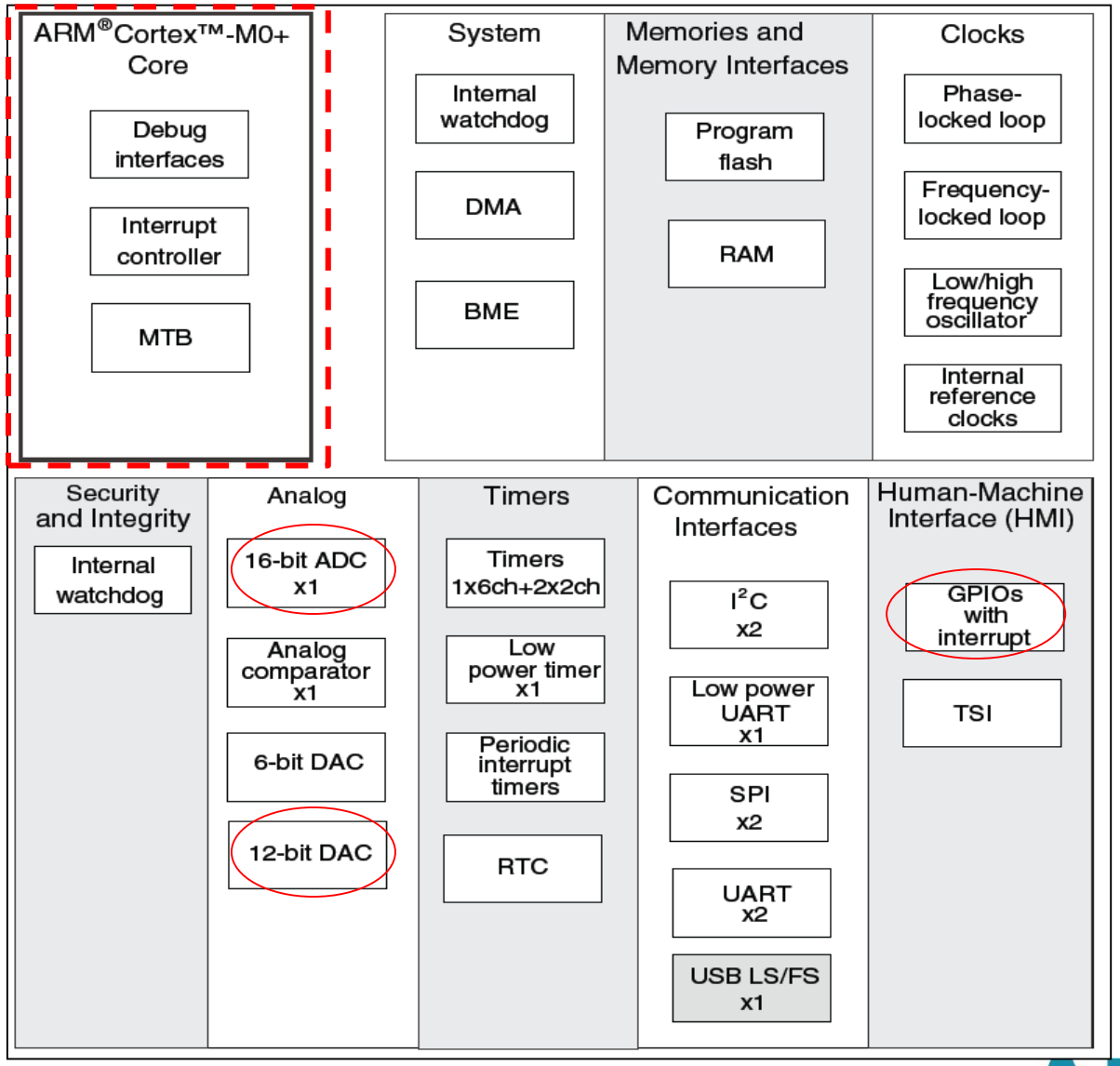
Module 1B - Cortex-M0+ CPU Core

Overview

- Cortex-M0+ Processor Core Registers
- Memory System and Addressing
- Thumb Instruction Set
- References
  - DDI0419C Architecture ARMv6-M Reference Manual

Microcontroller vs. Microprocessor

- Both have a CPU core to execute instructions
- Microcontroller has peripherals for embedded interfacing and control
  - Analog
  - Non-logic level signals
  - Timing
  - Clock generators
  - Communications
    - point to point
    - network
  - Reliability and safety

