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

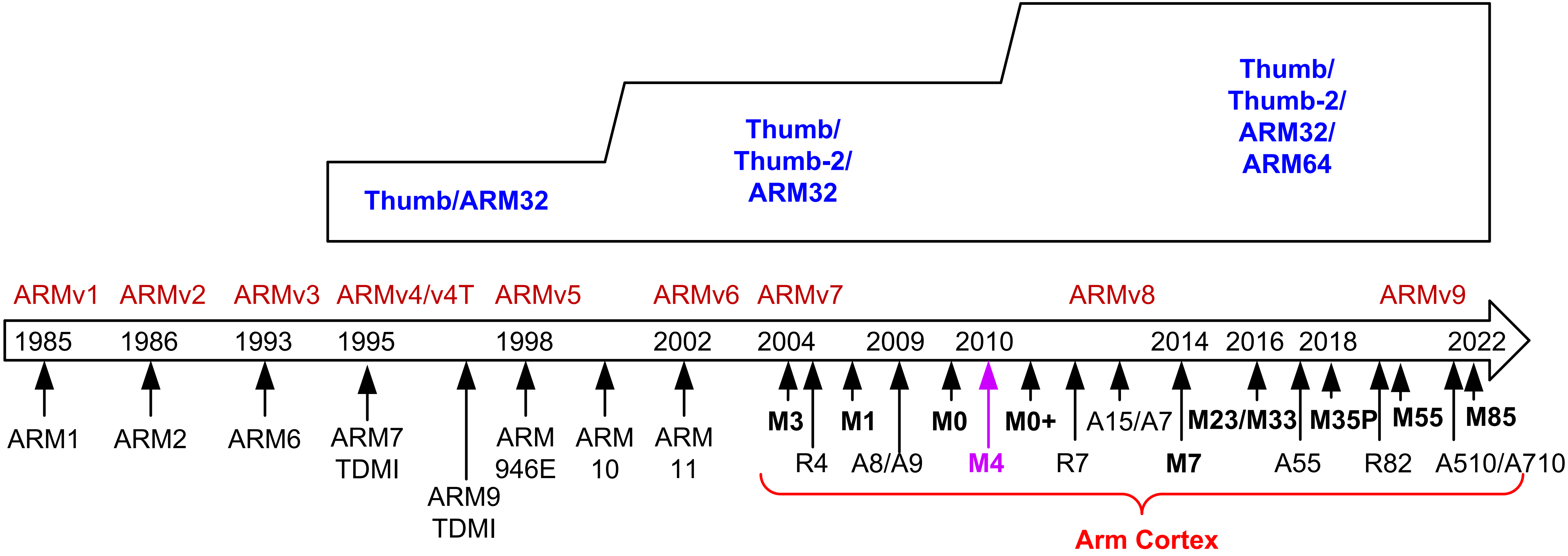
Chapter 3
ARM Instruction Set Architecture

Zonghua 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/

History

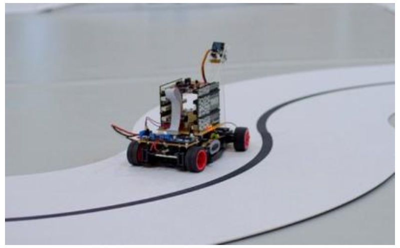

Thumb/ARM32
Thumb/ Thumb-2/ ARM32
Thumb/ Thumb-2/ ARM32/ ARM64
ARMv1
ARMv2
ARMv3
ARMv4/v4T
ARMv5
ARMv6
ARMv7
ARMv8
ARMv9
1985
1986
1993
1995
1998
2002
2004
2009
2010
2014
2016
2018
2022
ARM1
ARM2
ARM6
ARM7 TDMI
ARM9 TDMI
ARM 946E
ARM 10
ARM 11
M3
R4
M1
A8/A9
M0
M4
M0+
R7
A15/A7
M7
M23/M33
A55
M35P
R82
M55
M85
A510/A710
Arm Cortex

ARM Processors

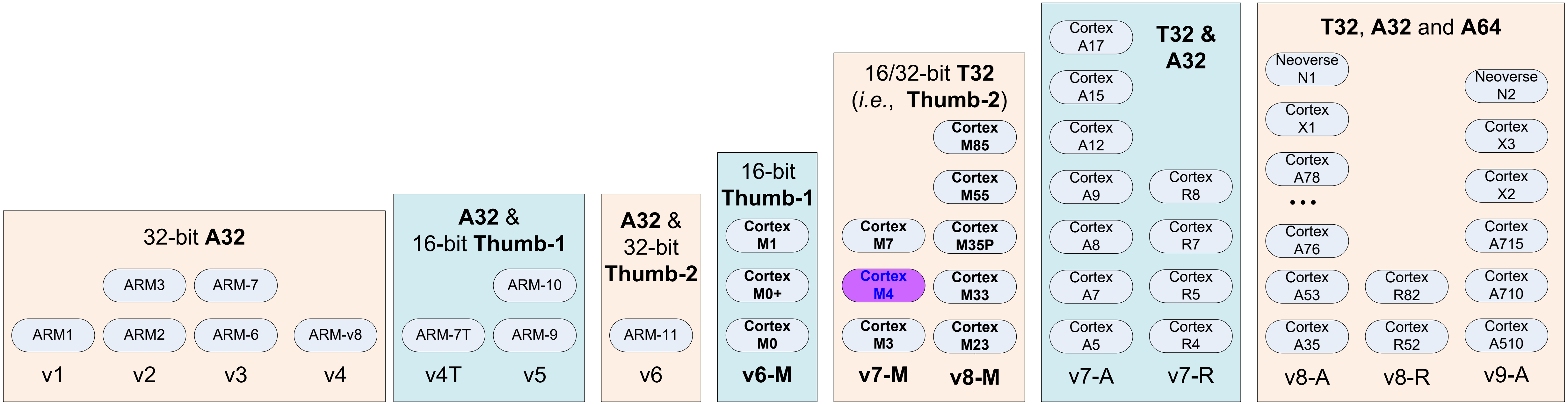
- ARM Cortex-A family:
  - **A**pplications processors
  - Support OS and high-performance applications
  - Such as Smartphones, Smart TV

- ARM Cortex-R family:
  - **R**eal-time processors with high performance and high reliability
  - Support real-time processing and mission-critical control

- ARM Cortex-M family:
  - **M**icrocontroller
  - Cost-sensitive, support SoC



ARM Family



32-bit **A32**

ARM1 v1

ARM3 ARM2 v2

ARM-7 ARM-6 v3

ARM-v8 v4

A32 & 16-bit **Thumb-1**

ARM-7T v4T

ARM-10 ARM-9 v5

A32 & 32-bit **Thumb-2**

ARM-11 v6

16-bit **Thumb-1**

Cortex M1 **Cortex M0+** **Cortex M0** **v6-M**

16/32-bit **T32** (*i.e.*, **Thumb-2**)

