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

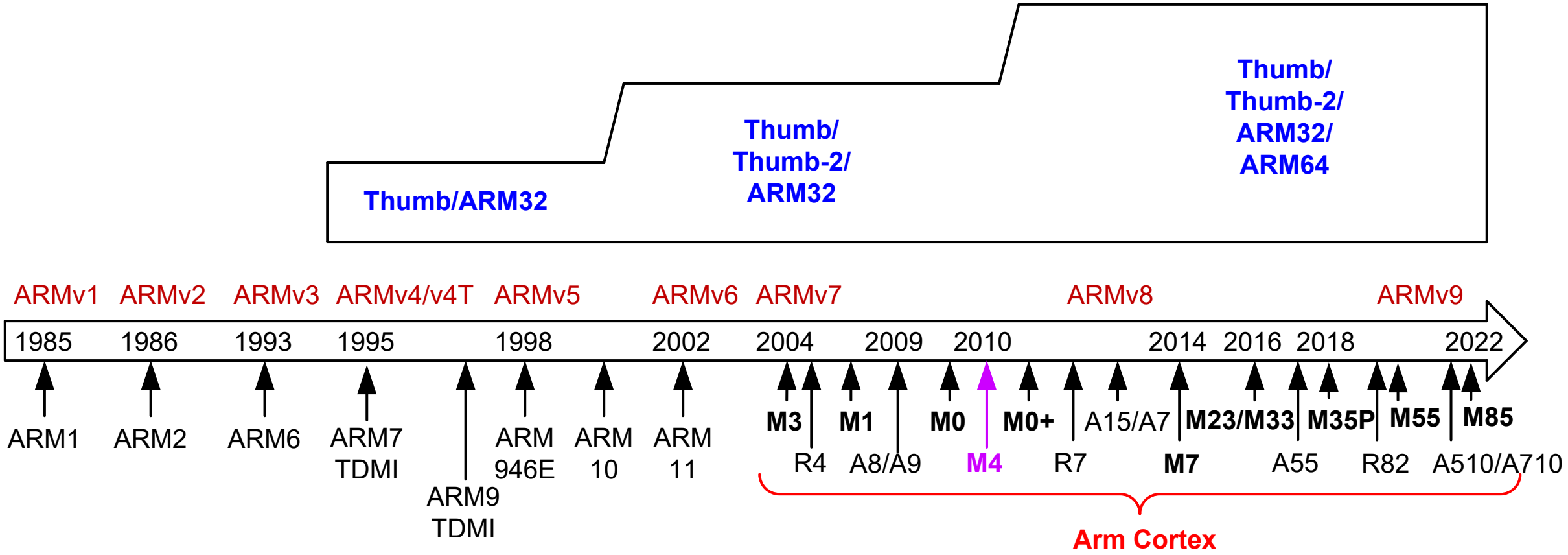
Chapter 3
ARM Instruction Set Architecture

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/

History

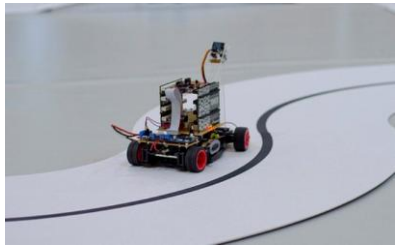

Thumb/ARM32
Thumb/
Thumb-2/
ARM32
Thumb/
Thumb-2/
ARM32/
ARM64
ARMv1
ARMv2
ARMv3
ARMv4/v4T
ARMv5
ARMv6
ARMv7
ARMv8
ARMv9
1985
1986
1993
1995
1998
2002
2004
2009
2010
2014
2016
2018
2022
ARM1
ARM2
ARM6
ARM7
TDMI
ARM9
TDMI
ARM
946E
ARM
10
ARM
11
M3
R4
M1
A8/A9
M0
M4
M0+
R7
A15/A7
M7
M23/M33
A55
M35P
R82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

ARM Processors

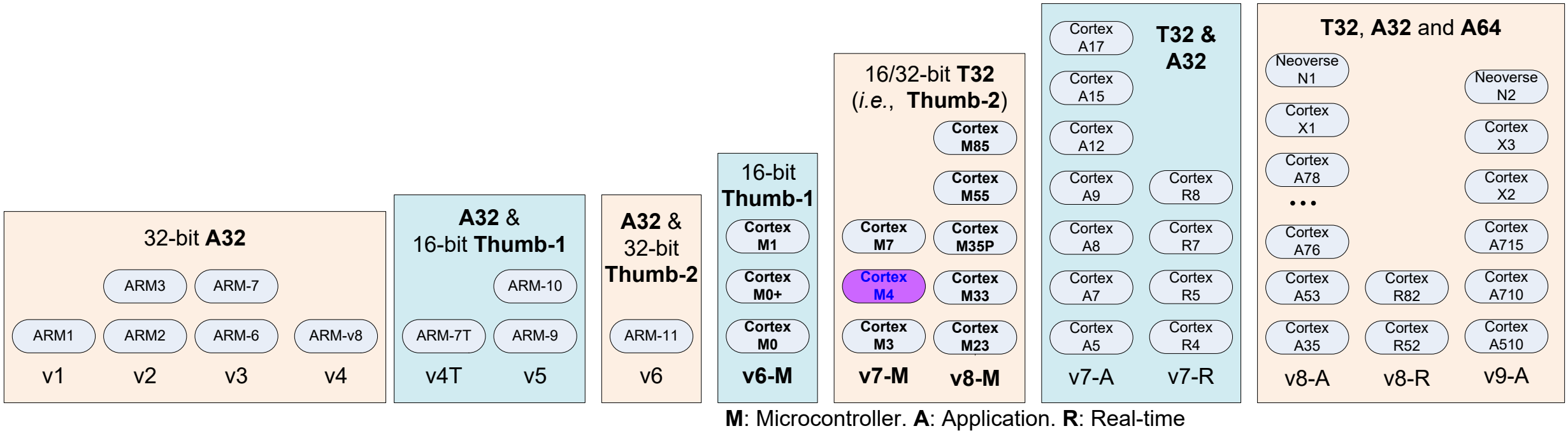
- ARM Cortex-A family:
  - **A**pplications processors
  - Support OS and high-performance applications
  - Such as Smartphones, Smart TV

