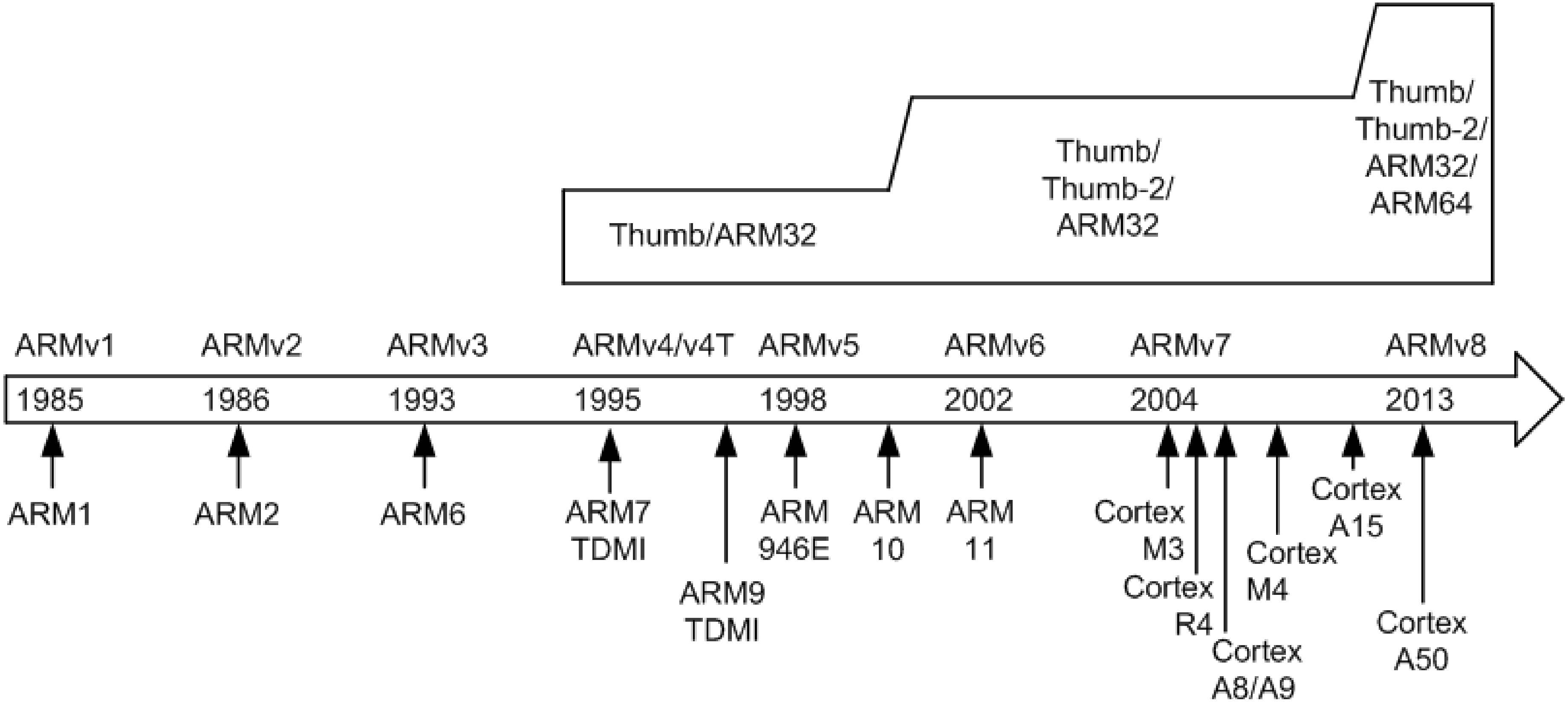
Embedded Systems with ARM Cortex-M3
Microcontrollers in Assembly Language and C