Cortex M7 **Cortex M4** **Cortex M3** **v7-M**

Cortex M85 **Cortex M55** **Cortex M35P** **Cortex M33** **Cortex M23** **v8-M**

T32 & **A32**

Cortex A17 Cortex A15 Cortex A12 Cortex A9 Cortex A8 Cortex A7 Cortex A5 v7-A

Cortex R8 Cortex R7 Cortex R5 Cortex R4 v7-R

T32, **A32** and **A64**

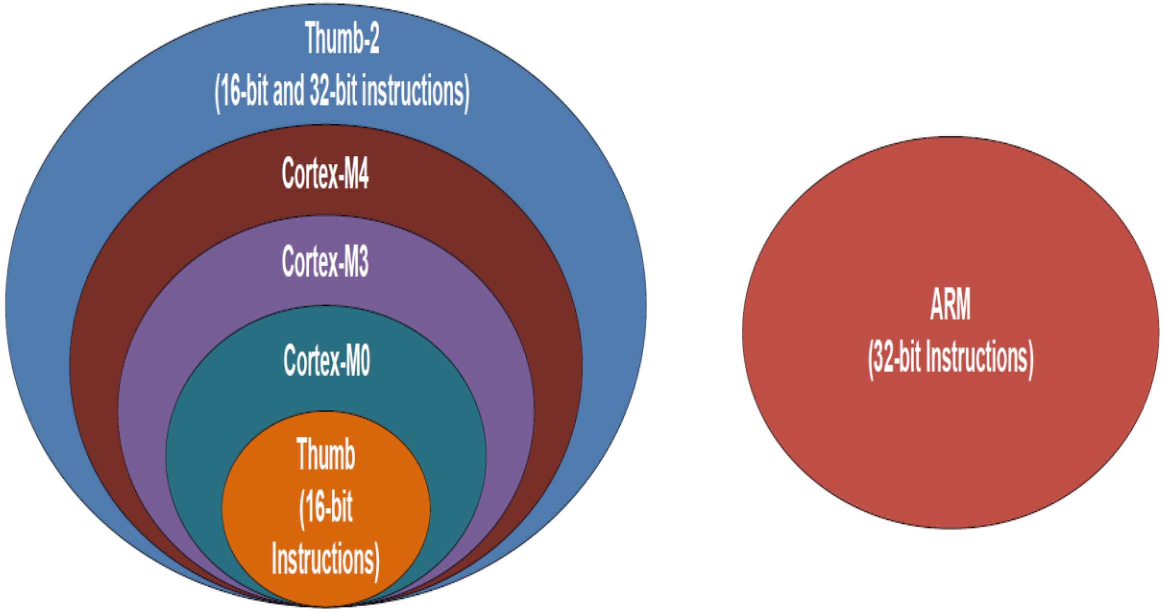
Neoverse N1 Cortex X1 Cortex A78 ••• Cortex A76 Cortex A53 Cortex A35 v8-A

Cortex R82 Cortex R52 v8-R

Neoverse N2 Cortex X3 Cortex X2 Cortex A715 Cortex A710 Cortex A510 v9-A

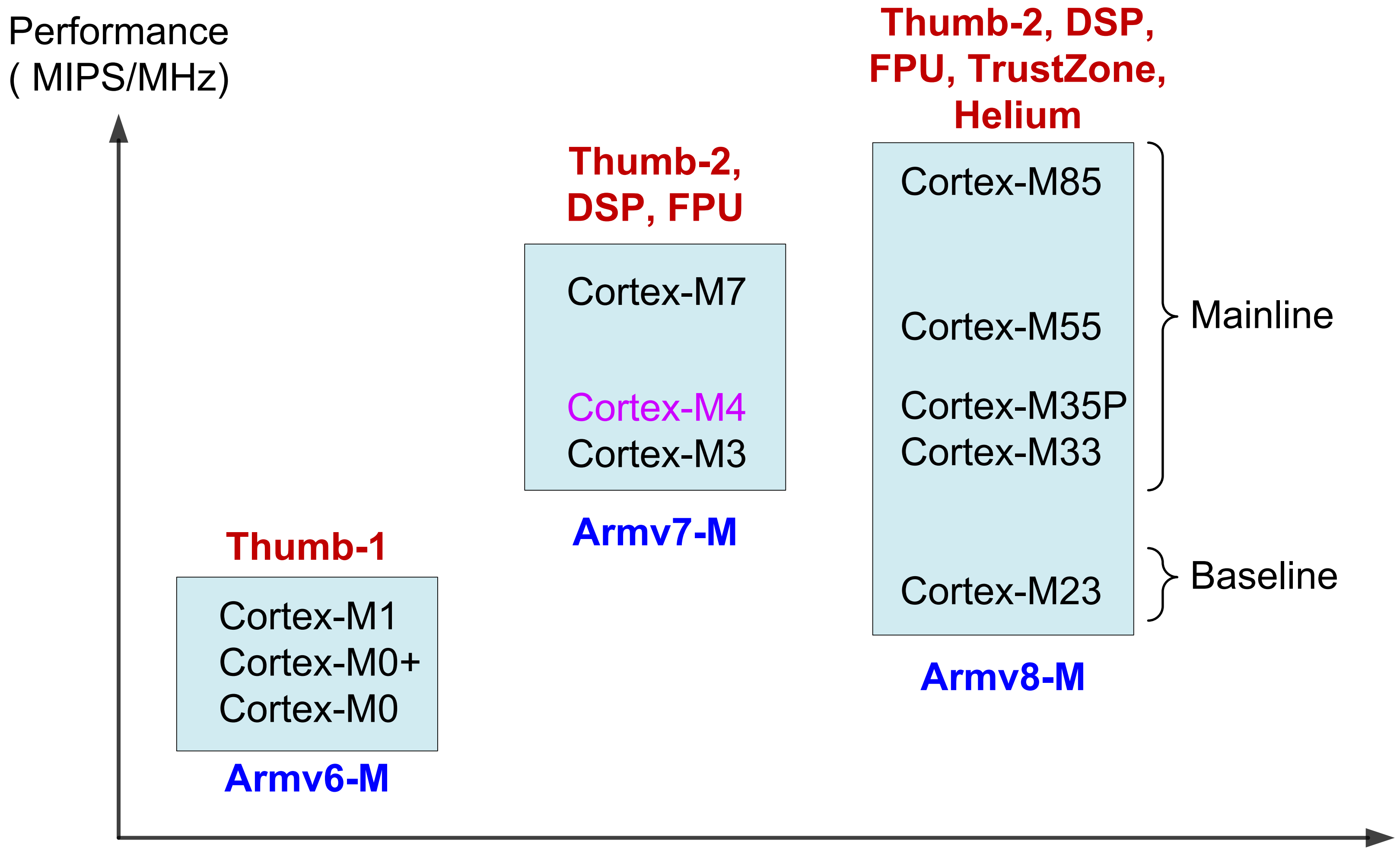
M: Microcontroller. **A**: Application. **R**: Real-time

Instruction Sets



- Instructions:
  - Encoded to binary machine code by assembler
  - Executed at runtime by hardware
- Early 32-bit ARM vs Thumb/Thumb-2
  - Early ARM has larger power consumption and larger program size
  - 16-bit Thumb, first used in ARM7TDMI processors in 1995
  - Thumb-2: a mix of 16-bit (high code density) and 32-bit (high performance) instructions
- ARM Cortex-M:
  - Subset of Thumb-2

ARM Processors



Performance (MIPS/MHz)