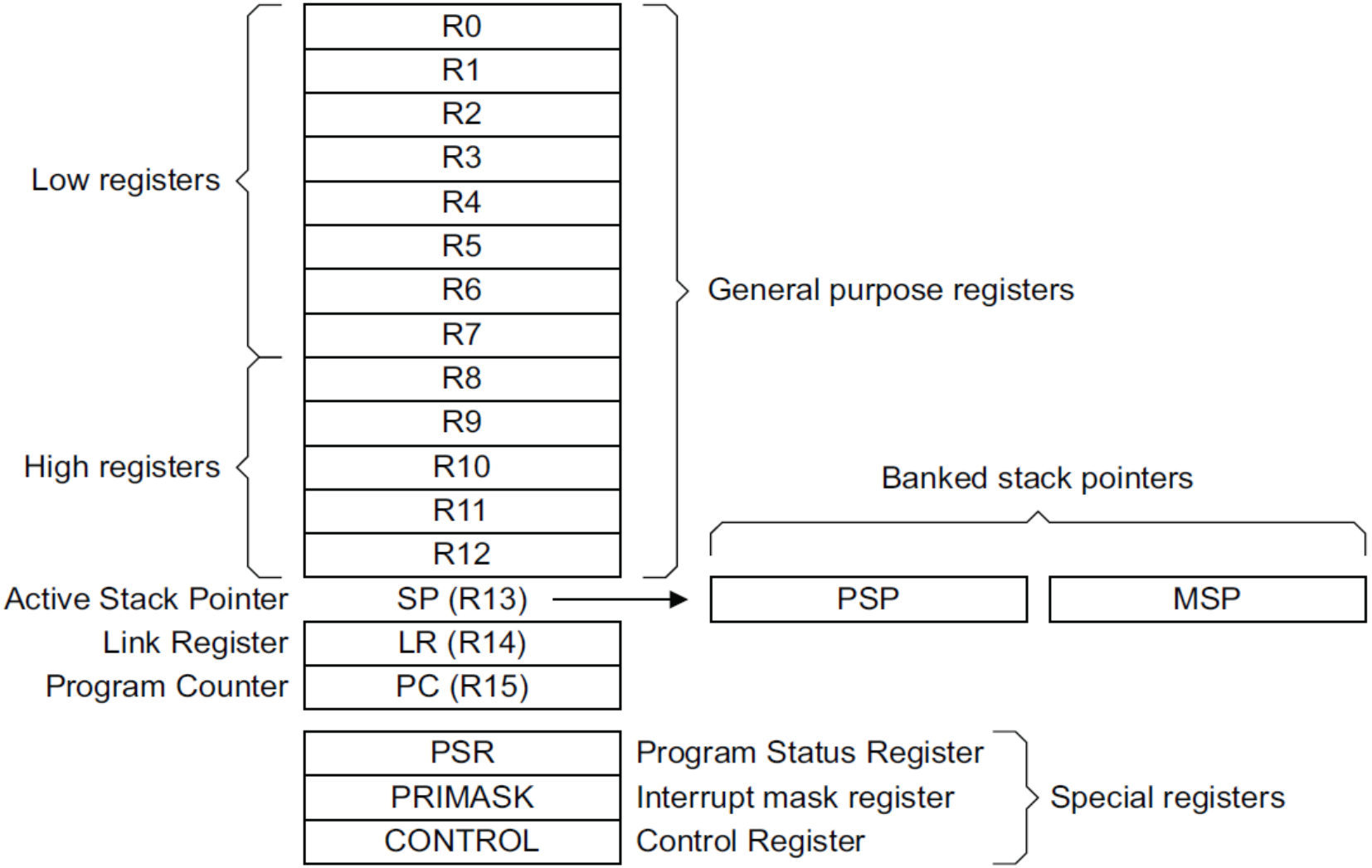


Architectures and Memory Speed

- Load/Store Architecture
  - Developed to simplify CPU design and improve performance
    - *Memory wall*: CPUs keep getting faster than memory
    - Memory accesses slow down CPU, limit compiler optimizations
    - Change instruction set to make most instructions *independent* of memory
  - Data processing instructions can access registers only
    1. Load data into the registers
    2. Process the data
    3. Store results back into memory
  - More effective when more registers are available