- 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

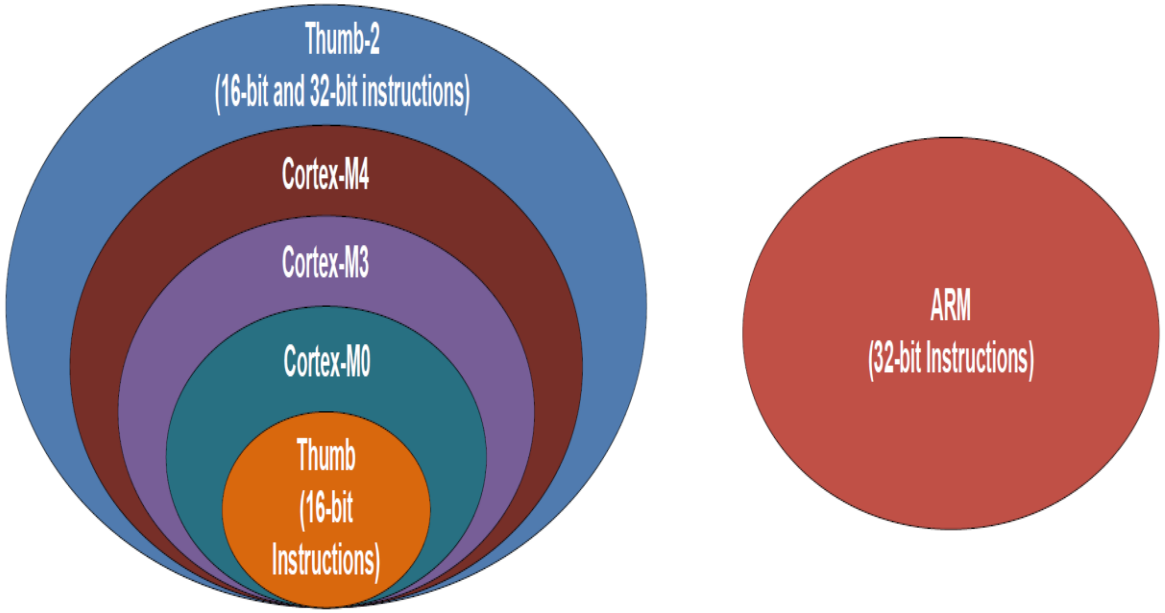


ARM Family



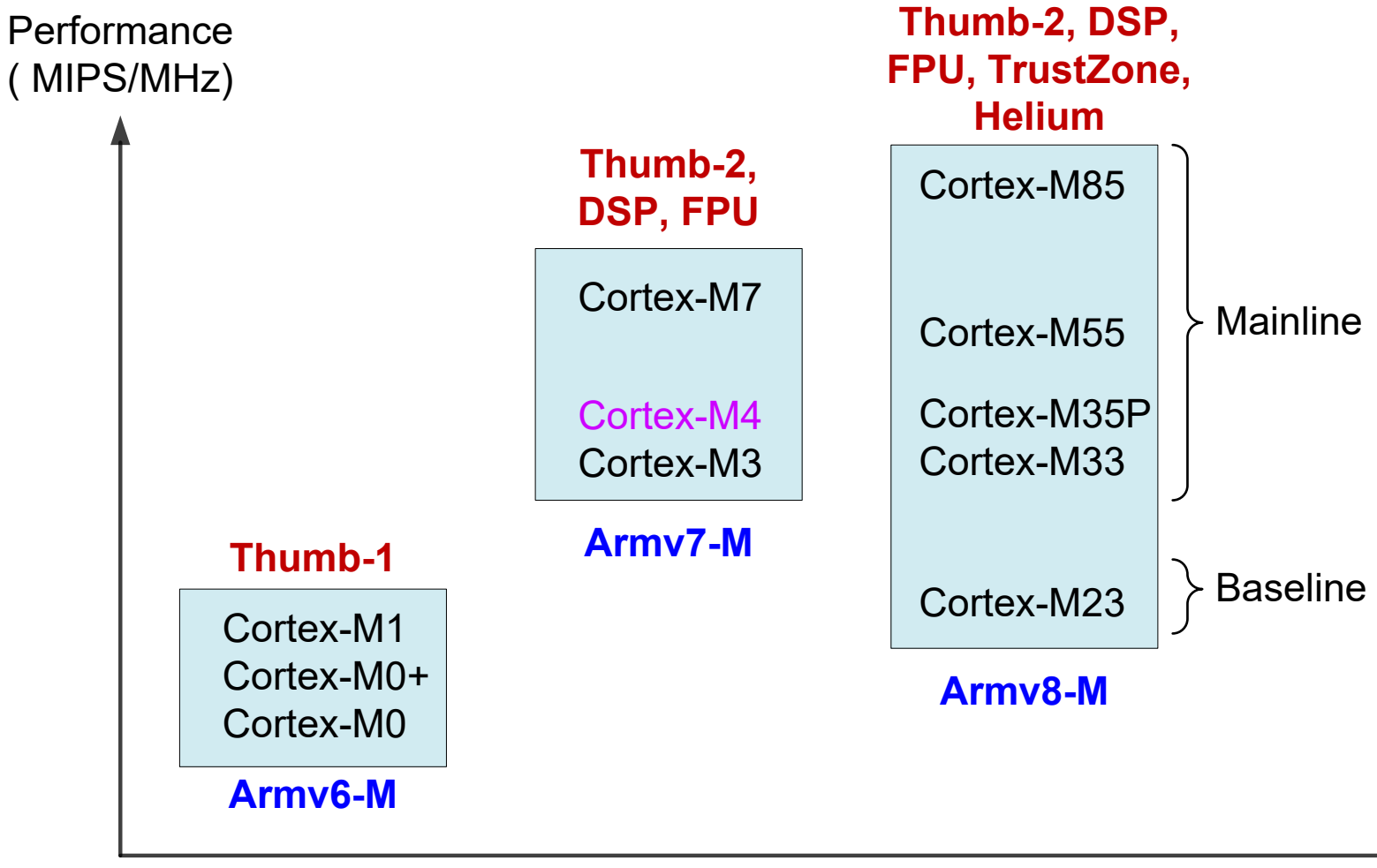
M: Microcontroller. **A**: Application. **R**: Real-time