Thumb-1

Cortex-M1
Cortex-M0+
Cortex-M0

Armv6-M

Thumb-2, DSP, FPU

Cortex-M7

Cortex-M4
Cortex-M3

Armv7-M

Thumb-2, DSP, FPU, TrustZone, Helium

Cortex-M85

Cortex-M55

Cortex-M35P
Cortex-M33

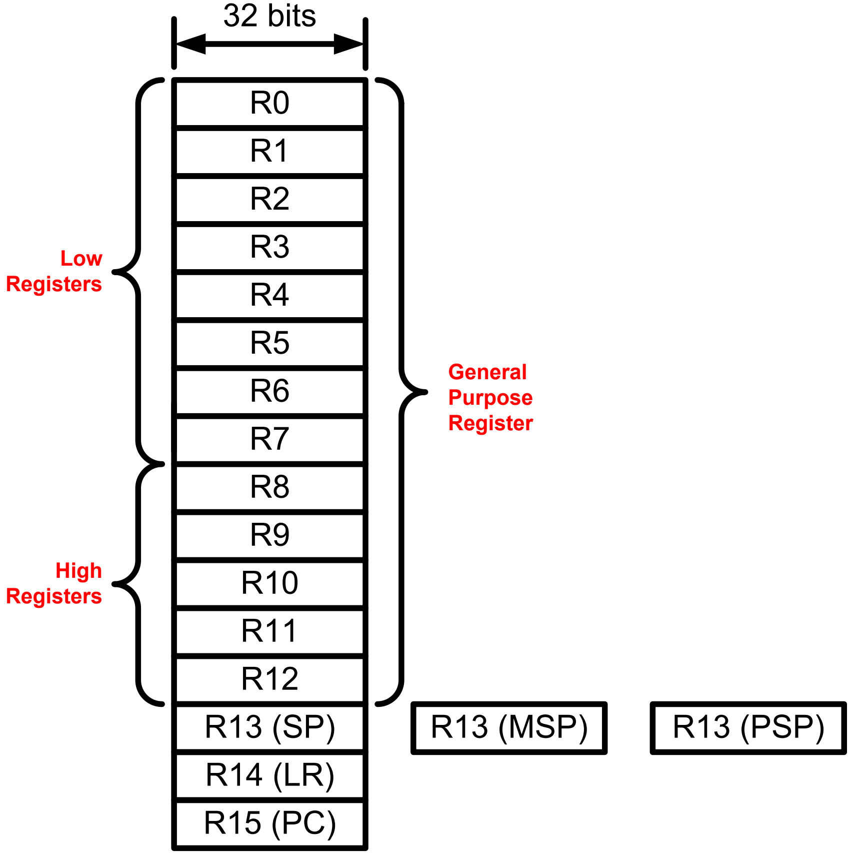
Cortex-M23

Armv8-M

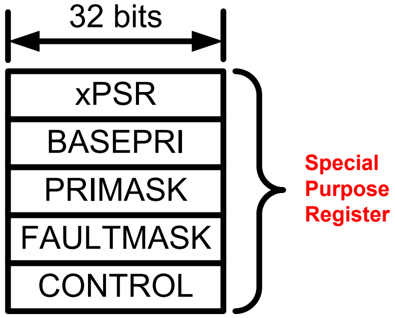
Mainline

Baseline

Processor Registers



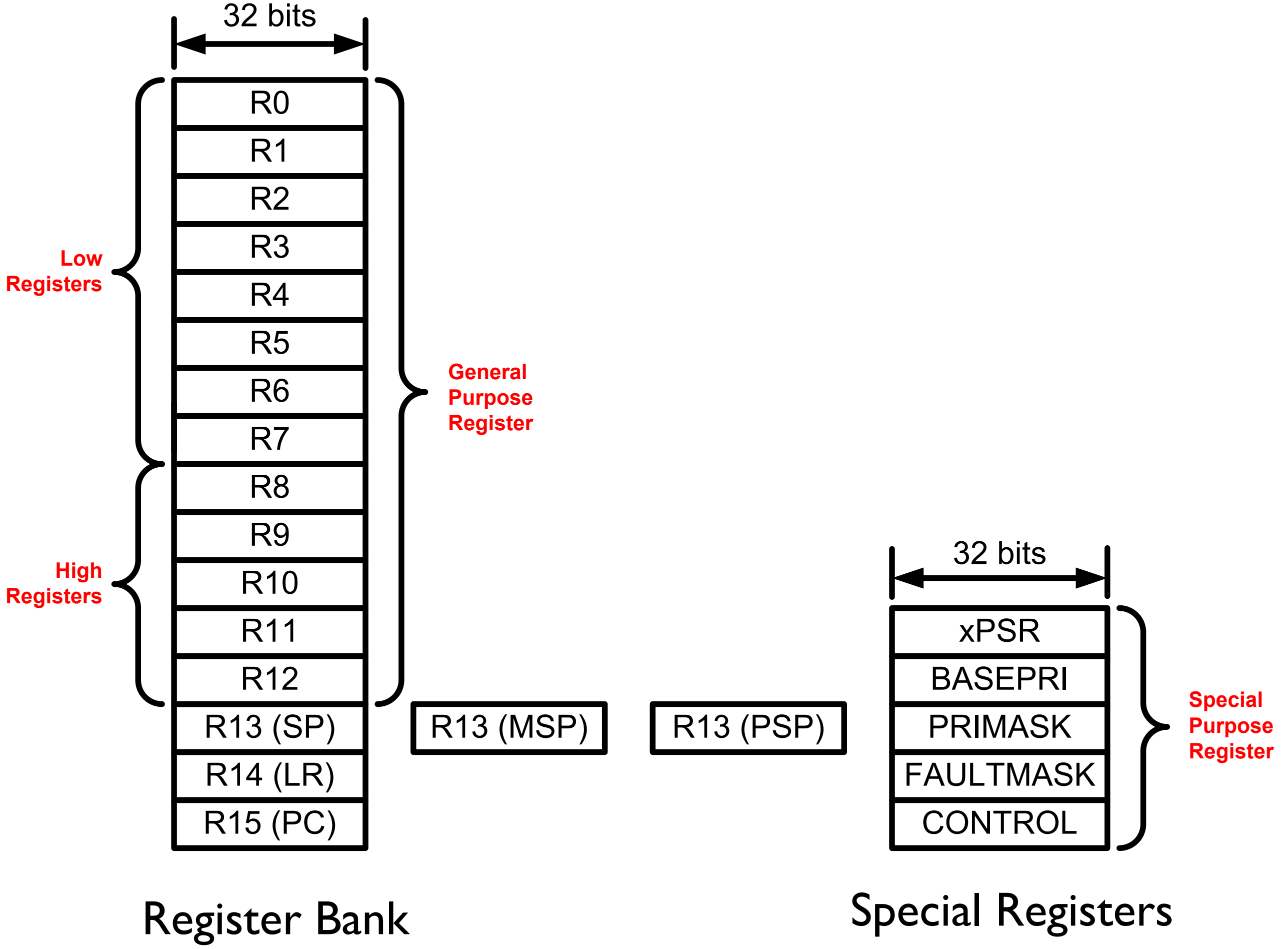
Register Bank



Special Registers

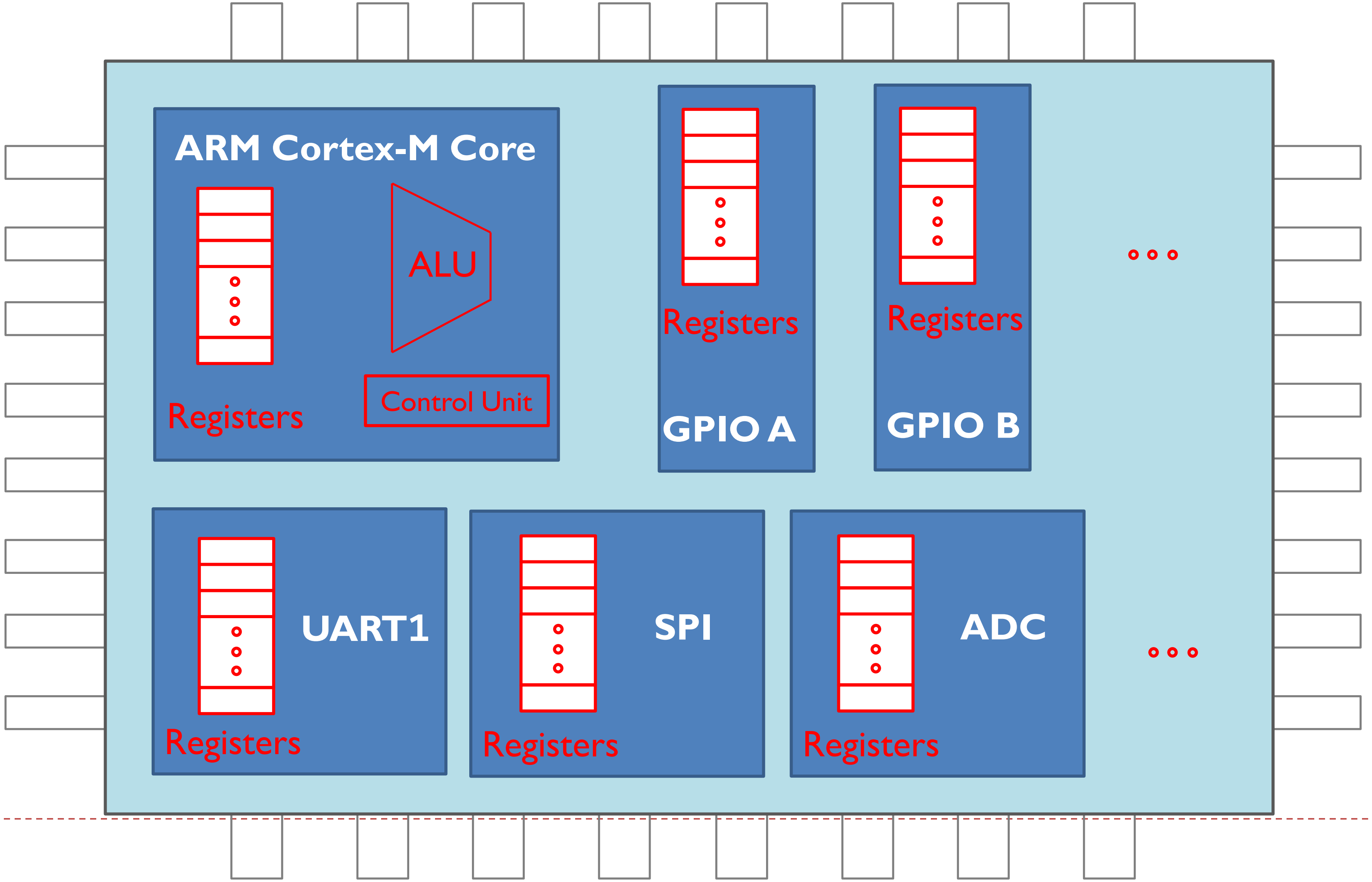
- Fastest way to read and write
- Registers are within the processor chip
- Each register has 32 bits
- ARM Cortex-M4 has
  - Register Bank: `R0 – R15`
    - **`R0-R12`**: 13 general-purpose registers
    - **`R13`**: Stack pointer (Shadow of MSP or PSP)
    - **`R14`**: Link register (LR)
    - **`R15`**: Program counter (PC)
  - Special registers
    - xPSR, BASEPRI, PRIMASK, etc

