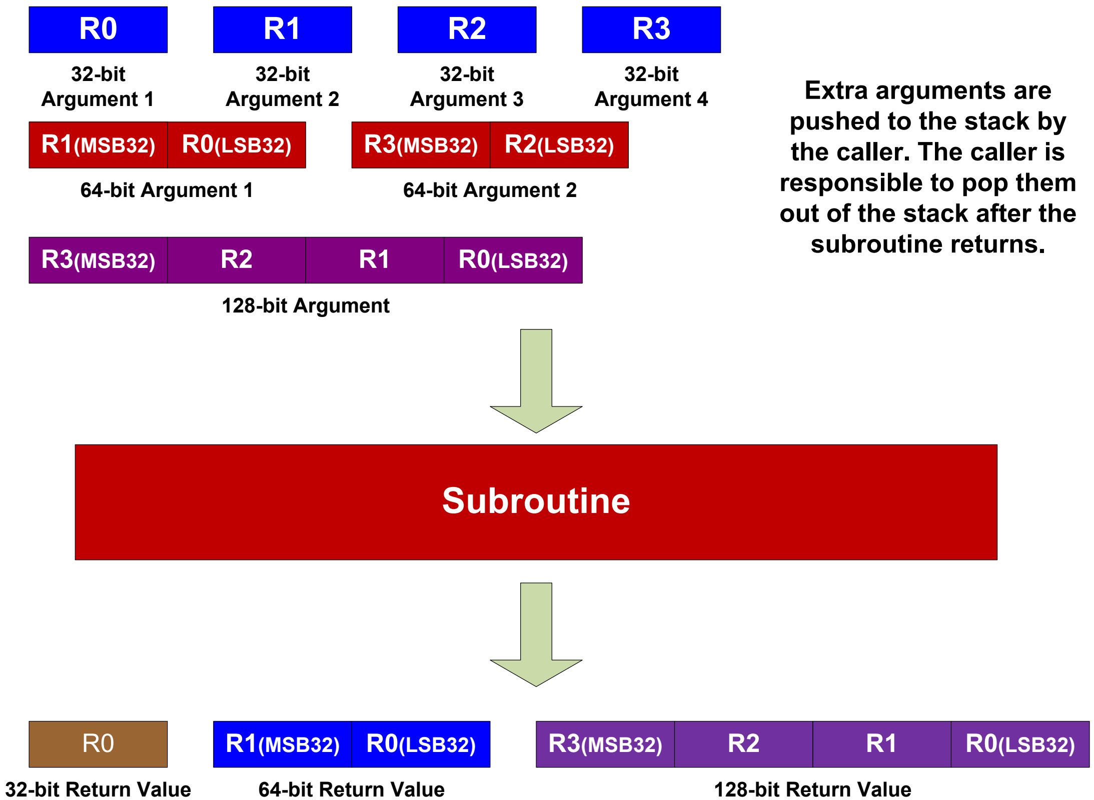
Carry flag C = 1 (Borrow flag = 0) upon an **unsigned** addition if the answer is wrong (true result > 2^n-1)
Carry flag C = 0 (Borrow flag = 1) upon an **unsigned** subtraction if the answer is wrong (true result < 0)
Overflow flag V =1 upon a **signed** addition or subtraction if the answer is wrong (true result > 2^{n-1}-1 or true result < -2^{n-1})
Overflow may occur when adding 2 operands with the same sign, or subtracting 2 operands with different signs; Overflow cannot occur when adding 2 operands with different signs or when subtracting 2 operands with the same sign.
Tip: Convert subtraction to addition with Two's complement. If two operands have same sign, and the result has opposite sign, then V = 1; else V = 0

	Unsigned Addition	Unsigned Subtraction	Signed Addition or Subtraction
Carry flag	true result > 2^n-1 ➔ Carry flag=1 Borrow flag=0 (Result incorrect)	true result < 0 ➔ Carry flag=0 Borrow flag=1 (Result incorrect)	N/A
Overflow flag	N/A	N/A	true result > 2^{n-1}-1 or true result < -2^{n-1} ➔ Overflow flag=1 (Result incorrect)