Instruction Sets

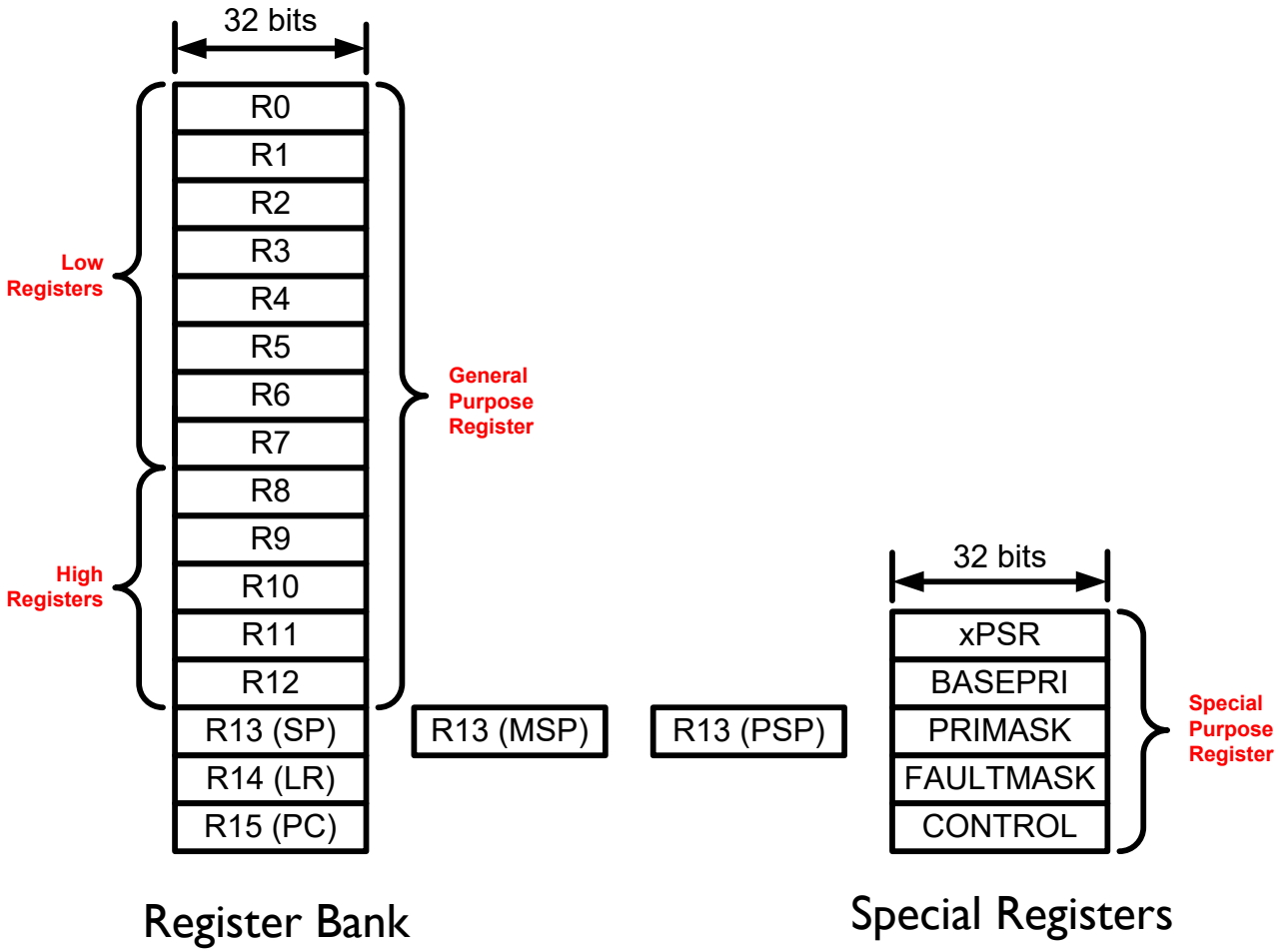


- Instructions:
  - Encoded to binary machine code by assembler
  - Executed at runtime by hardware
- Early 32-bit ARM vs Thumb/Thumb-2
  - Early ARM has larger power consumption and larger program size
  - 16-bit Thumb, first used in ARM7TDMI processors in 1995
  - Thumb-2: a mix of 16-bit (high code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