Processor Registers



- Low Registers (R0 – R7)
  - Can be accessed by any instruction
- High Register (R8 – R12)
  - Can only be accessed by some instructions
- Stack Pointer (R13)
  - Cortex-M4 supports two stacks
  - Main SP (MSP) for privileged access (e.g. exception handler)
  - Process SP (PSP) for application access
- Program Counter (R15)
  - Memory address of the current instruction

Processor Registers *vs* Peripheral Registers

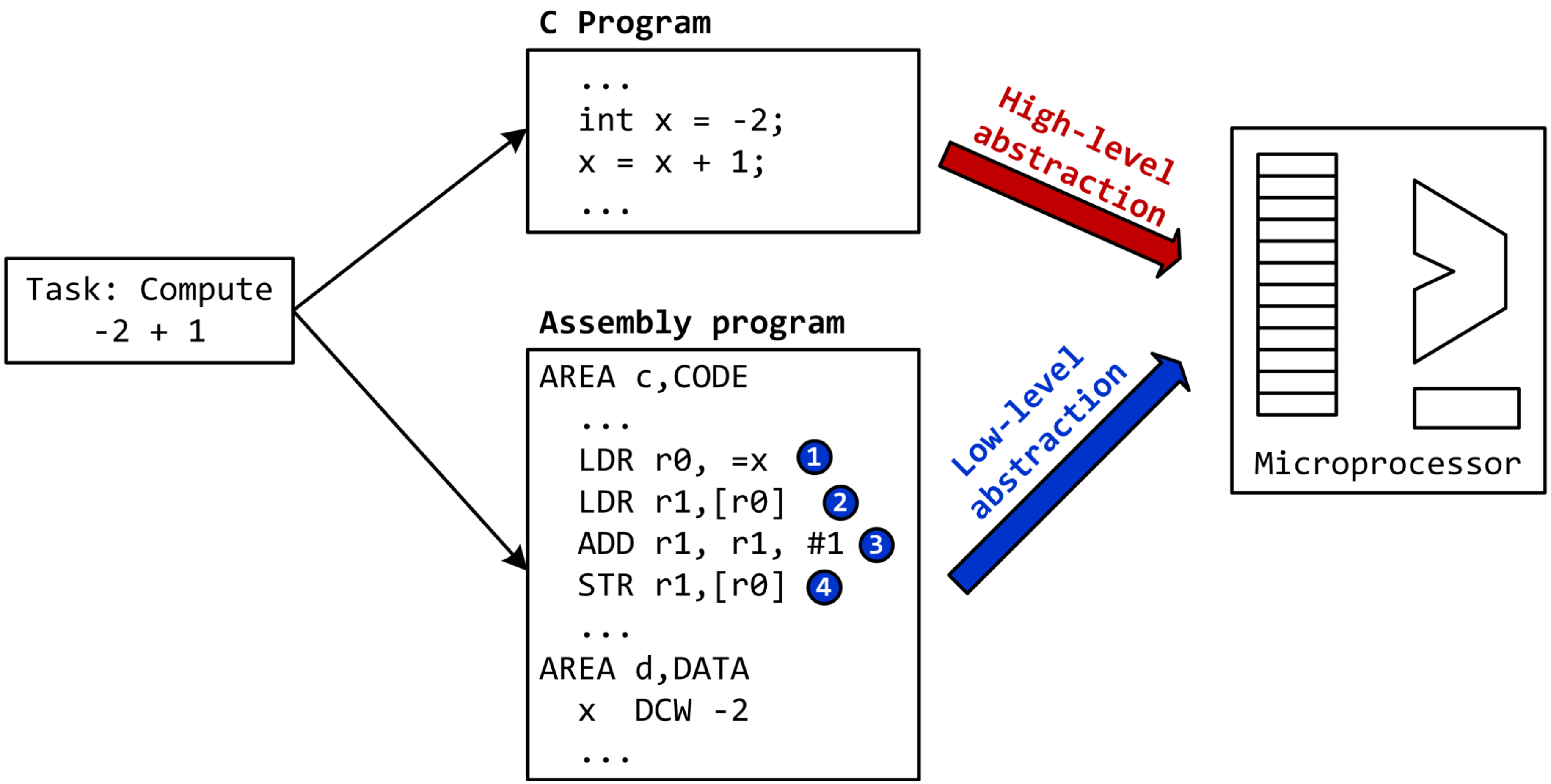


Processor Registers vs Peripheral Registers

- Processor can directly access processor registers
  - `ADD r3,r1,r0   ; r3 = r1 + r0`

- Processor access peripheral registers via memory mapped I/O
  - Each peripheral register is assigned a fixed memory address at the chip design stage
  - Processor treats peripherals registers the same as data memory
  - Processor uses load/store instructions to read from/write to memory (to be covered in future lectures)

C *vs* Assembly



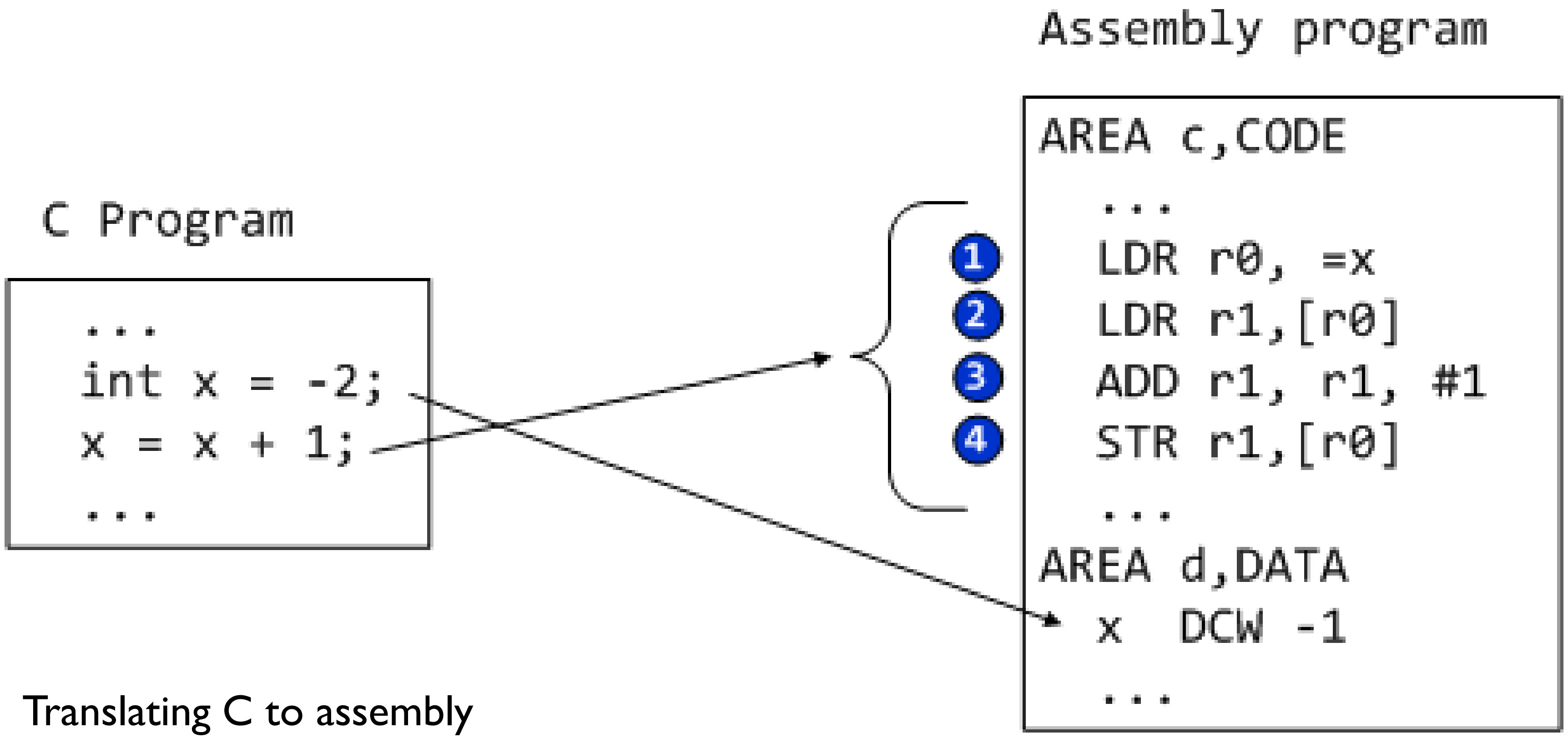
C Program

```
  ...
  int x = -2;
  x = x + 1;
  ...
```