Chapter 3
ARM Instruction Set Architecture

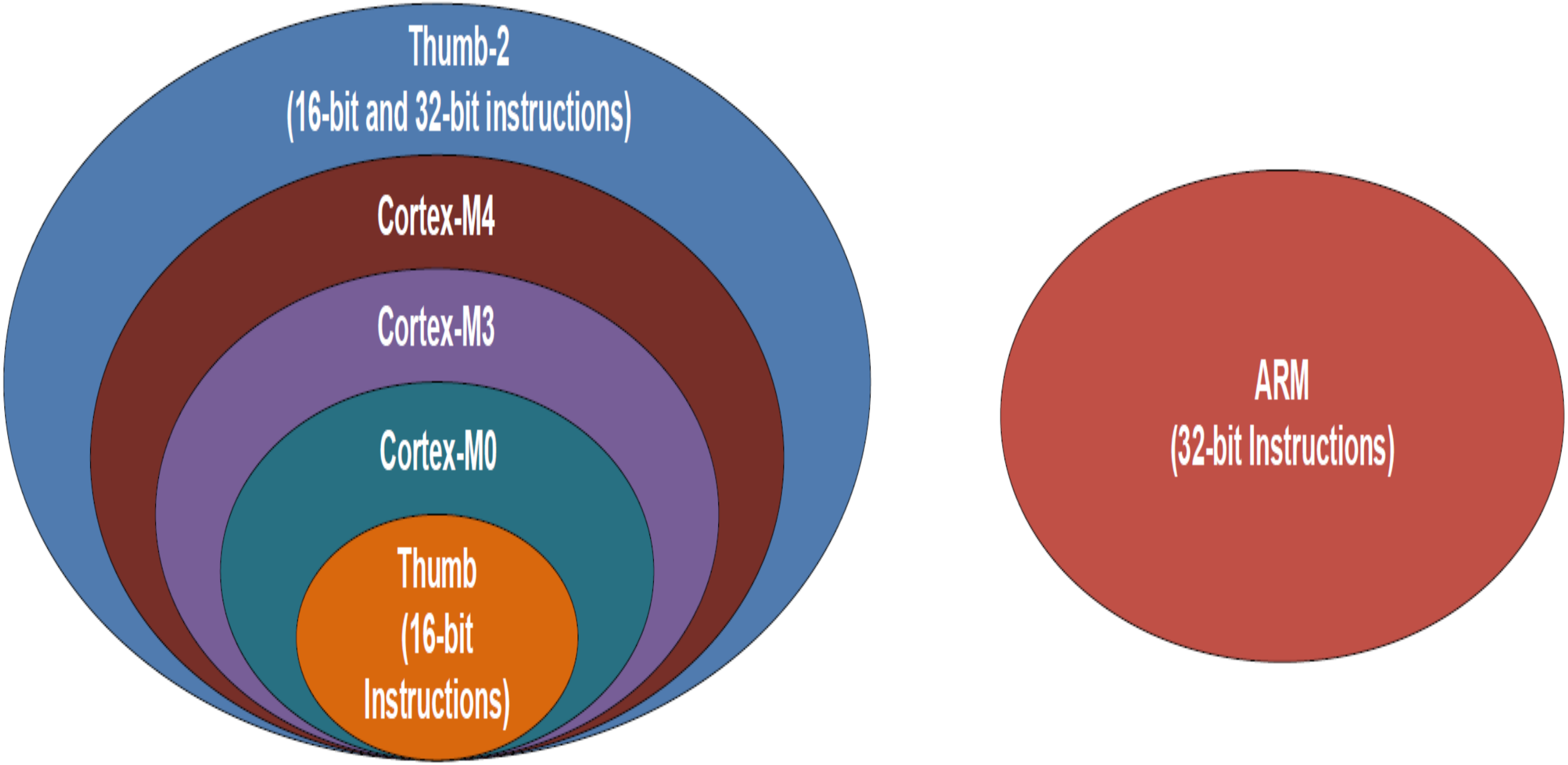
Dr. Yifeng Zhu
Electrical and Computer Engineering
University of Maine