ARM Processors

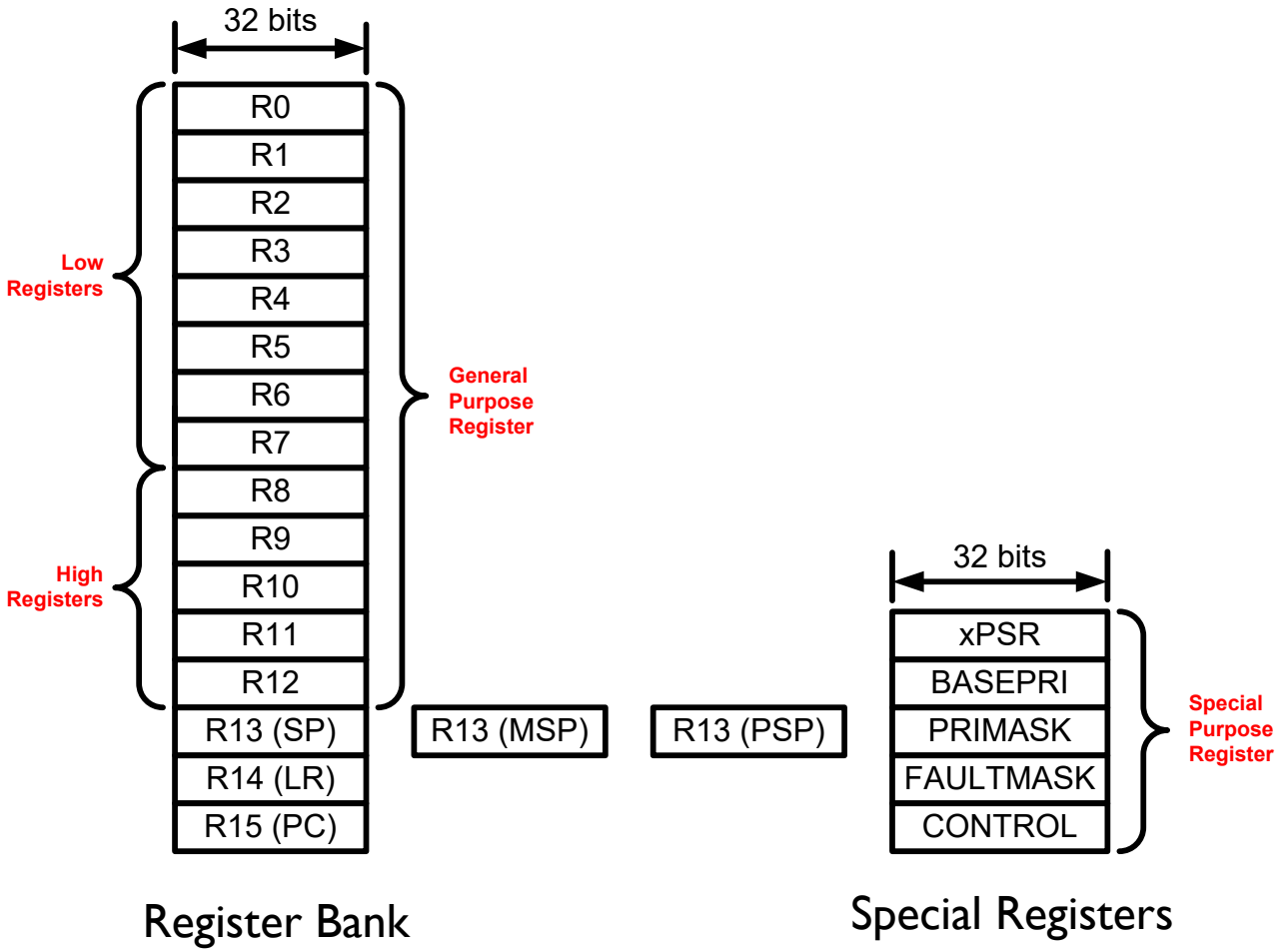


Processor Registers



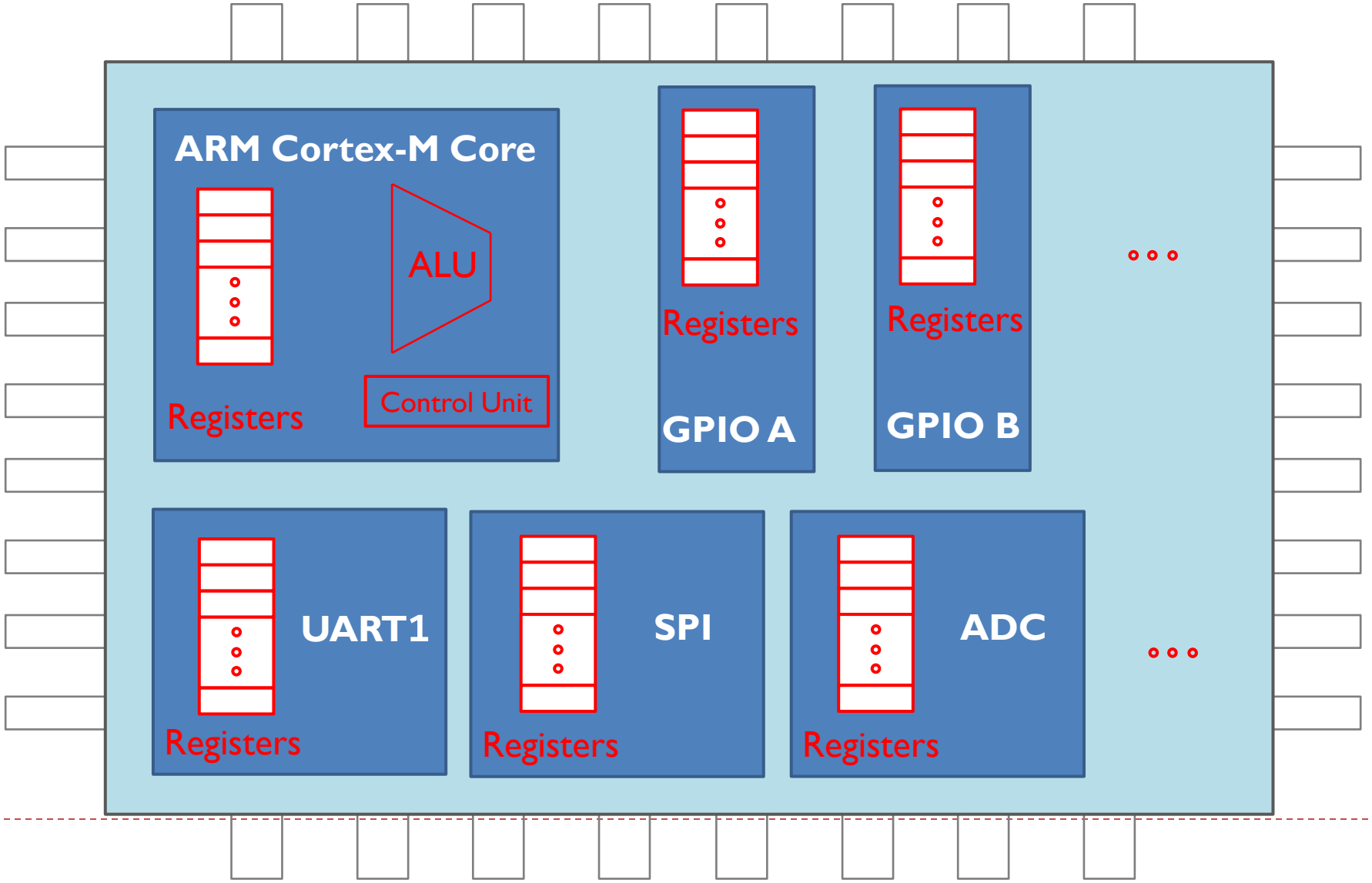
- Fastest way to read and write
- Registers are within the processor chip
- Each register has 32 bits
- ARM Cortex-M4 has
  - Register Bank: `R0 – R15`
    - **R0-R12**: 13 general-purpose registers
    - **R13**: Stack pointer (Shadow of MSP or PSP)
    - **R14**: Link register (LR)
    - **R15**: Program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