- Register/Memory Architecture
  - Data processing instructions can access memory or registers
  - Memory wall is not very high at lower CPU speeds (e.g. under 50 MHz)

ARM Processor Core Registers



ARM Processor Core Registers (32 bits each)

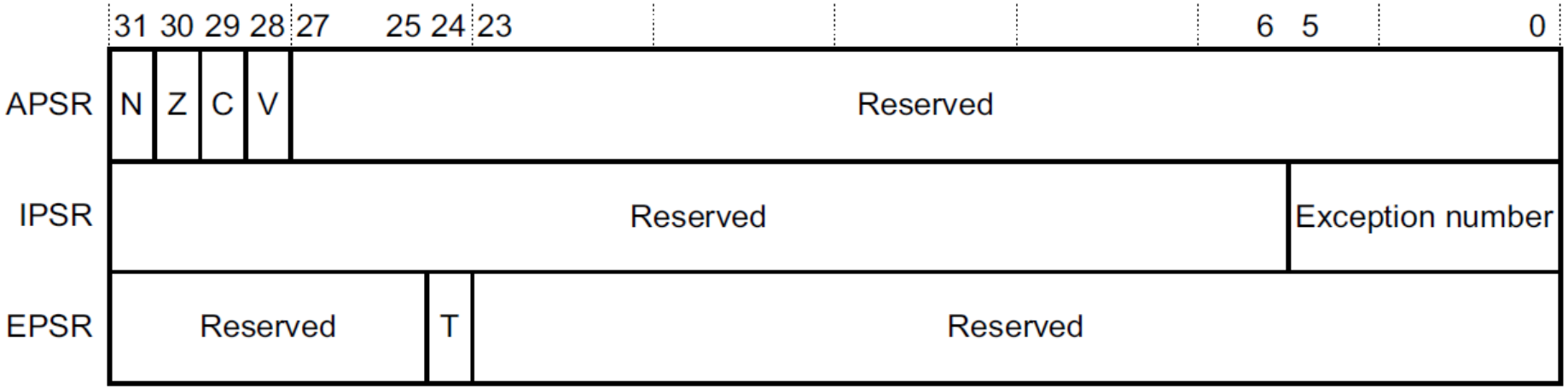
- R0-R12 - General purpose registers for data processing

- SP - Stack pointer (R13)
  - Can refer to one of two SPs
    - Main Stack Pointer (MSP)
    - Process Stack Pointer (PSP)
  - Uses MSP initially, and whenever in Handler mode
  - When in Thread mode, can select either MSP or PSP using SPSEL flag in CONTROL register.

- LR - Link Register (R14)
  - Holds return address when called with Branch & Link instruction (B&L)

- PC - program counter (R15)