Spring 2015

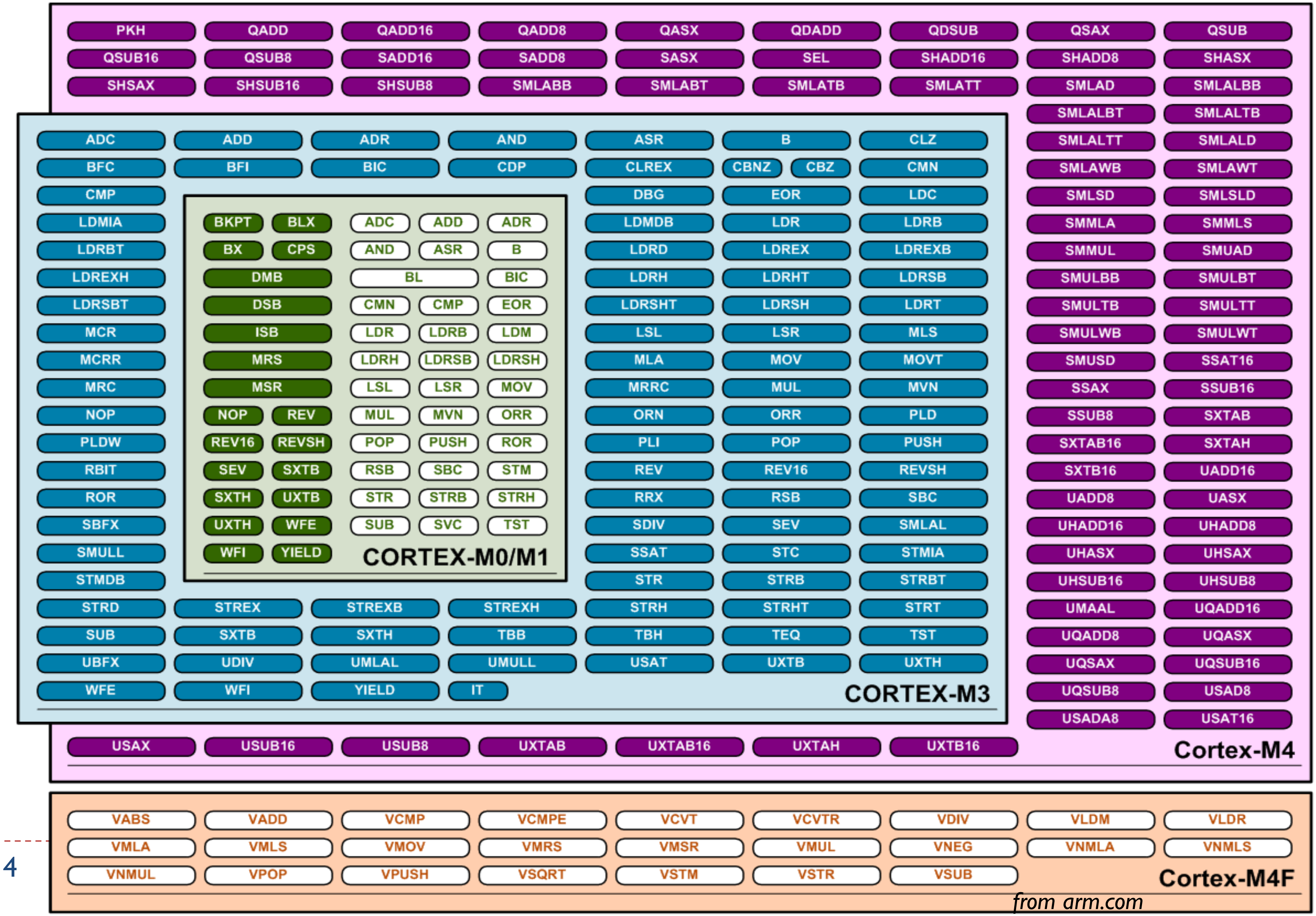
History


Thumb/ARM32
Thumb/ Thumb-2/ ARM32
Thumb/ Thumb-2/ ARM32/ ARM64
ARMv1
1985
ARMv2
1986
ARMv3
1993
ARMv4/v4T
1995
ARMv5
1998
ARMv6
2002
ARMv7
2004
ARMv8
2013
ARM1
ARM2
ARM6
ARM7 TDMI
ARM9 TDMI
ARM 946E
ARM 10
ARM 11
Cortex M3
Cortex R4
Cortex A8/A9
Cortex M4
Cortex A15
Cortex A50

Instruction Sets



Instruction Sets



PKH QADD QADD16 QADD8 QASX QDADD QDSUB QSAX QSUB
QSUB16 QSUB8 SADD16 SADD8 SASX SEL SHADD16 SHADD8 SHASX
SHSAX SHSUB16 SHSUB8 SMLABB SMLABT SMLATB SMLATT SMLAD SMLALBB
SMLALBT SMLALTB
SMLALTT SMLALD
SMLAWB SMLAWT
SMLSD SMLSLD
SMMLA SMMLS
SMMUL SMUAD
SMULBB SMULBT
SMULTB SMULTT
SMULWB SMULWT
SMUSD SSAT16
SSAX SSUB16
SSUB8 SXTAB
SXTAB16 SXTAH
SXTB16 UADD16
UADD8 UASX
UHADD16 UHADD8
UHASX UHSAX
UHSUB16 UHSUB8
UMAAL UQADD16
UQADD8 UQASX
UQSAX UQSUB16
UQSUB8 USAD8
USADA8 USAT16
USAX USUB16 USUB8 UXTAB UXTAB16 UXTAH UXTB16
Cortex-M4