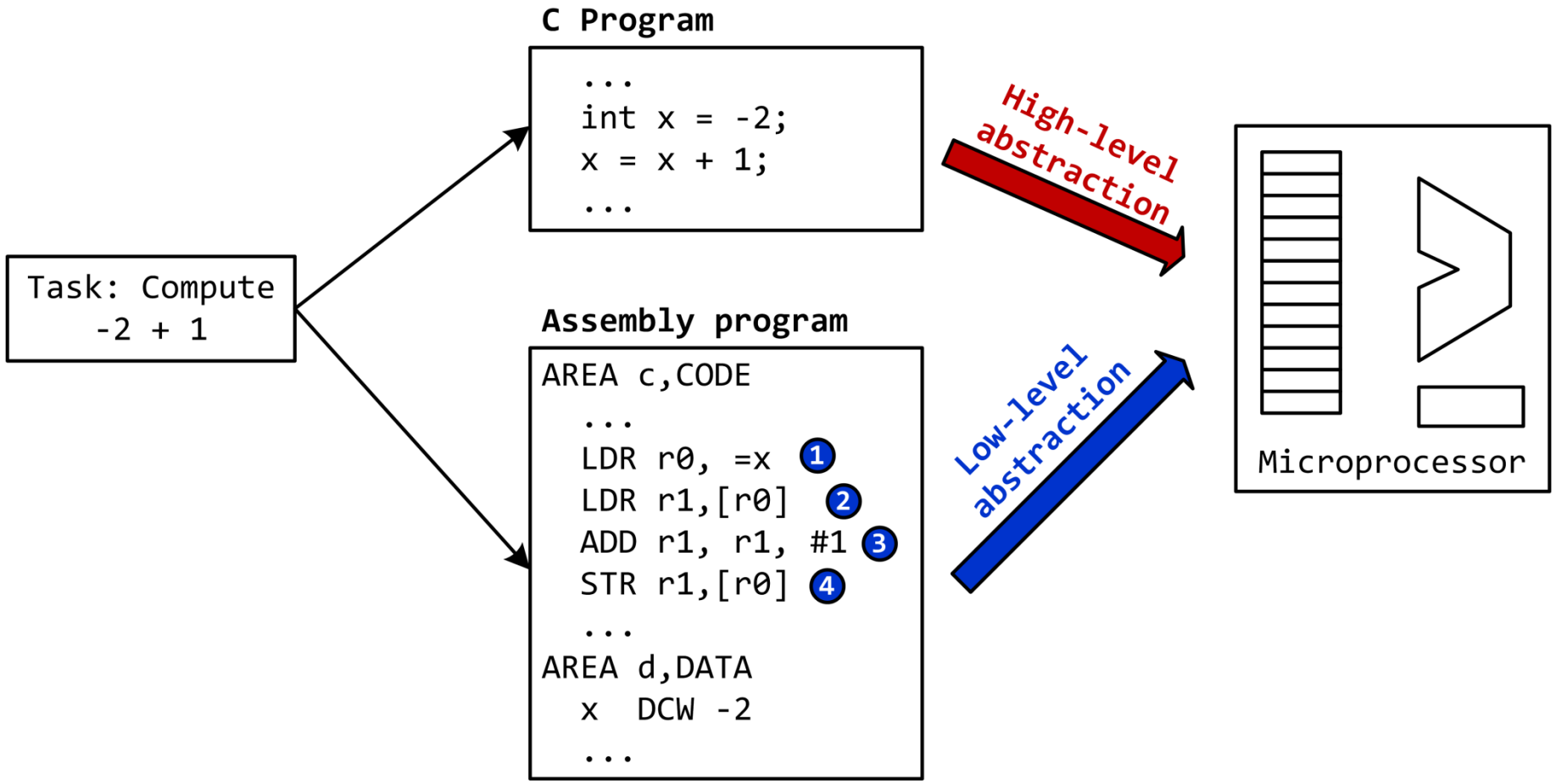


Processor Registers vs Peripheral Registers

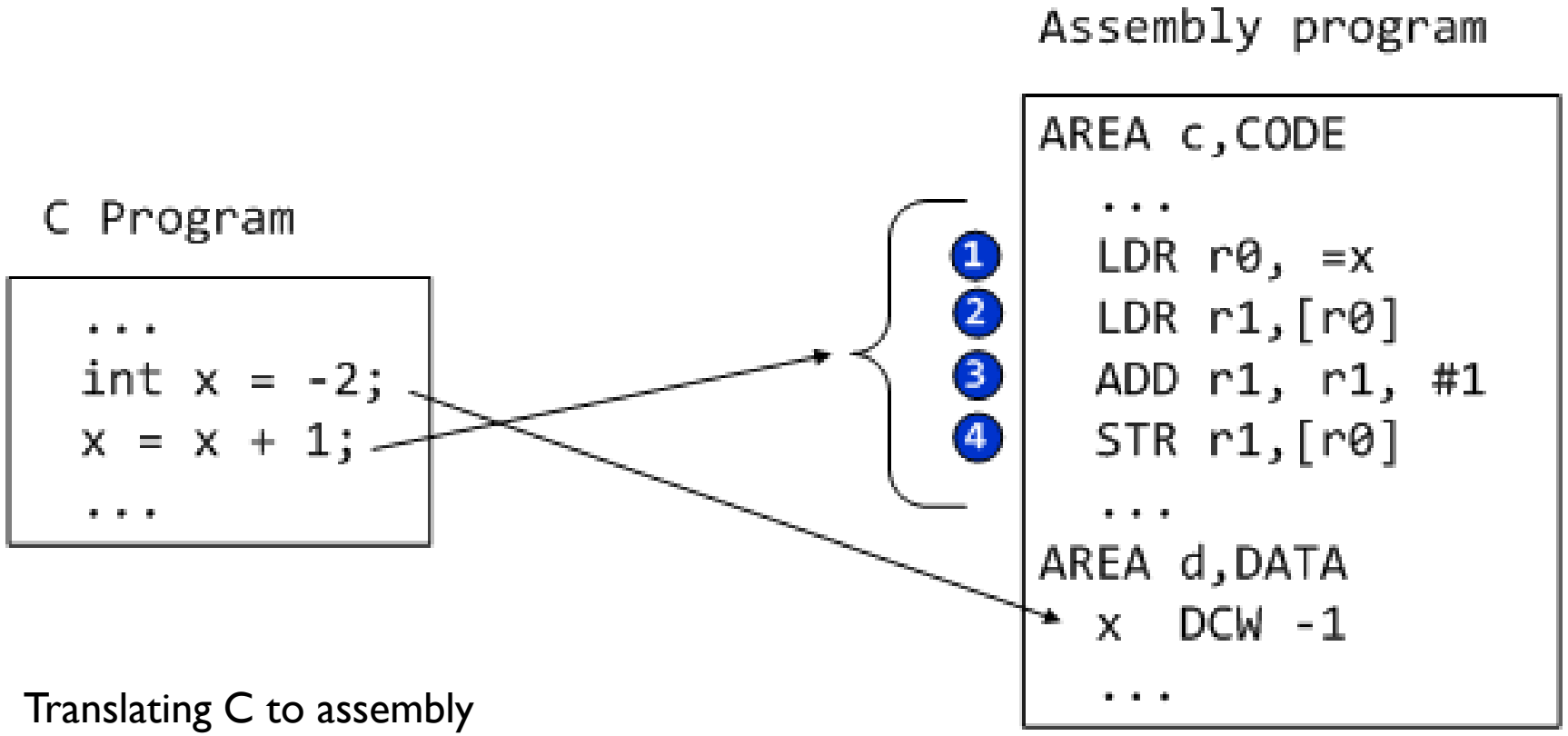
- Processor can directly access processor registers
  - `ADD r3,r1,r0   ; r3 = r1 + r0`