Link Register (LR)

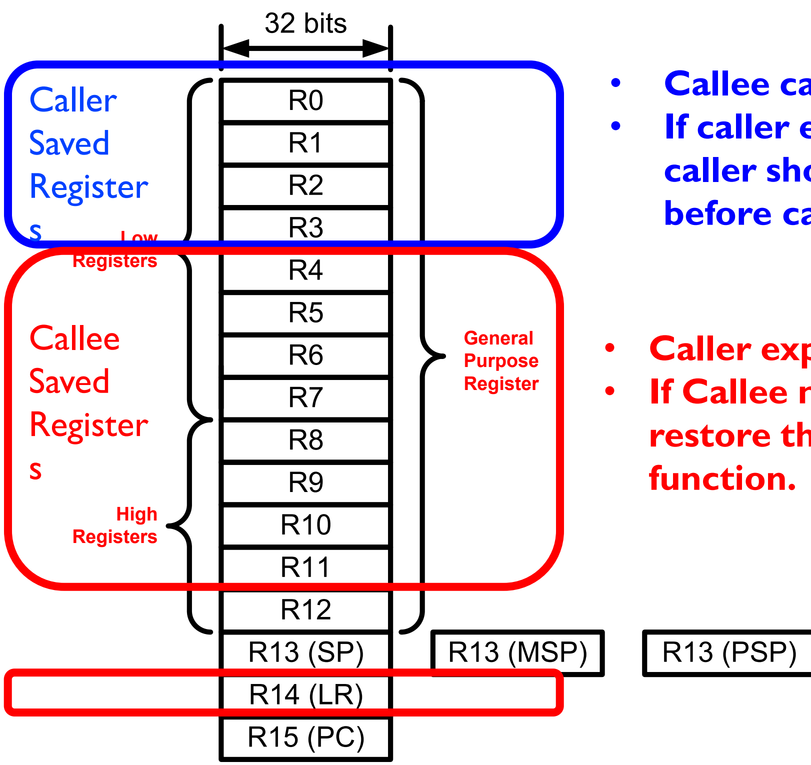


Passing Arguments and Returning Value



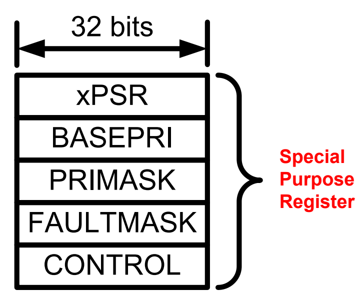
- Each argument with size ≤ 32 bits, e.g., 8-bit char, or 16-bit short, or 32-bit int, is passed in a 32-bit register.
  - Cannot pack multiple arguments into one register.
- The subroutine can take arguments larger than 32 bits. For example, a double-word variable, such as 64-bit long, is passed in 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