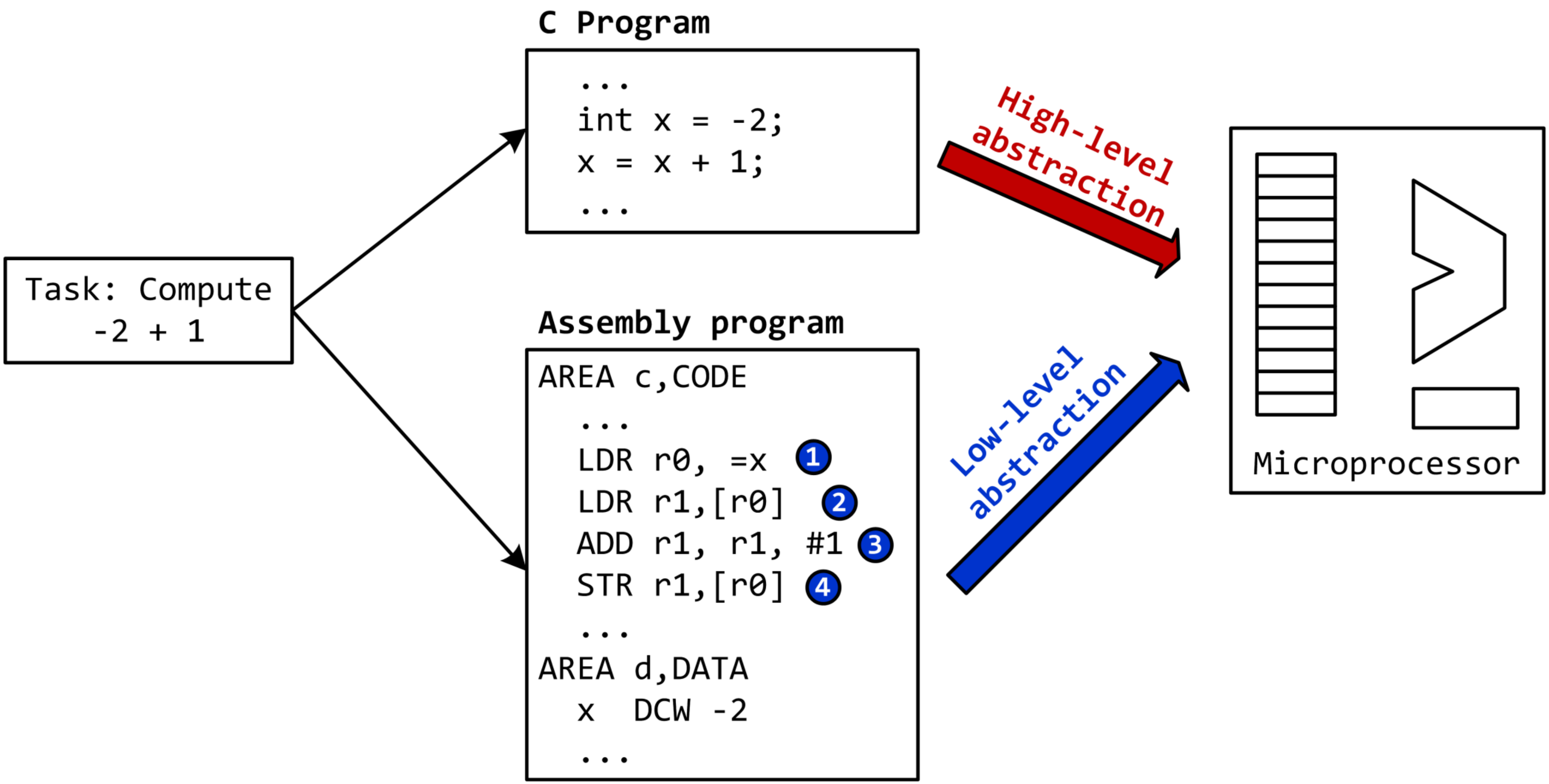
ADC ADD ADR AND ASR B CLZ
BFC BFI BIC CDP CLREX CBNZ CBZ CMN
CMP
LDMIA
LDRBT
LDREXH
LDRSBT
MCR
MCRR
MRC
NOP
PLDW
RBIT
ROR
SBFX
SMULL
STMDB
DBG EOR LDC
LDMDB LDR LDRB
LDRD LDREX LDREXB
LDRH LDRHT LDRSB
LDRSHT LDRSH LDRT
LSL LSR MLS
MLA MOV MOVT
MRRC MUL MVN
ORN ORR PLD
PLI POP PUSH
REV REV16 REVSH
RRX RSB SBC
SDIV SEV SMLAL
SSAT STC STMIA
STR STRB STRBT
STRD STREX STREXB STREXH STRH STRHT STRT
SUB SXTB SXTH TBB TBH TEQ TST
UBFX UDIV UMLAL UMULL USAT UXTB UXTH
WFE WFI YIELD IT
CORTEX-M3

BKPT BLX ADC ADD ADR
BX CPS AND ASR B
DMB BL BIC
DSB CMN CMP EOR
ISB LDR LDRB LDM
MRS LDRH LDRSB LDRSH
MSR LSL LSR MOV
NOP REV MUL MVN ORR
REV16 REVSH POP PUSH ROR
SEV SXTB RSB SBC STM
SXTH UXTB STR STRB STRH
UXTH WFE SUB SVC TST
WFI YIELD
CORTEX-M0/M1

VABS VADD VCMP VCMPE VCVT VCVTR VDIV VLDM VLDR
VMLA VMLS VMOV VMRS VMSR VMUL VNEG VNMLA VNMLS
VNMUL VPOP VPUSH VSQRT VSTM VSTR VSUB
Cortex-M4F