Assembly program

```
AREA c,CODE
  ...
  LDR r0, =x ①
  LDR r1,[r0]  ②
  ADD r1, r1, #1 ③
  STR r1,[r0] ④
  ...
AREA d,DATA
  x  DCW -2
  ...
```

Load-Modify-Store



C Program

```
...
int x = -2;
x = x + 1;
...
```

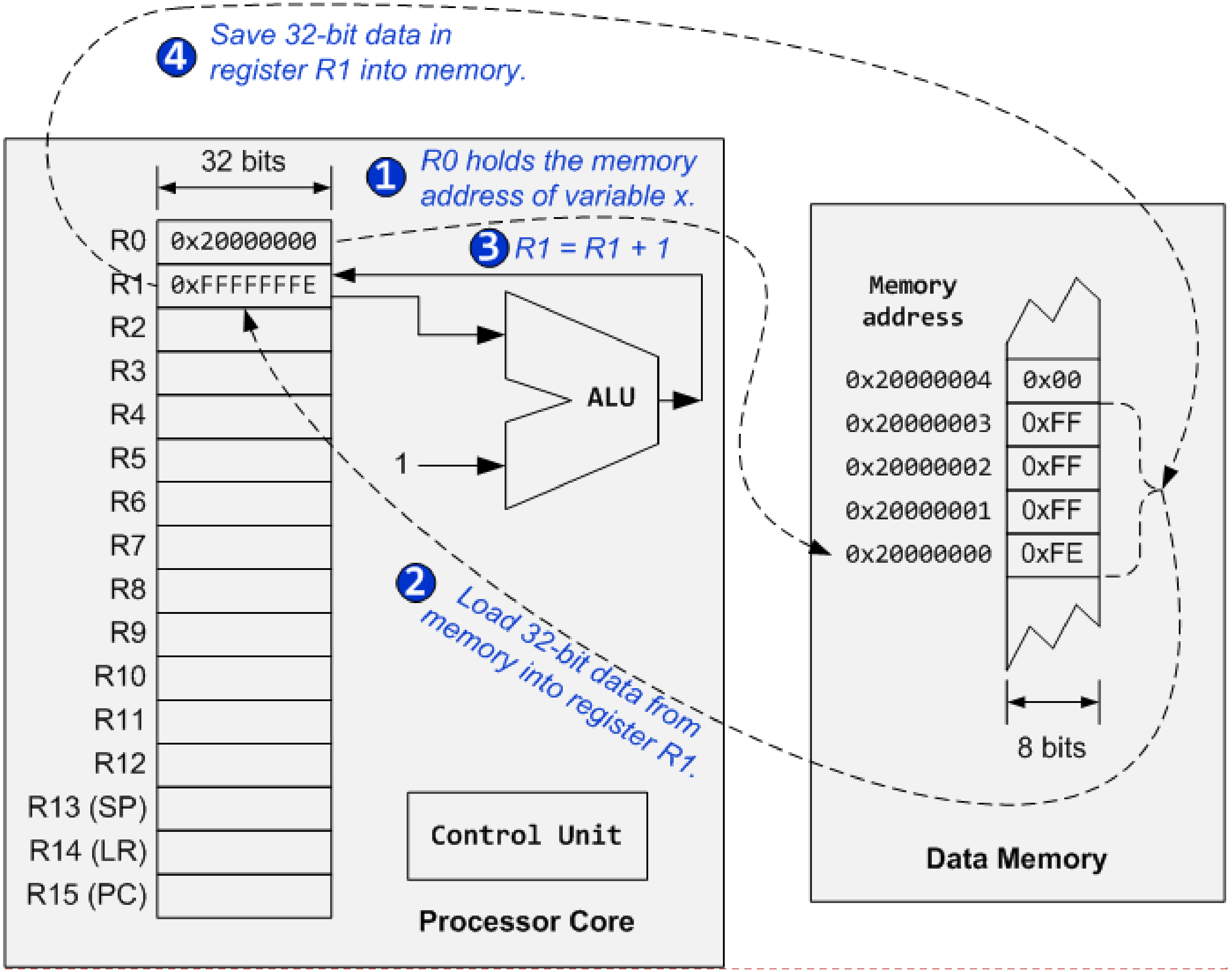
Assembly program

```
AREA c,CODE
   ...
   LDR r0, =x
   LDR r1,[r0]
   ADD r1, r1, #1
   STR r1,[r0]
   ...
AREA d,DATA
   x  DCW -1
   ...
```

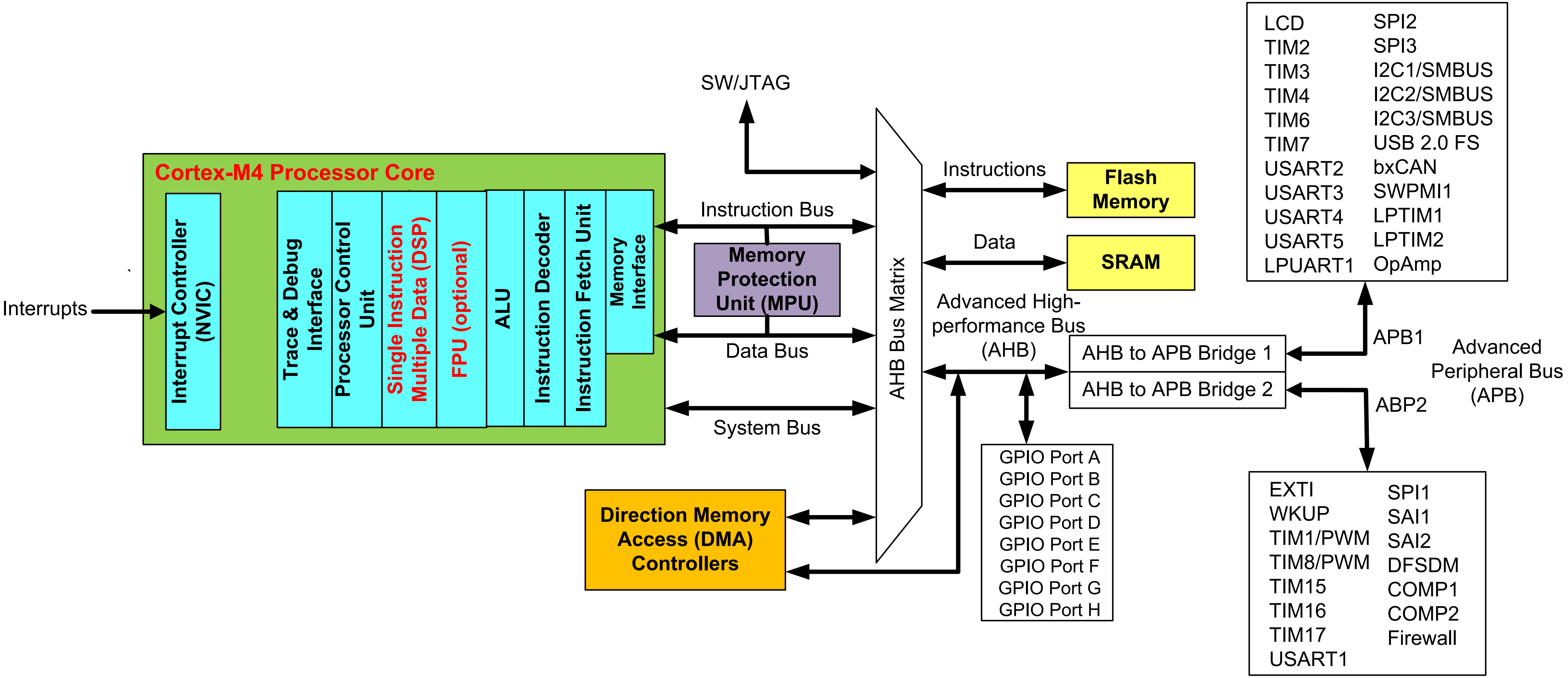
Translating C to assembly

- Load values from memory into registers
- Modify value by applying arithmetic operations
- Store result from register to memory