- 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



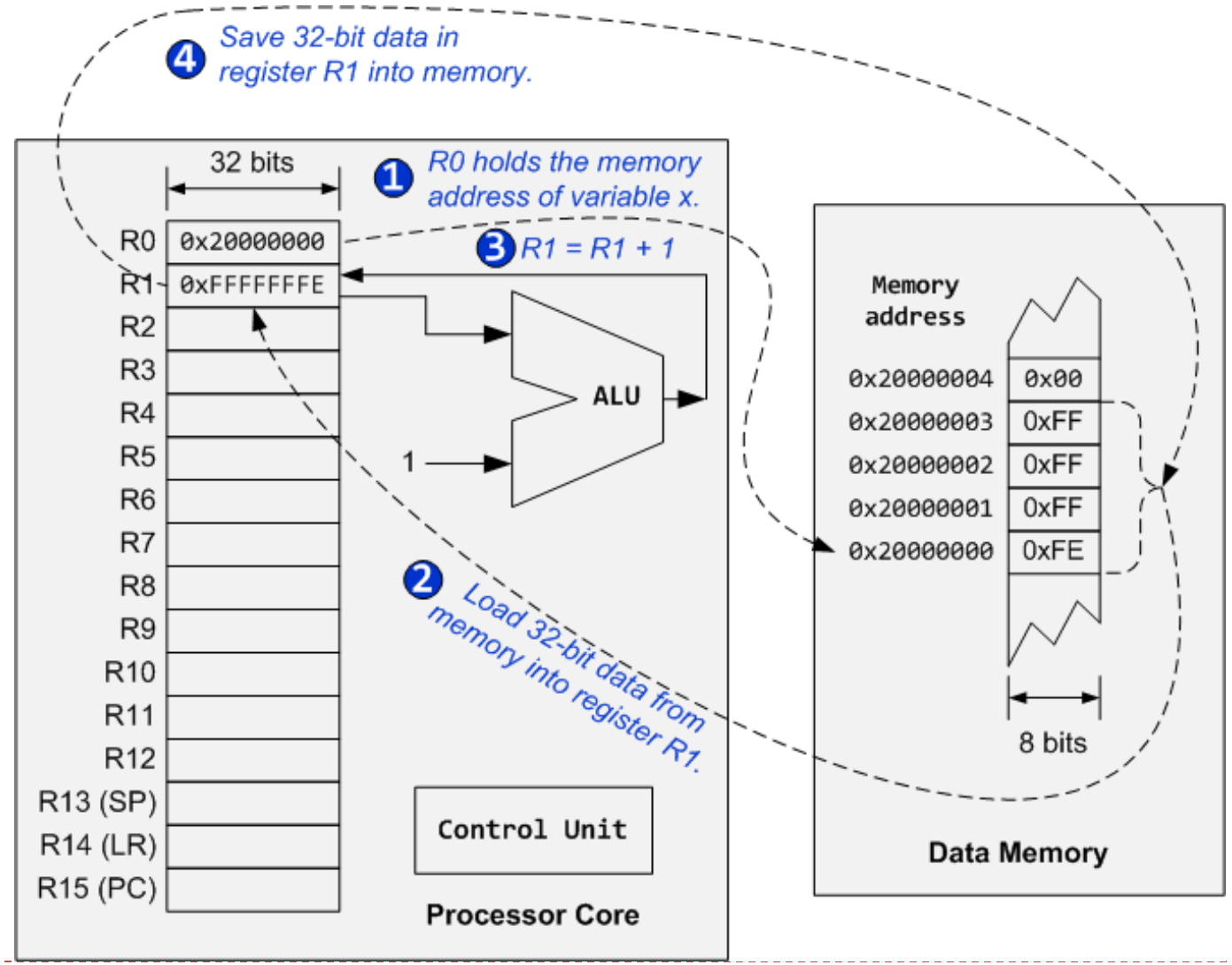
Load-Modify-Store



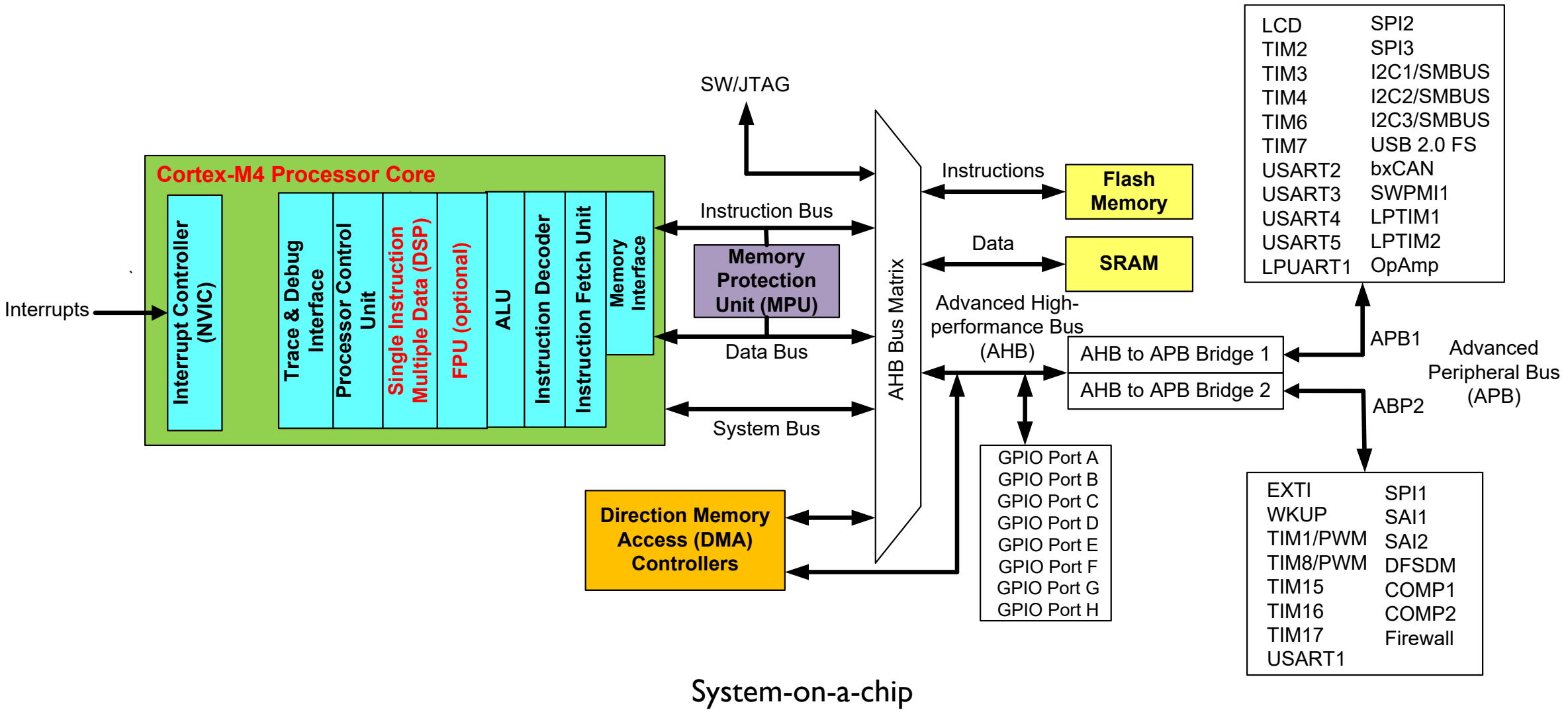
Translating C to assembly

- 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).