from arm.com

From C to Assembly



Task: Compute -2 + 1

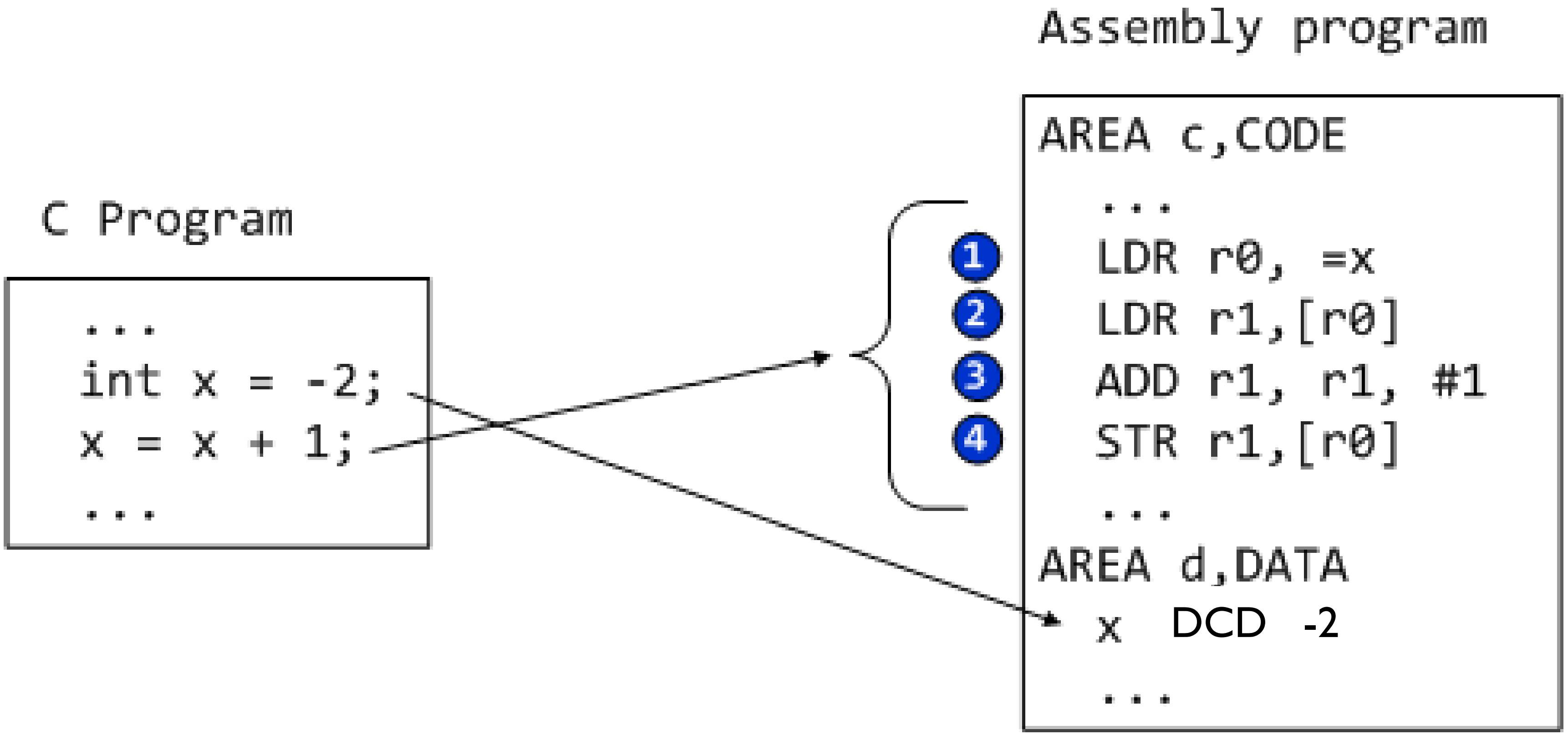
C Program

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

Assembly program

```
AREA c,CODE
  ...
  LDR r0, =x ❶
  LDR r1,[r0]  ❷
  ADD r1, r1, #1 ❸
  STR r1,[r0] ❹
  ...
AREA d,DATA
  x  DCW -2
  ...
```