Callee Saved Registers *vs* Caller Saved Registers

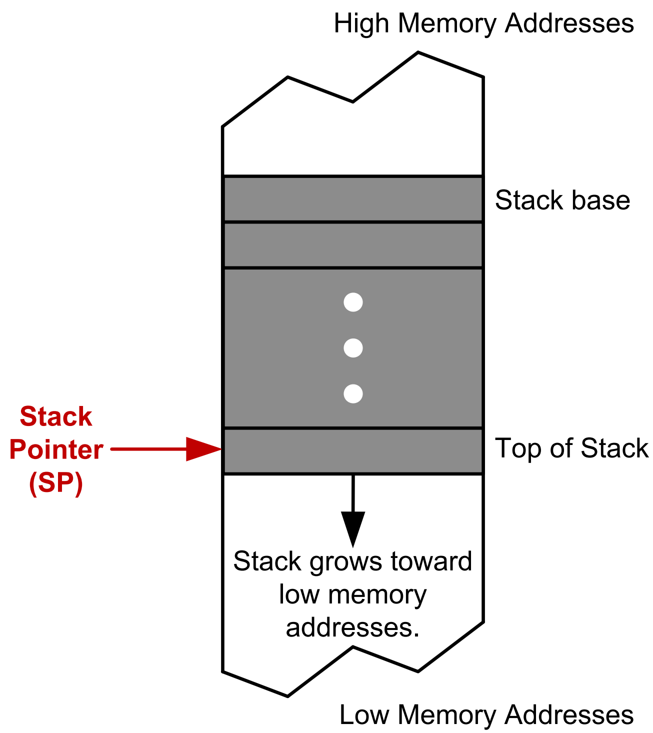


- **Callee can freely modify R0, R1, R2, and R3**
- **If caller expects their values are retained, caller should push them onto the stack before calling the callee**

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



Full Descending Stack



PUSH {register_list}
equivalent to:
STMDB SP!, {register_list}

DB: Decrement Before

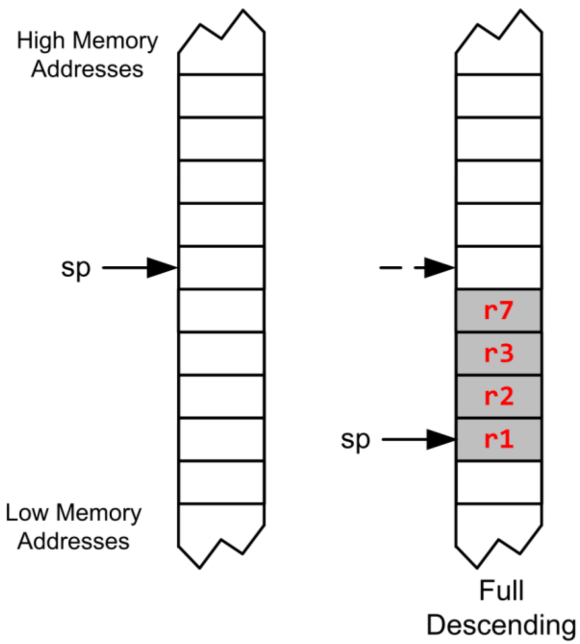
POP {register_list}
equivalent to:
LDMIA SP!, {register_list}

IA: Increment After

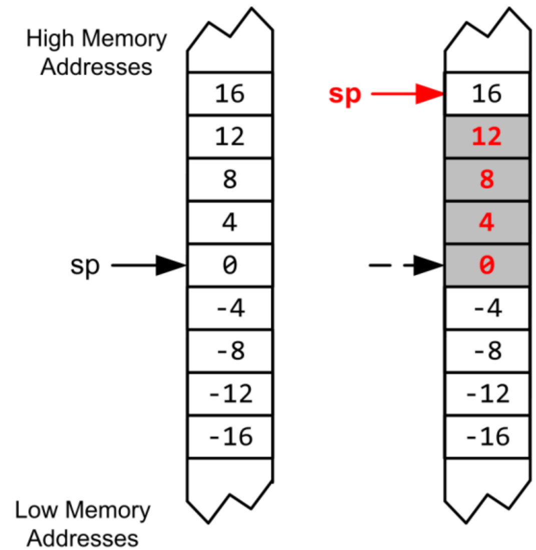
Stack Recap

Largest-numbered register is pushed first but popped last.

`PUSH {r3, r1, r7, r2}`



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




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

Pop to the smallest-numbered register first.