Load-Modify-Store



ARM Cortex-M4 Organization (STM32L4)



System-on-a-chip

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
```

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).

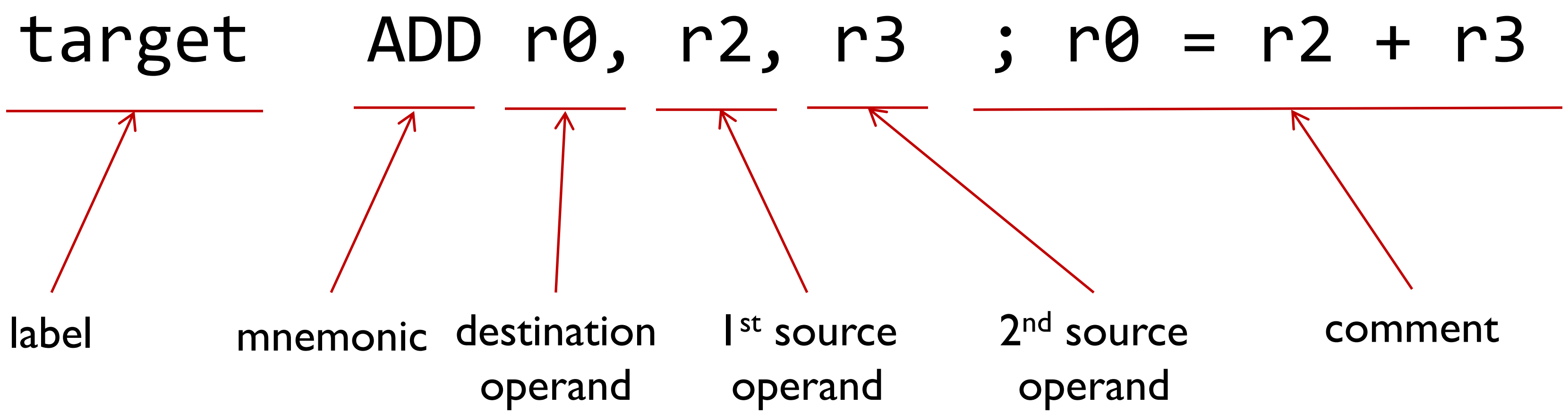
Instruction Format: Comments