Load-Modify-Store



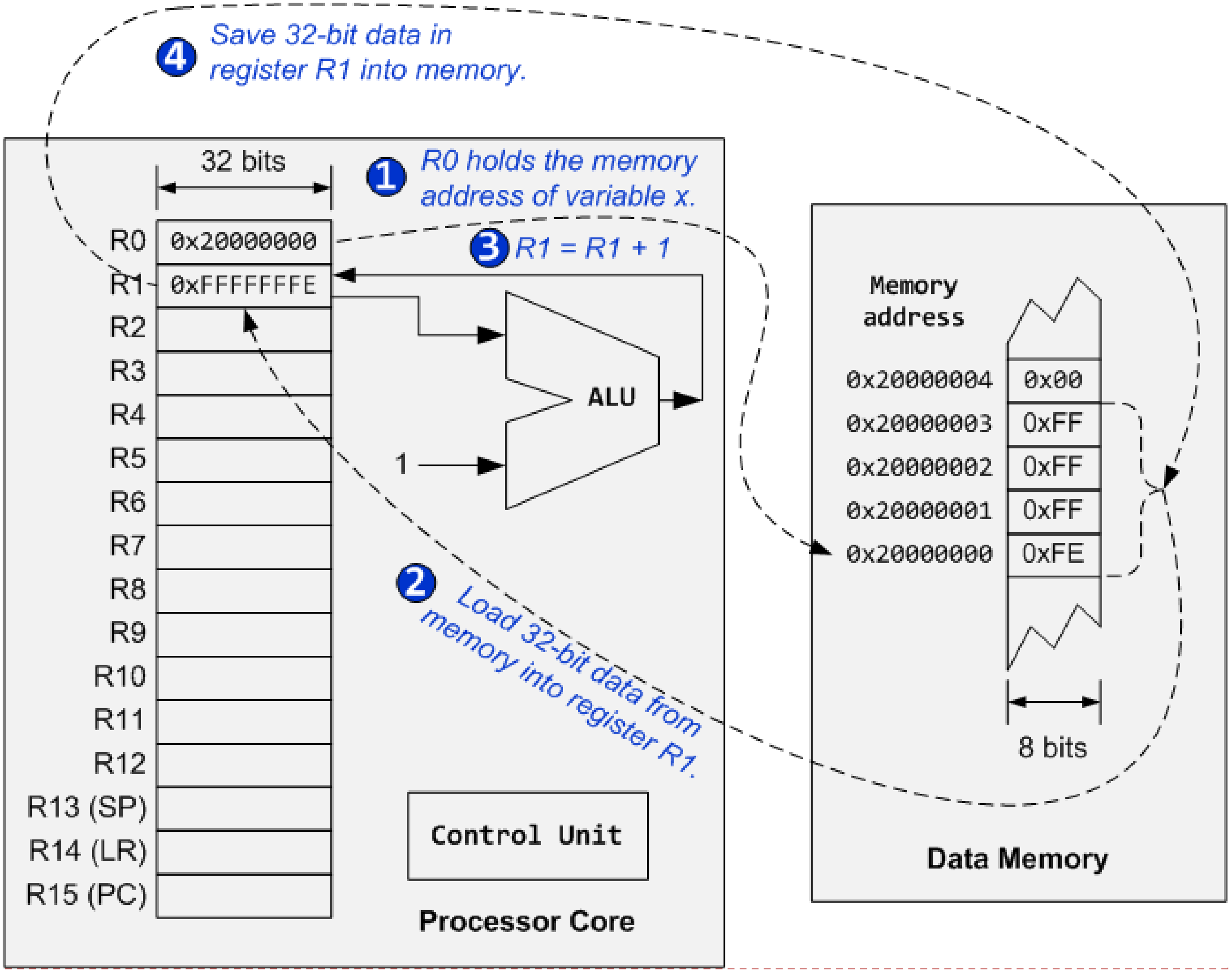
C Program

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

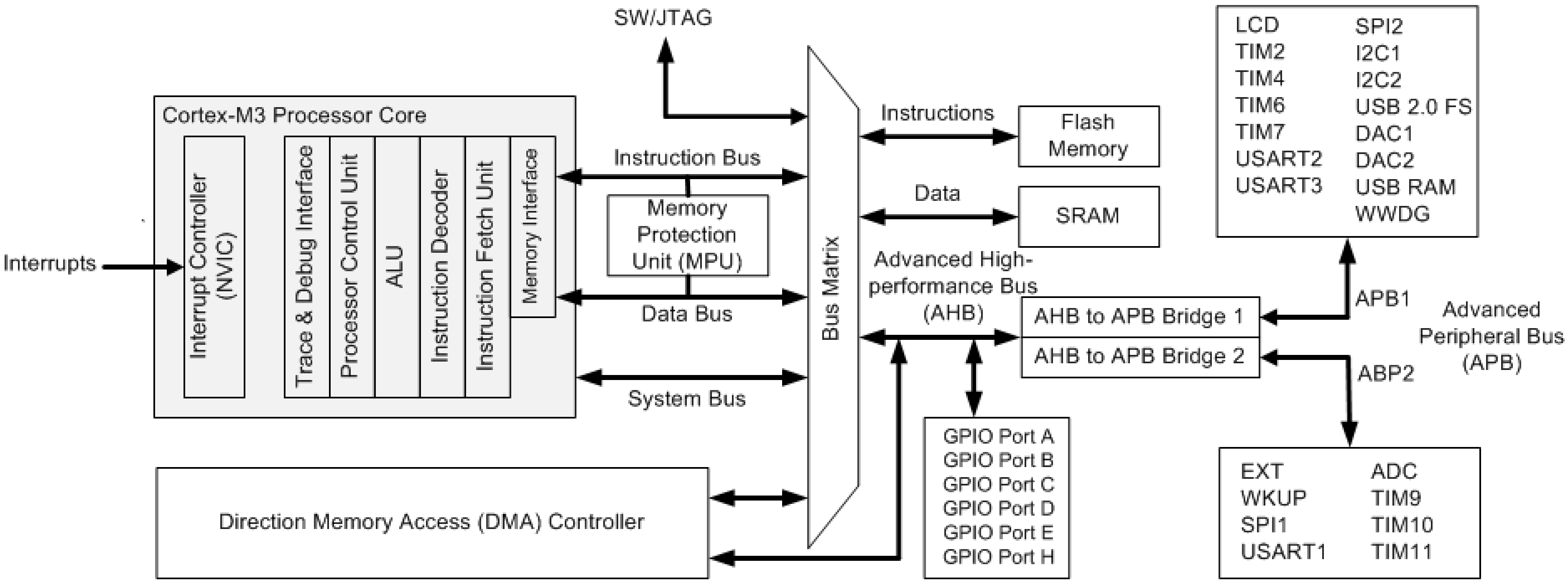
Assembly program

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

Load-Modify-Store



ARM Cortex-M3 Organization



System-on-a-chip

Assembly Instructions Supported

- 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

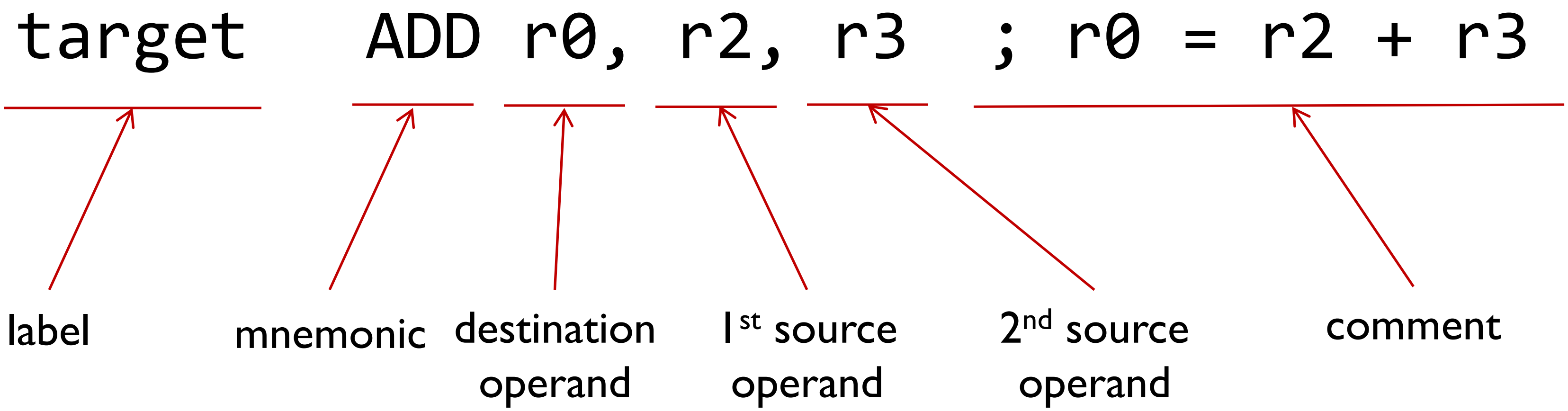
ARM Instruction Format

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

- Label is a reference to the memory address of this instruction.
- Mnemonic represents the operation to be performed.
- The number of operands varies, depending on each specific instruction. Some instructions have no operands at all.
  - Typically, operand1 is the destination register, and operand2 and operand3 are source operands.
  - operand2 is usually a register.
  - operand3 may be a register, an immediate number, a register shifted to a constant amount of bits, or a register plus an offset (used for memory access).
- Everything after the semicolon ";" is a comment, which is an annotation explicitly declaring 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