Calling a Subroutine

Caller: **BL** ***label*** (Branch and Link)

- Step 1: `LR = PC + 4`
- Step 2: `PC = label`
  - *label* is name of subroutine
  - Compiler translates label to memory address
  - After call, LR holds return address (the instruction following the call)

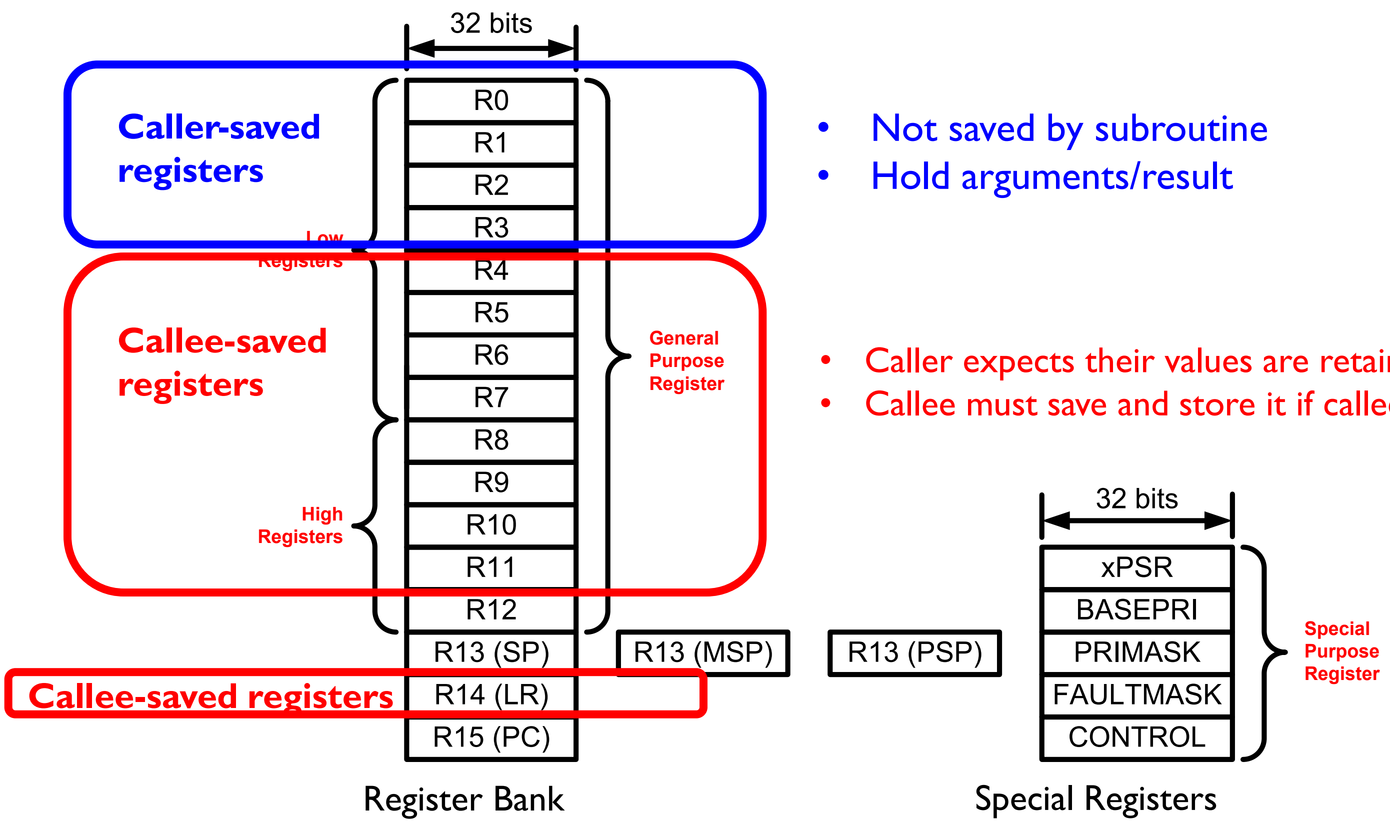
Callee: **BX LR** (Branch and Exchange) at end of procedure

- `PC = LR`
  - Return to caller by setting PC to LR
- Equivalently:
  - PUSH {LR} at start of procedure
  - POP {PC} at end of procedure