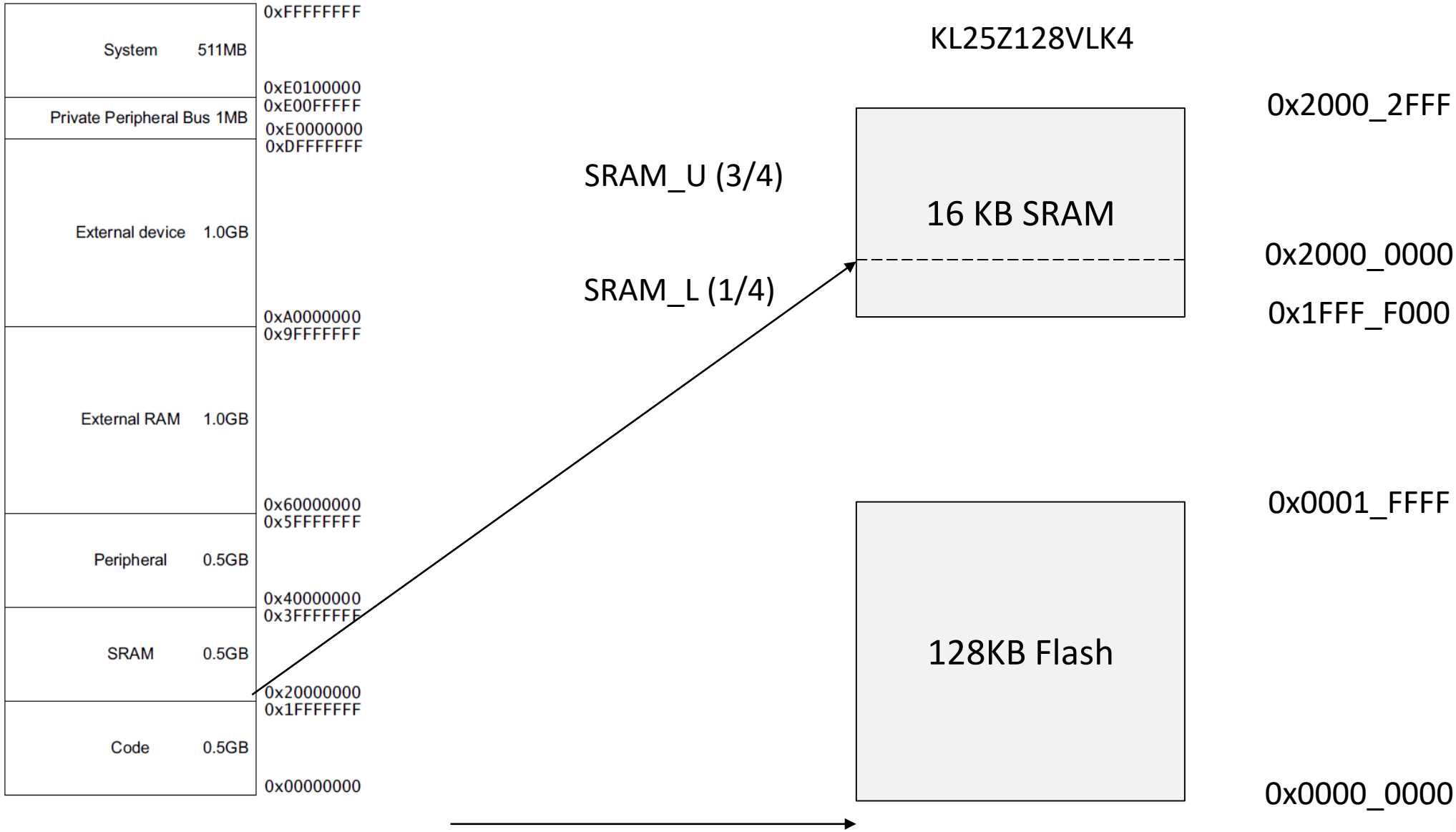
ARM Processor Core Registers



- Program Status Register (PSR) is three views of same register
  - Application PSR (APSR)
    - Condition code flag bits Negative, Zero, oVerflow, Carry
  - Interrupt PSR (IPSR)
    - Holds exception number of currently executing ISR
  - Execution PSR (EPSR)
    - Thumb state