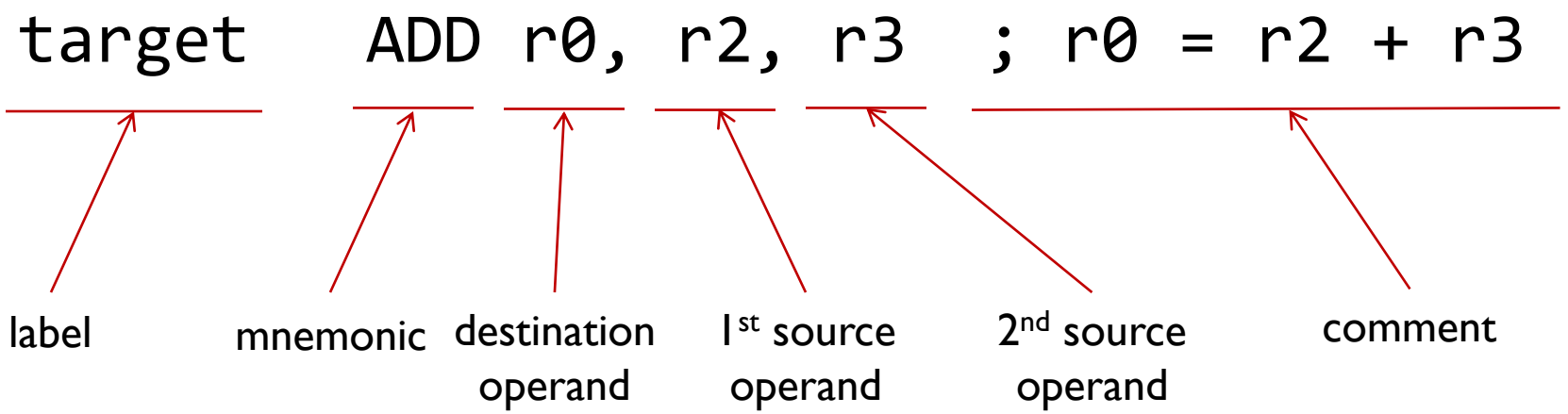
Instruction Format: Comments

```
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
```



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
```

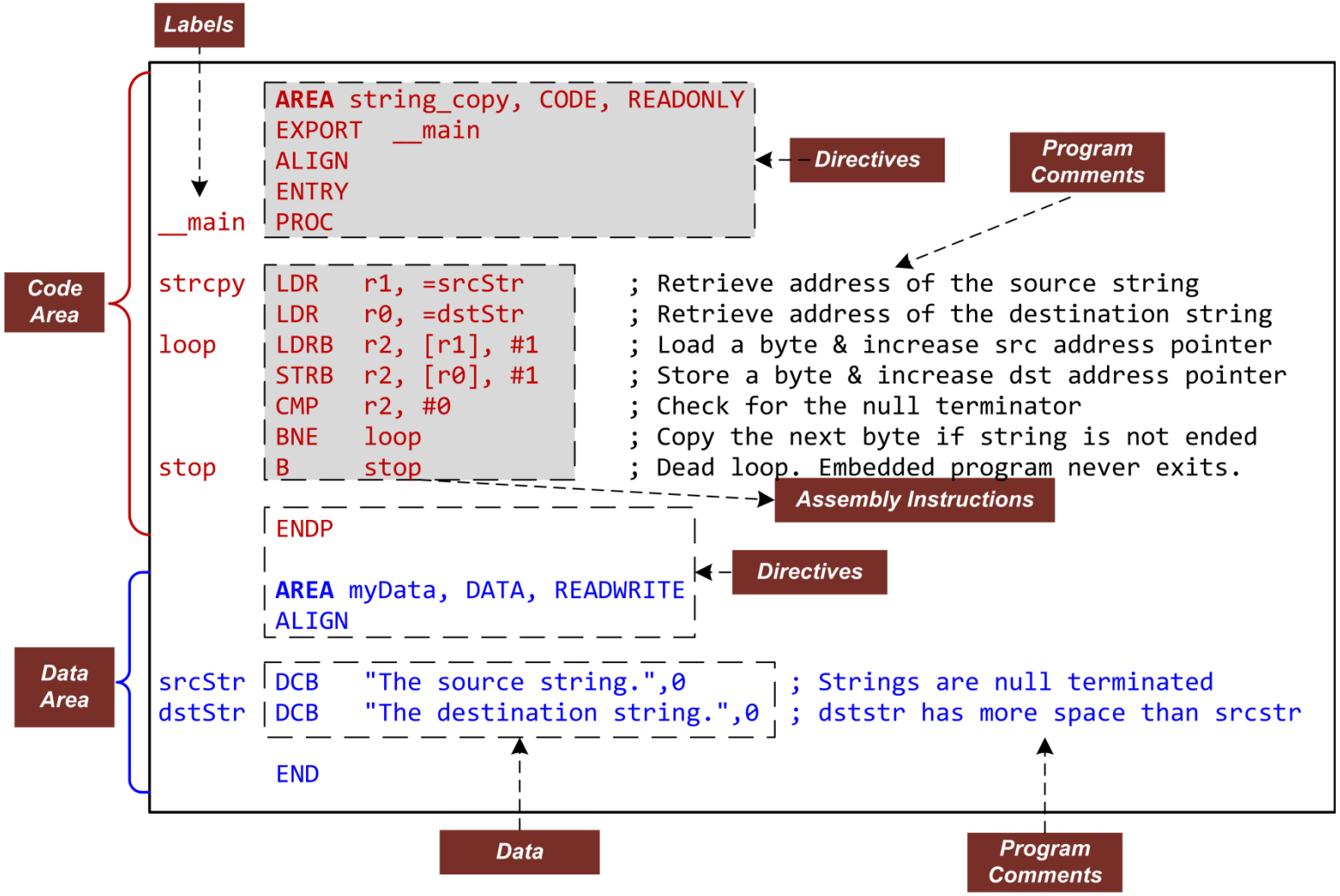
Example Assembly Program: Copying a String

Code Area

Data Area

```
        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



Assembly Directives

- Directives are **NOT** instructions. Instead, they are used to provide key information for assembly.

`AREA`	Make a new block of data or code
`ENTRY`	Declare an entry point where the program execution starts
`ALIGN`	Align data or code to a particular memory boundary
`DCB`	Allocate one or more bytes (8 bits) of data
`DCW`	Allocate one or more half-words (16 bits) of data
`DCD`	Allocate one or more words (32 bits) of data
`SPACE`	Allocate a zeroed block of memory with a particular size
`FILL`	Allocate a block of memory and fill with a given value.
`EQU`	Give a symbol name to a numeric constant
`RN`	Give a symbol name to a register
`EXPORT`	Declare a symbol and make it referable by other source files
`IMPORT`	Provide a symbol defined outside the current source file
`INCLUDE/GET`	Include a separate source file within the current source file
`PROC`	Declare the start of a procedure
`ENDP`	Designate the end of a procedure
`END`	Designate the end of a source file

Directive: AREA

```
          AREA myData, DATA, READWRITE ; Define a data section
Array     DCD 1, 2, 3, 4, 5            ; Define an array with five integers

          AREA myCode, CODE, READONLY  ; Define a code section
          EXPORT  __main               ; Make __main visible to the linker
          ENTRY                        ; Mark the entrance to the entire program
__main    PROC                         ; PROC marks the begin of a subroutine
          ...                          ; Assembly program starts here.
          ENDP                         ; Mark the end of a subroutine
          END                          ; Mark the end of a program
```

- The AREA directive indicates to the assembler the start of a new data or code section.
- Areas are the basic independent and indivisible unit processed by the **linker**.
- Each area is identified by a name and areas within the same source file **cannot share the same name.**
- An assembly program must have **at least one code area**.
- By default, a code area can only be read (READONLY) and a data area may be read from and written to (READWRITE).