Caller Program	Subroutine/Callee	

`...`
`BL foo`
`...`
	`foo PROC`
	`...`
	`BX LR`
	`EDP`
		`foo PROC`
		`PUSH {LR}`
		`...`
		`POP {PC} ; pops LR into PC (returns)`
		`EDP`

Caller-saved Registers *vs* Callee-saved Registers



- Not saved by subroutine
- Hold arguments/result

- Caller expects their values are retained
- Callee must save and store it if callee modifies it

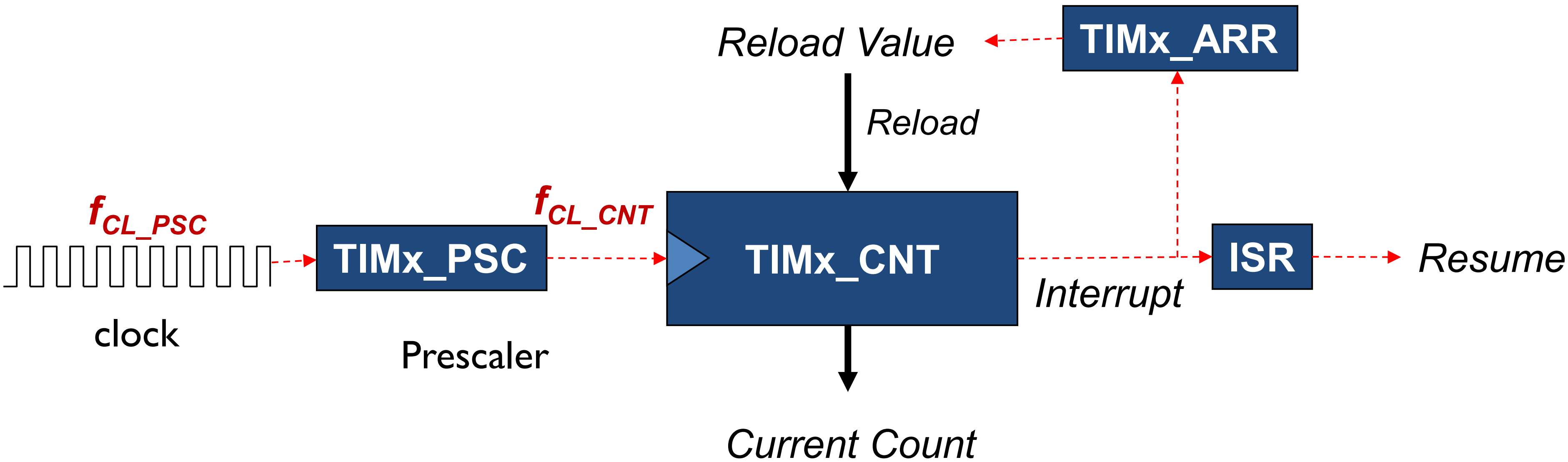
Chapter 8 Subroutines Summary I

- 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?
  - On the stack

Chapter 8 Subroutines Summary II

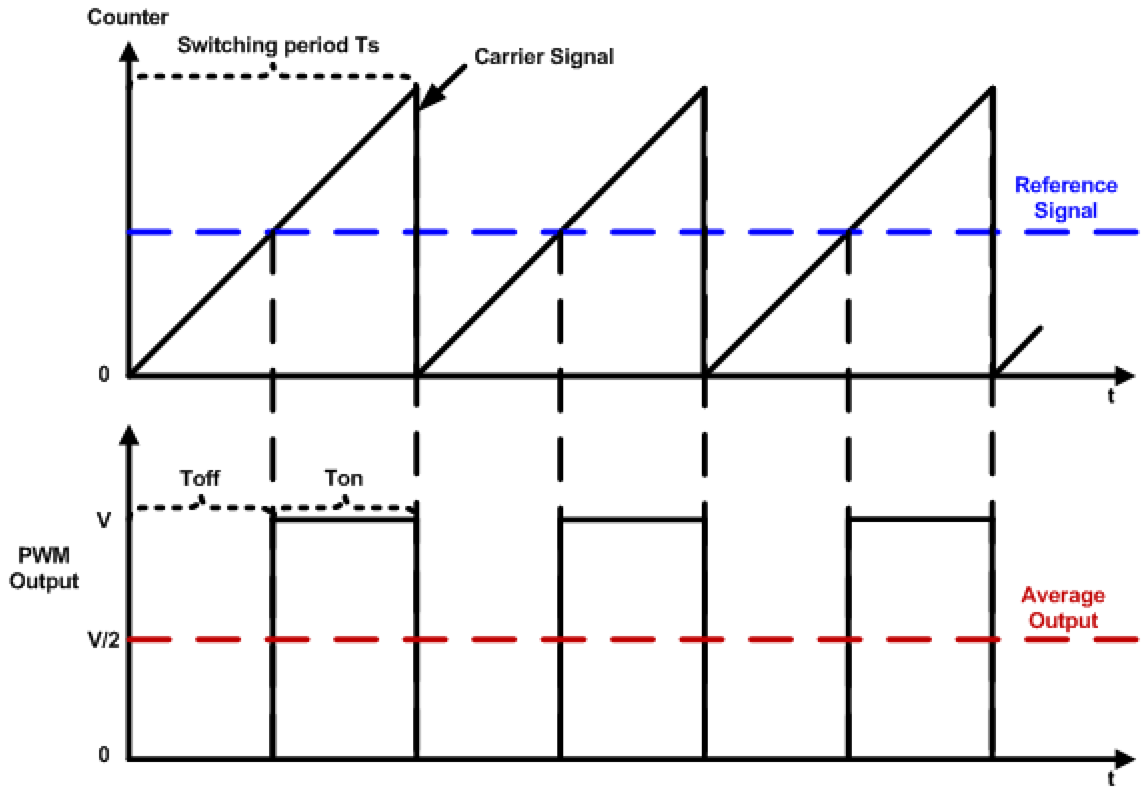
- ARM Cortex-M uses full descending stack
- How to pass arguments into a subroutine?
  - Each 8-, 16- or 32-bit parameter is passed via `r0`, `r1`, `r2`, `r3`
  - Extra parameters are passed via the stack
- What registers should be preserved?
  - Caller-saved registers *vs* callee-saved registers
- How to preserve the running environment for the caller?