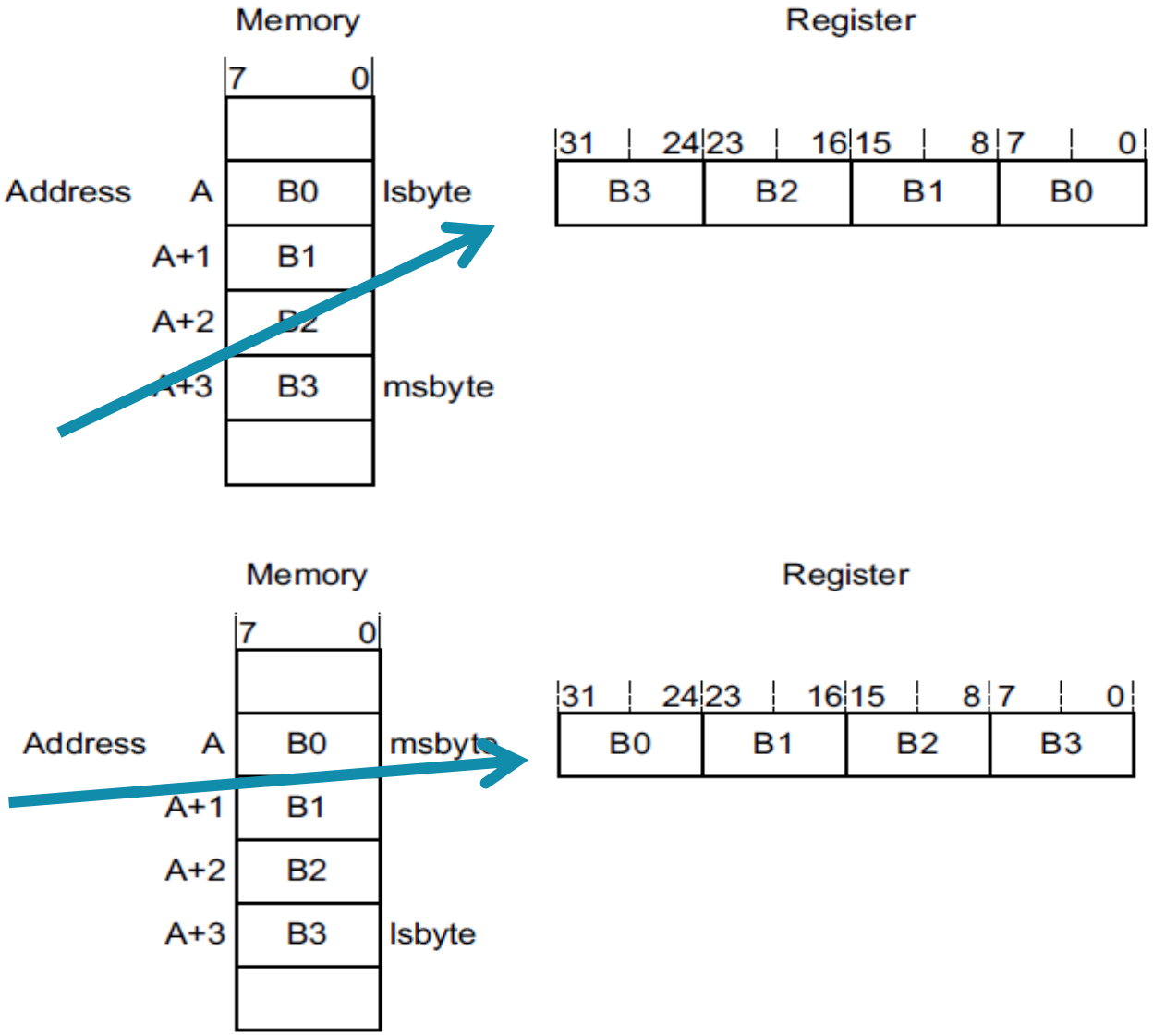
ARM

Memory Maps For Cortex M0+ and MCU



Endianness

- For a multi-byte value, in what order are the bytes stored?
- Little-Endian: Start with least-significant byte
- Big-Endian: Start with most-significant byte
- ARMv6-M is Little Endian
- Data: Depends on implementation, or from reset configuration
  - Kinetis processors are little-endian



ARM, Thumb and Thumb-2 Instructions

- ARM instructions optimized for resource-rich high-performance computing systems
  - Deeply pipelined processor, high clock rate, wide (e.g. 32-bit) memory bus
- Low-end embedded computing systems are different
  - Slower clock rates, shallow pipelines
  - Different cost factors – e.g. code size matters much more, bit and byte operations critical
- Modifications to ARM ISA to fit low-end embedded computing
  - 1995: Thumb instruction set
    - 16-bit instructions
    - Reduces memory requirements (and performance slightly)
  - 2003: Thumb-2 instruction set
    - Adds some 32 bit instructions
    - Improves speed with little memory overhead
  - CPU decodes instructions based on whether in Thumb state or ARM state - controlled by T bit

Instruction Set

- Cortex-M0+ core implements ARMv6-M Thumb instructions
- Only uses Thumb instructions, always in Thumb state
  - Most instructions are 16 bits long, some are 32 bits
  - Most 16-bit instructions can only access low registers (R0-R7), but some can access high registers (R8-R15)
- Conditional execution only supported for 16-bit branch
- 32 bit address space
- Half-word aligned instructions
- See ARMv6-M Architecture Reference Manual for specifics per instruction (Section A.6.7)

Assembly Instructions

- Arithmetic and logic
  - Add, Subtract, Multiply, Divide, Shift, Rotate
- Data movement
  - Load, Store, Move
- Compare and branch
  - Compare, Test, If-then, Branch, compare and branch on zero
- Miscellaneous
  - Breakpoints, wait for events, interrupt enable/disable, data memory barrier, data synchronization barrier

Instruction Format: Labels

```
label   mnemonic operand1, operand2, operand3   ; comments
```