Directive: ENTRY

```
          AREA myData, DATA, READWRITE ; Define a data section
Array     DCD 1, 2, 3, 4, 5            ; Define an array with five integers

          AREA myCode, CODE, READONLY  ; Define a code section
          EXPORT  __main               ; Make __main visible to the linker
          ENTRY                        ; Mark the entrance to the entire program
__main    PROC                         ; PROC marks the begin of a subroutine
          ...                          ; Assembly program starts here.
          ENDP                         ; Mark the end of a subroutine
          END                          ; Mark the end of a program
```

- The ENTRY directive marks **the start point** to execute a program.
- There must be **exactly one** ENTRY directive in an application, no matter how many source files the application has.

Directive: END

```
          AREA myData, DATA, READWRITE ; Define a data section
Array     DCD 1, 2, 3, 4, 5            ; Define an array with five integers

          AREA myCode, CODE, READONLY  ; Define a code section
          EXPORT  __main               ; Make __main visible to the linker
          ENTRY                        ; Mark the entrance to the entire program
__main    PROC                         ; PROC marks the begin of a subroutine
          ...                          ; Assembly program starts here.
          ENDP                         ; Mark the end of a subroutine
          END                          ; Mark the end of a program
```

- The END directive indicates the end of a source file.
- Each assembly program must end with this directive.

Directive: PROC and ENDP

```
          AREA myData, DATA, READWRITE ; Define a data section
Array     DCD 1, 2, 3, 4, 5            ; Define an array with five integers

          AREA myCode, CODE, READONLY  ; Define a code section
          EXPORT  __main               ; Make __main visible to the linker
          ENTRY                        ; Mark the entrance to the entire program
__main    PROC                         ; PROC marks the begin of a subroutine
          ...                          ; Assembly program starts here.
          ENDP                         ; Mark the end of a subroutine
          END                          ; Mark the end of a program
```

- PROC and ENDP are to mark the start and end of a function (also called subroutine or procedure).
- A single source file can contain multiple subroutines, with each of them defined by a pair of PROC and ENDP.
- PROC and ENDP cannot be nested. We cannot define a function within another function.

Directive: EXPORT and IMPORT

```
          AREA myData, DATA, READWRITE ; Define a data section
Array     DCD 1, 2, 3, 4, 5            ; Define an array with five integers

          AREA myCode, CODE, READONLY  ; Define a code section
          EXPORT  __main               ; Make __main visible to the linker
          ENTRY                        ; Mark the entrance to the entire program
__main    PROC                         ; PROC marks the begin of a subroutine
          ...                          ; Assembly program starts here.
          ENDP                         ; Mark the end of a subroutine
          END                          ; Mark the end of a program
```

- The EXPORT declares a symbol and makes this **symbol visible** to the linker.
- The IMPORT gives the assembler a symbol that is **not defined locally** in the current assembly file. The symbol must be defined in another file.
- The IMPORT is similar to the "extern" keyword in C.

Directive: Defining Data

Directive	Description	Memory Space
DCB	Define Constant Byte	Reserve 8-bit values
DCW	Define Constant Half-word	Reserve 16-bit values
DCD	Define Constant Word	Reserve 32-bit values
DCQ	Define Constant	Reserve 64-bit values
DCFS	Define single-precision floating-point numbers	Reserve 32-bit values
DCFD	Define double-precision floating-point numbers	Reserve 64-bit values
SPACE	Defined Zeroed Bytes	Reserve a number of zeroed bytes
FILL	Defined Initialized Bytes	Reserve and fill each byte with a value

Directive: Defining Data

```
 AREA    myData, DATA, READWRITE
hello    DCB    "Hello World!",0   ; Allocate a string that is null-terminated
dollar   DCB    2,10,0,200         ; Allocate integers ranging from -128 to 255
scores   DCD    2,3.5,-0.8,4.0     ; Allocate 4 words containing decimal values
miles    DCW    100,200,50,0       ; Allocate integers between –32768 and 65535
Pi_S     DCFS   3.14               ; Allocate a single-precision floating number
Pi_D     DCFD   3.14               ; Allocate a double-precision floating number
p        SPACE  255                ; Allocate 255 bytes of zeroed memory space
f        FILL   20,0xFF,1          ; Allocate 20 bytes and set each byte to 0xFF
binary   DCB    2_01010101         ; Allocate a byte in binary
octal    DCB    8_73               ; Allocate a byte in octal
char     DCB    ‘A’                ; Allocate a byte initialized to ASCII of ‘A’
```

Directive: EQU and RN

```
; Interrupt Number Definition (IRQn)
BusFault_IRQn    EQU  -11        ; Cortex-M3 Bus Fault Interrupt
SVCall_IRQn      EQU   -5        ; Cortex-M3 SV Call Interrupt
PendSV_IRQn      EQU   -2        ; Cortex-M3 Pend SV Interrupt
SysTick_IRQn     EQU   -1        ; Cortex-M3 System Tick Interrupt

Dividend         RN    6         ; Defines dividend for register 6
Divisor          RN    5         ; Defines divisor for register 5
```

- The EQU directive associates a symbolic name to a numeric constant.
- Similar to the use of `#define` in a C program, the EQU can be used to define a constant in an assembly code.
- The RN directive gives a symbolic name to a specific register.

Directive: ALIGN

```
    AREA example, CODE, ALIGN = 3   ; Memory address begins at a multiple of 8
    ADD r0, r1, r2                  ; Instructions start at a multiple of 8

    AREA myData, DATA, ALIGN = 2    ; Address starts at a multiple of four
a   DCB 0xFF                        ; The first byte of a 4-byte word
    ALIGN 4, 3                      ; Align to the last byte (3) of a word (4)
b   DCB 0x33                        ; Set the fourth byte of a 4-byte word
c   DCB 0x44                        ; Add a byte to make next data misaligned
    ALIGN                           ; Force the next data to be aligned
d   DCD 12345                       ; Skip three bytes and store the word
```

Directive: INCLUDE or GET

```
        INCLUDE constants.s       ; Load Constant Definitions
        AREA main, CODE, READONLY
        EXPORT  __main
        ENTRY
__main  PROC
        ...
        ENDP
        END
```

- The INCLUDE or GET directive is to include an assembly source file within another source file.
- It is useful to include constant symbols defined by using EQU and stored in a separate source file.