```
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
```



```
target   ADD r0, r2, r3  ; r0 = r2 + r3
```

label

mnemonic

destination operand

1st source operand

2nd source operand

comment

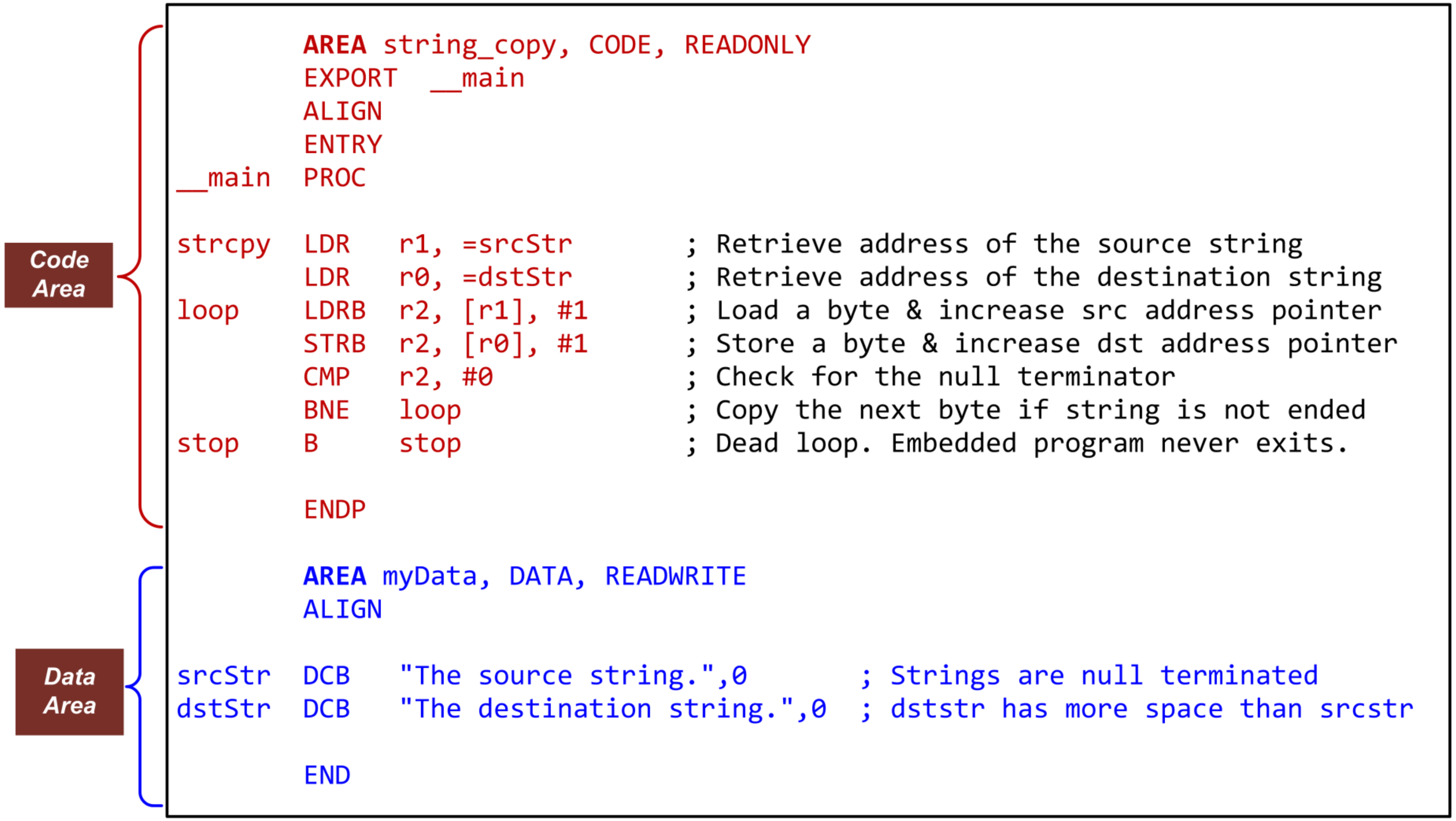
ARM Instruction Format

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

Examples: Variants of the ADD instruction

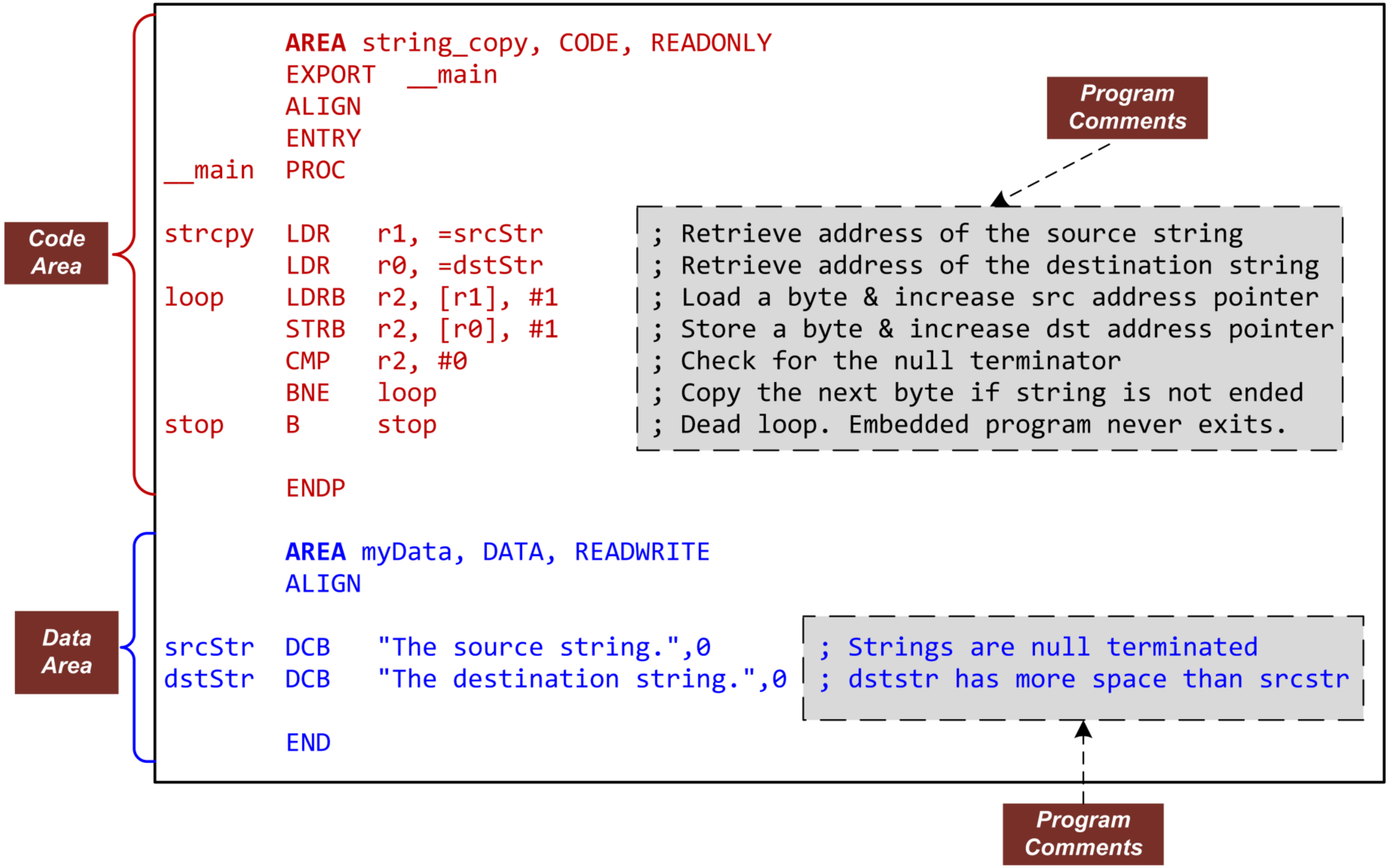
```
ADD r1, r2, r3      ; r1 = r2 + r3
ADD r1, r3          ; r1 = r1 + r3
ADD r1, r2, #4      ; r1 = r2 + 4
ADD r1, #15         ; r1 = r1 + 15
```

Example Assembly Program: Copying a String



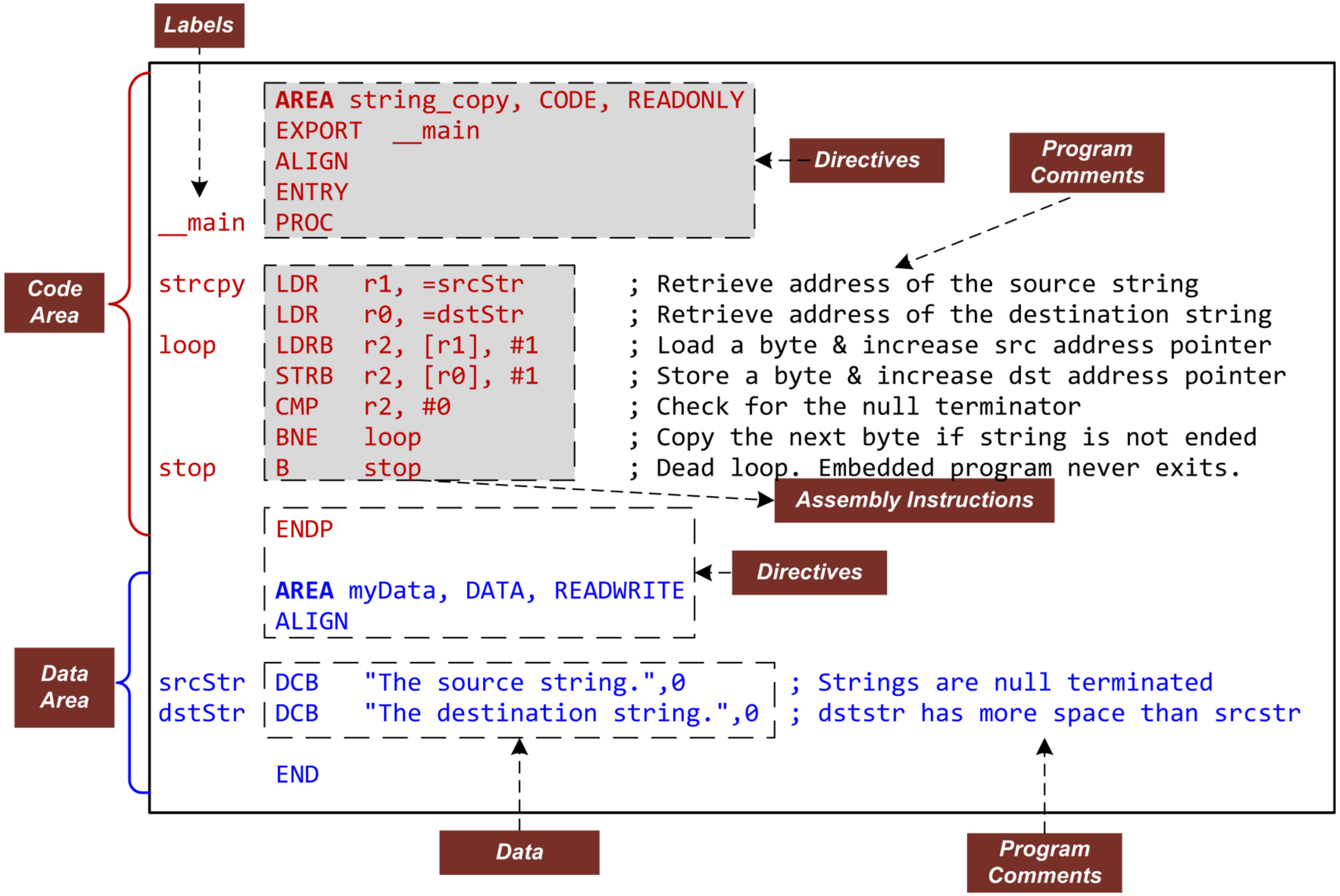
```
        AREA string_copy, CODE, READONLY
        EXPORT  __main
        ALIGN
        ENTRY
__main  PROC

strcpy  LDR   r1, =srcStr       ; Retrieve address of the source string
        LDR   r0, =dstStr       ; Retrieve address of the destination string
loop    LDRB  r2, [r1], #1      ; Load a byte & increase src address pointer
        STRB  r2, [r0], #1      ; Store a byte & increase dst address pointer
        CMP   r2, #0            ; Check for the null terminator
        BNE   loop              ; Copy the next byte if string is not ended
stop    B     stop              ; Dead loop. Embedded program never exits.

        ENDP

        AREA myData, DATA, READWRITE
        ALIGN

srcStr  DCB   "The source string.",0       ; Strings are null terminated
dstStr  DCB   "The destination string.",0  ; dststr has more space than srcstr

        END
```

Example Assembly Program: Copying a String



```
        AREA string_copy, CODE, READONLY
        EXPORT  __main
        ALIGN
        ENTRY
__main  PROC

strcpy  LDR   r1, =srcStr        ; Retrieve address of the source string
        LDR   r0, =dstStr        ; Retrieve address of the destination string
loop    LDRB  r2, [r1], #1       ; Load a byte & increase src address pointer
        STRB  r2, [r0], #1       ; Store a byte & increase dst address pointer
        CMP   r2, #0             ; Check for the null terminator
        BNE   loop               ; Copy the next byte if string is not ended
stop    B     stop               ; Dead loop. Embedded program never exits.