- Place marker, marking the memory address of the current instruction
- Used by branch instructions to implement **if-then** or **goto**
- Must be unique

Instruction Format: Mnemonic

```
label   mnemonic operand1, operand2, operand3  ; comments
```

- The name of the instruction
- Operation to be performed by processor core

Instruction Format: Operands

```
label   mnemonic operand1, operand2, operand3   ; comments
```

- Operands
  - Registers
  - Constants (called *immediate values*)
- Number of operands varies
  - No operands: DSB
  - One operand: BX LR
  - Two operands: CMP R1, R2
  - Three operands: ADD R1, R2, R3
  - Four operands: MLA R1, R2, R3, R4
- Normally
  - operand1 is the destination register, and operand2 and operand3 are source operands.
  - operand2 is usually a register, and the first source operand
  - operand3 may be a register, an immediate number, a register shifted to a constant number of bits, or a register plus an offset (used for memory access).

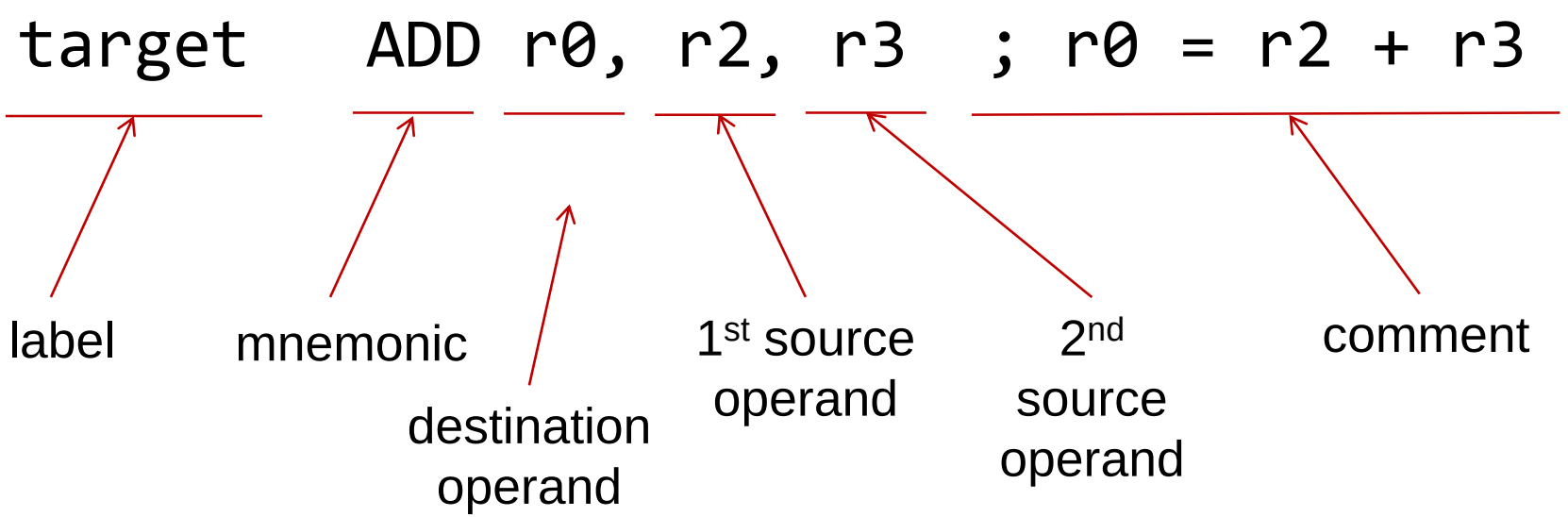
Instruction Format: Comments

```
label   mnemonic operand1, operand2, operand3  ; comments
```

- Everything after the semicolon (;) is a comment
- Explain programmers' intentions or assumptions

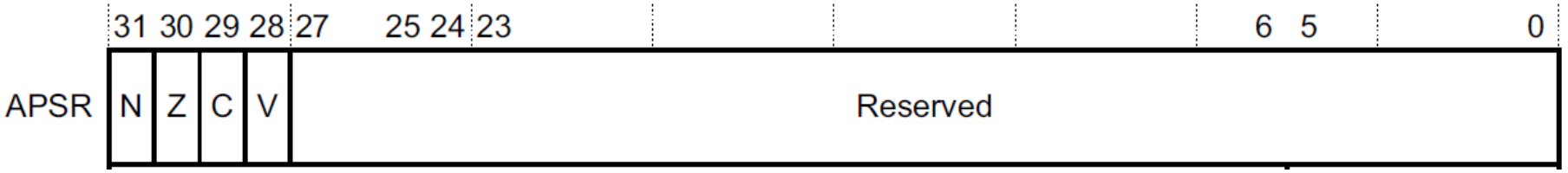
ARM Instruction Format

```
label mnemonic operand1, operand2, operand3    ; comments
```



ARM

Update Condition Codes in APSR?