  - Via stack

Timer's Clock



$$f_{CK\_CNT} = \frac{f_{CL\_PSC}}{Prescaler + 1}$$

PWM Duty Cycle = Ton/Time Period



$$duty\ cycle = \frac{pulse\ on\ time}{pulse\ switching\ period} \times 100\% = \frac{T_{on}}{T_s} \times 100\%$$

Summary of Equations

- Timer clock frequency f_{CK_CNT} vs. CPU Clock Frequency f_{SOURCE} (f_{CL_PSC})

$$f_{CK\_CNT} = \frac{f_{SOURCE}}{PSC + 1}$$

- Timer interrupt frequency f_{Timer} with up-counting or down-counting mode:

$$f_{Timer} = \frac{f_{CK\_CNT}}{ARR + 1}; \ Timer\ Period = \frac{ARR + 1}{f_{CK\_CNT}} = (ARR + 1) * Clock\ Period$$

- Timer interrupt frequency f_{Timer} with center-aligned counting mode: $f_{Timer} = \frac{f_{CK\_CNT}}{2 * ARR}; Timer\ Period = (2 * ARR) * Clock\ Period$

- PWM duty cycle for Mode 1 (Low-True):

$$Duty\ Cycle = \frac{\mathrm{CCR}}{\mathrm{ARR} + 1}$$

- PWM duty cycle for Mode 2 (High-True):

$$Duty\ Cycle = 1 - \frac{\mathrm{CCR}}{\mathrm{ARR} + 1}$$