        ENDP

        AREA myData, DATA, READWRITE
        ALIGN

srcStr  DCB   "The source string.",0       ; Strings are null terminated
dstStr  DCB   "The destination string.",0  ; dststr has more space than srcstr

        END
```

Example Assembly Program: Copying a String



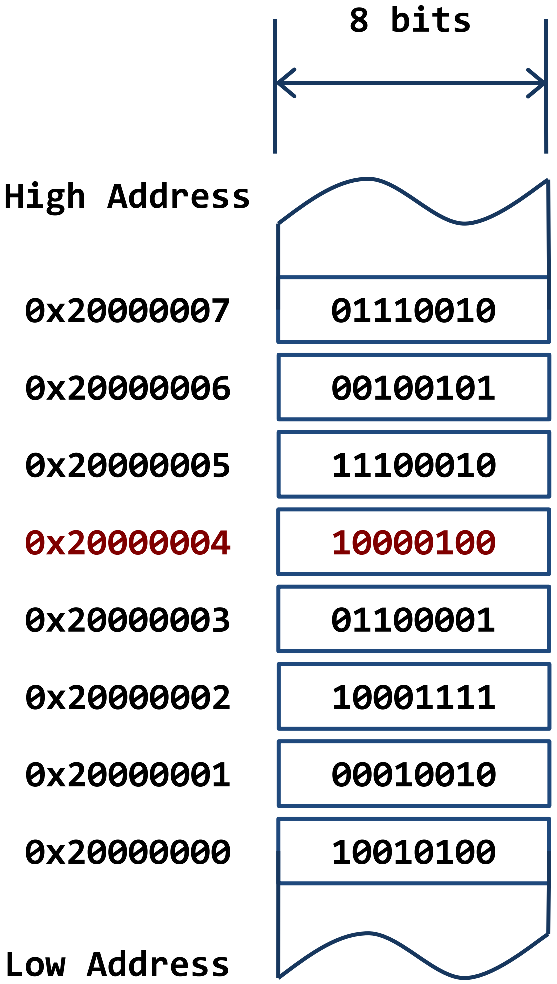
```
        AREA string_copy, CODE, READONLY
        EXPORT   __main
        ALIGN
        ENTRY
__main  PROC

strcpy  LDR    r1, =srcStr        ; Retrieve address of the source string
        LDR    r0, =dstStr        ; Retrieve address of the destination string
loop    LDRB   r2, [r1], #1       ; Load a byte & increase src address pointer
        STRB   r2, [r0], #1       ; Store a byte & increase dst address pointer
        CMP    r2, #0             ; Check for the null terminator
        BNE    loop               ; Copy the next byte if string is not ended
stop    B      stop               ; Dead loop. Embedded program never exits.

        ENDP

        AREA myData, DATA, READWRITE
        ALIGN

srcStr  DCB    "The source string.",0       ; Strings are null terminated
dstStr  DCB    "The destination string.",0  ; dststr has more space than srcstr

        END
```

Logic View of Memory

- By grouping bits together we can store more values
  - 8 bits = 1 **byte**
  - 16 bits = 2 bytes = 1 **halfword**
  - 32 bits = 4 bytes = 1 **word**
- From software perspective, memory is an addressable array of **bytes**.
  - The byte stored at the memory address `0x20000004` is `0b10000100`
  - A word can only be stored at an address that's divisible by 4 (Word-address mod 4 = 0, binary address ends with 00)
  - Memory address of a word is the lowest address of all 4 bytes in that word.
  - A halfword can only be stored at an address that's divisible by 2 (Halfword-address mod 2 = 0, binary address ends with 0)
  - Memory address of a halfword is the lowest address of all 2 bytes in that word.

0b10000100 → 0x84 → 132

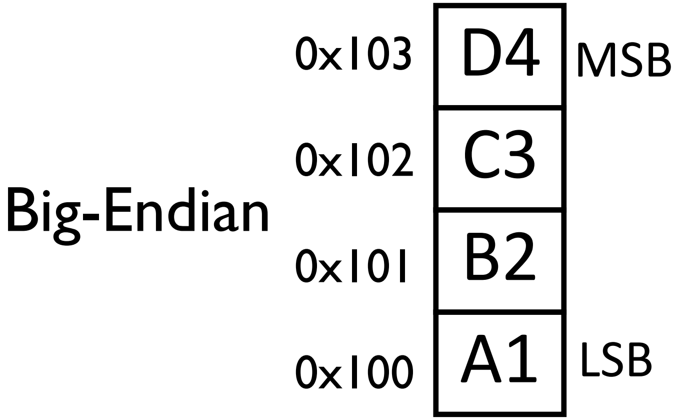
Binary Hexadecimal Decimal



0x20000007	01110010
0x20000006	00100101
0x20000005	11100010
0x20000004	10000100
0x20000003	01100001
0x20000002	10001111
0x20000001	00010010
0x20000000	10010100

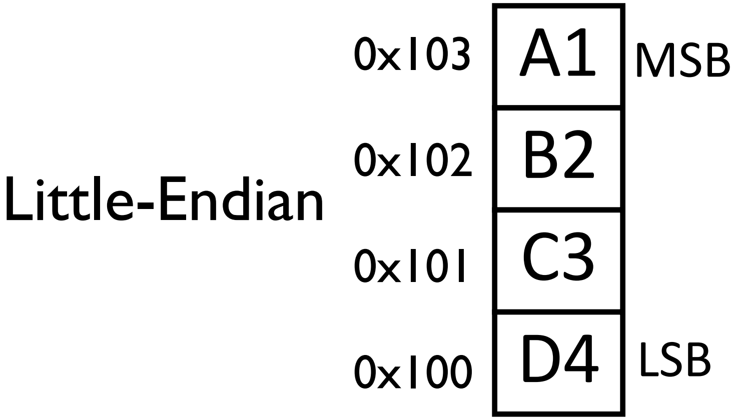
Memory Byte Ordering

- Two possible byte orderings:
  - Little-endian: the LSB (Least Significant Byte) is stored at the lowest address.
  - Big-endian: the LSB (Least Significant Byte) is stored at the highest address.
  - Intel processors use Little-Endian; ARM processors can be configured as either Little- or Big-endian; STM32 microcontrollers based on ARM Cortex-M3 use little-endian
- Example: 4-byte data 0xA1B2C3D4 at memory address 0x100
  - Memory address is Byte address, not bit address



Big-Endian

0x103	D4	MSB
0x102	C3	
0x101	B2	
0x100	A1	LSB



Little-Endian

0x103	A1	MSB
0x102	B2	
0x101	C3	
0x100	D4	LSB

Data Alignment

- Assume a byte-addressable memory with a data bus that is 32 bits (4 bytes) wide
- Consider 16 bytes of memory (addresses 0 to 15) arranged as four 32-bit words (4 bytes each)

Address 15	Address 14	Address 13	Address 12
Address 11	Address 10	Address 9	Address 8
Address 7 (MSbyte)	Address 6	Address 5	Address 4 (LSbyte)
Address 3	Address 2	Address 1	Address 0

Well-aligned: each word begins on a mod-4 address, which can be read in a single memory cycle

Address 15	Address 14	Address 13	Address 12
Address 11	Address 10	Address 9 (MSbyte)	Address 8
Address 7	Address 6 (LSbyte)	Address 5	Address 4
Address 3	Address 2	Address 1	Address 0

Ill-aligned: a word begins on address 6, not a mod-4 address, which can be read in 2 memory cycles

The first read cycle would retrieve 4 bytes from addresses 4 through 7; of these, the bytes from addresses 4 and 5 are discarded, and those from addresses 6 and 7 are moved to the far right;
The second read cycle retrieves 4 bytes from addresses 8 through 11; the bytes from addresses 10 and 11 are discarded, and those from addresses 8 and 9 are moved to the far left;
Finally, the two halves are combined to form the desired 32-bit operand:

		Address 7	Address 6 (LSbyte)

Address 9 (MSbyte)	Address 8		

Address 9 (MSbyte)	Address 8	Address 7	Address 6 (LSbyte)