- “S” suffix indicates the instruction updates APSR
  - ADD vs. ADDS
  - ADC vs. ADCS
  - SUB vs. SUBS
  - MOV vs. MOVS

ARM

Instruction Set Summary

Instruction Type	Instructions
Move	MOV
Load/Store	LDR, LDRB, LDRH, LDRSH, LDRSB, LDM, STR, STRB, STRH, STM
Add, Subtract, Multiply	ADD, ADDS, ADCS, ADR, SUB, SUBS, SBCS, RSBS, MULS
Compare	CMP, CMN
Logical	ANDS, EORS, ORRS, BICS, MVNS, TST
Shift and Rotate	LSLS, LSRS, ASRS, RORS
Stack	PUSH, POP
Conditional branch	IT, B, BL, B{cond}, BX, BLX
Extend	SXTH, SXTB, UXTH, UXTB
Reverse	REV, REV16, REVSH
Processor State	SVC, CPSID, CPSIE, SETEND, BKPT
No Operation	NOP
Hint	SEV, WFE, WFI, YIELD

Load/Store Register

- ARM is a load/store architecture, so must process data in registers (not memory)

- LDR: load register with word (32 bits) from memory
  - LDR <Rt>, source address

- STR: store register contents (32 bits) to memory
  - STR <Rt>, destination address

Load-Modify-Store

C statement

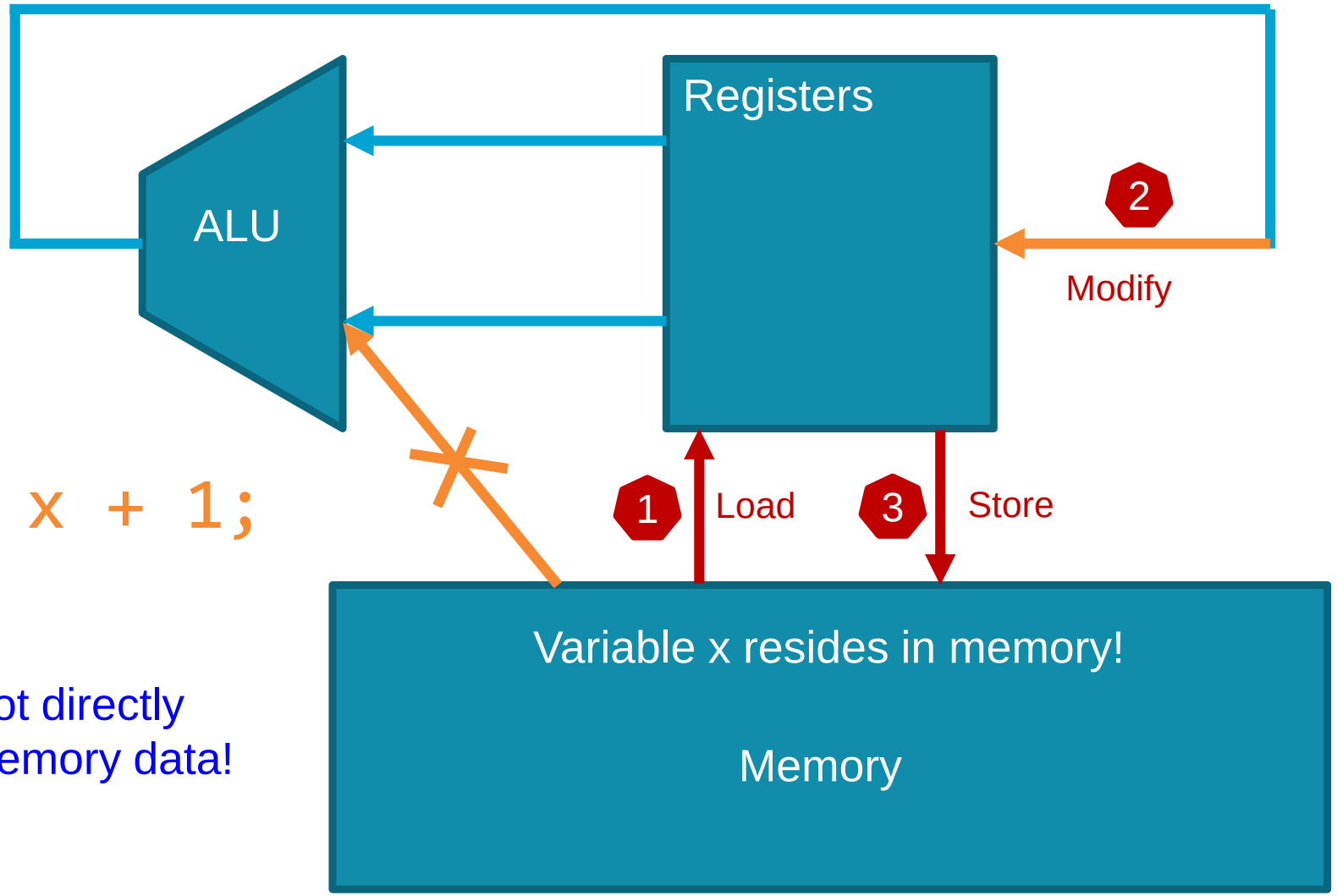
```
x = x + 1;
```

Assume variable X resides in memory and is a 32-bit integer

```
; Assume the memory address of x is stored in
r1

LDR r0, [r1]      ; load value of x from memory
ADD r0, r0, #1    ; x = x + 1
STR r0, [r1]      ; store x into memory
```

3 Steps: Load, Modify, Store



ALU cannot directly operate memory data!

ARM

Example 1: Adding Two Integers

```
int x = 1;
int y = 2;
int z;
```

If values are in registers

- Value of `x` in `r0`
- Value of `y` in `r1`
- Value of `z` in `r2`

C Statement

```
z = x + y;
```

Assembly Statement

ARM

Adding Two Integers

```
int x = 1;
int y = 2;
int z;
```

If values are in registers

- Value of `x` in `r0`
- Value of `y` in `r1`
- Value of `z` in `r2`

C Statement

```
z = x + y;
```

Assembly Statement

```
ADD r2, r1, r0
```

Destination

Source Operand 2

Source Operand 1

Adding Two Integers

```
int x = 1;
int y = 2;
int z;
```

If addresses are in registers

- Address of `x` in `r0`
- Address of `y` in `r1`
- Address of `z` in `r2`

C Statement

```
z = x + y;
```

```
LDR r3, [r0] ; Read x
LDR r4, [r1] ; Read y
ADD r5, r3, r4
STR r5, [r2] ; Write z
```

Example 2: Set a Bit in C

```
a |= (1 << k)
```

or

```
a = a | (1 << k)
```

Example: k = 5

a	a_7	a_6	**a_5**	a_4	a_3	a_2	a_1	a_0
1 << k	0	0	**1**	0	0	0	0	0
a | (1 << k)	a_7	a_6	**1**	a_4	a_3	a_2	a_1	a_0

The other bits should not be affected.

ARM

Set a Bit in Assembly

```
a |= (1 << 5)
```

```
Solution:
    MOVS r4, #1         ; r4 = 1
    LSLS r4, r4, #5     ; r4 = 1<<5
    ORRS r0, r0, r4     ; r0 = r0 | 1<<5
```


ARM

Example 3: 64 Bit Addition

```
start
  ; C = A + B
  ; Two 64-bit integers A (r1,r0) and B (r3, r2).
  ; Result C (r5, r4)
  ; A = 00000002FFFFFFFF
  ; B = 0000000400000001
  LDR  r0, =0xFFFFFFFF  ;  A's lower 32 bits
  LDR  r1, =0x00000002  ;  A's upper 32 bits
  LDR  r2, =0x00000001  ;  B's lower 32 bits
  LDR  r3, =0x00000004  ;  B's upper 32 bits

  ;  Add A and B
  ADDS r4, r2, r0  ; C[31..0] = A[31..0] +  B[31..0],  update Carry
  ADC  r5, r3, r1   ; C[64..32] = A[64..32] + B[64..32]  +  Carry

stop  B  stop
```