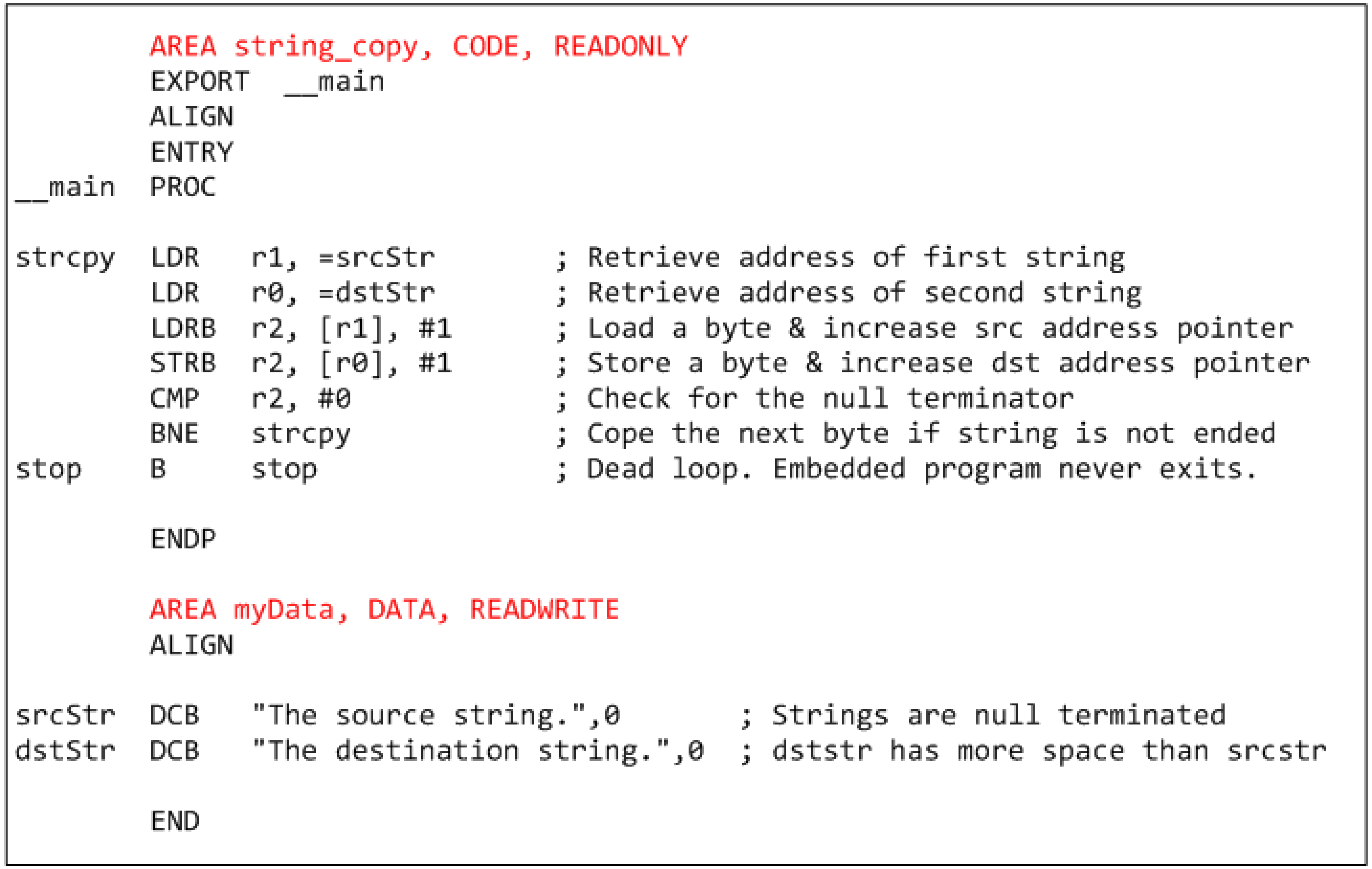
ARM Instruction Format

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

Examples: Variants of the ADD instruction

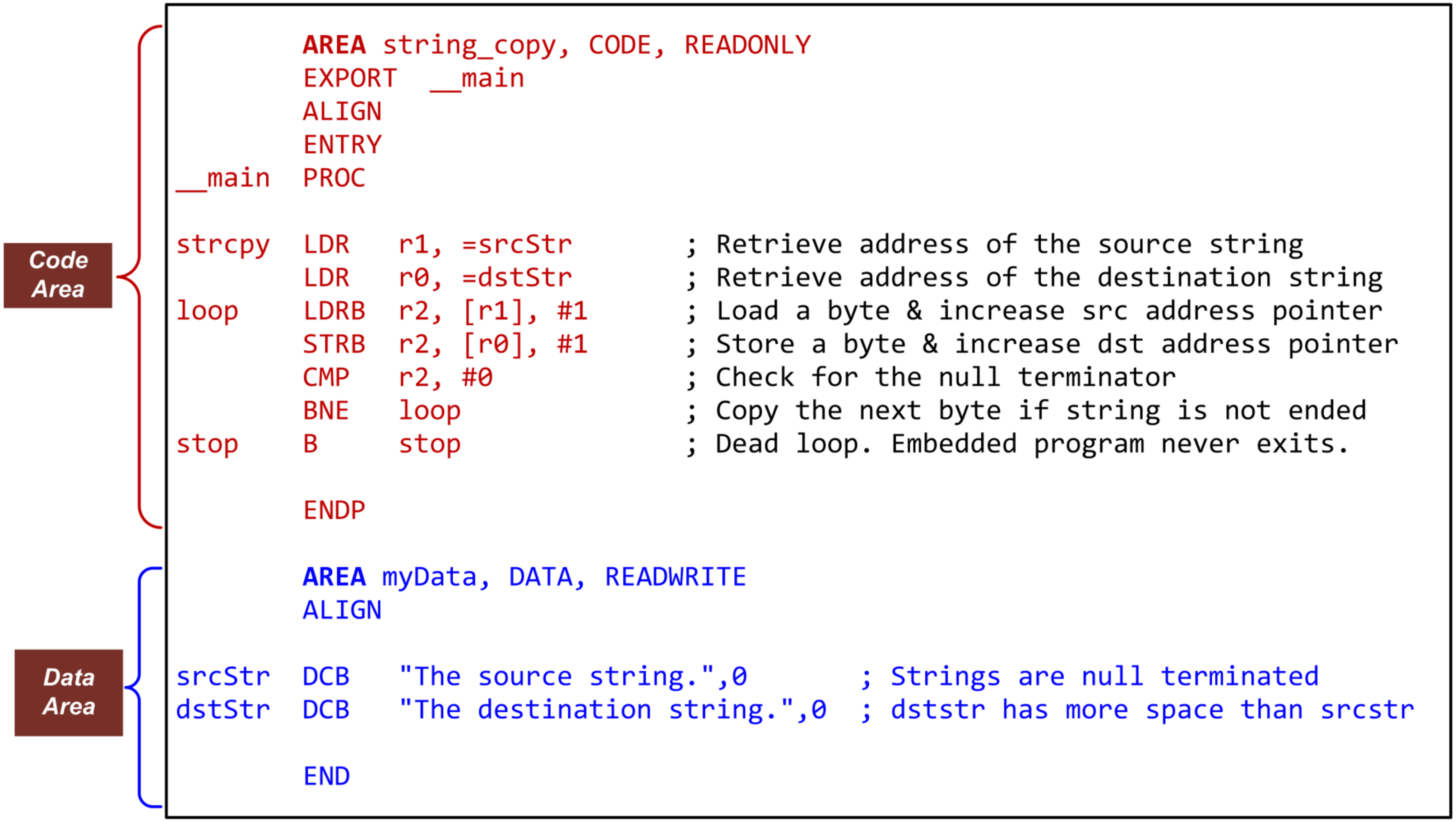
```
ADD r1, r2, r3      ; r1 = r2 + r3
ADD r1, r3          ; r1 = r1 + r3
ADD r1, r2, #4      ; r1 = r2 + 4
ADD r1, #15         ; r1 = r1 + 15
```

First Assembly



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

strcpy  LDR   r1, =srcStr       ; Retrieve address of first string
        LDR   r0, =dstStr       ; Retrieve address of second string
        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   strcpy            ; Cope 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
```

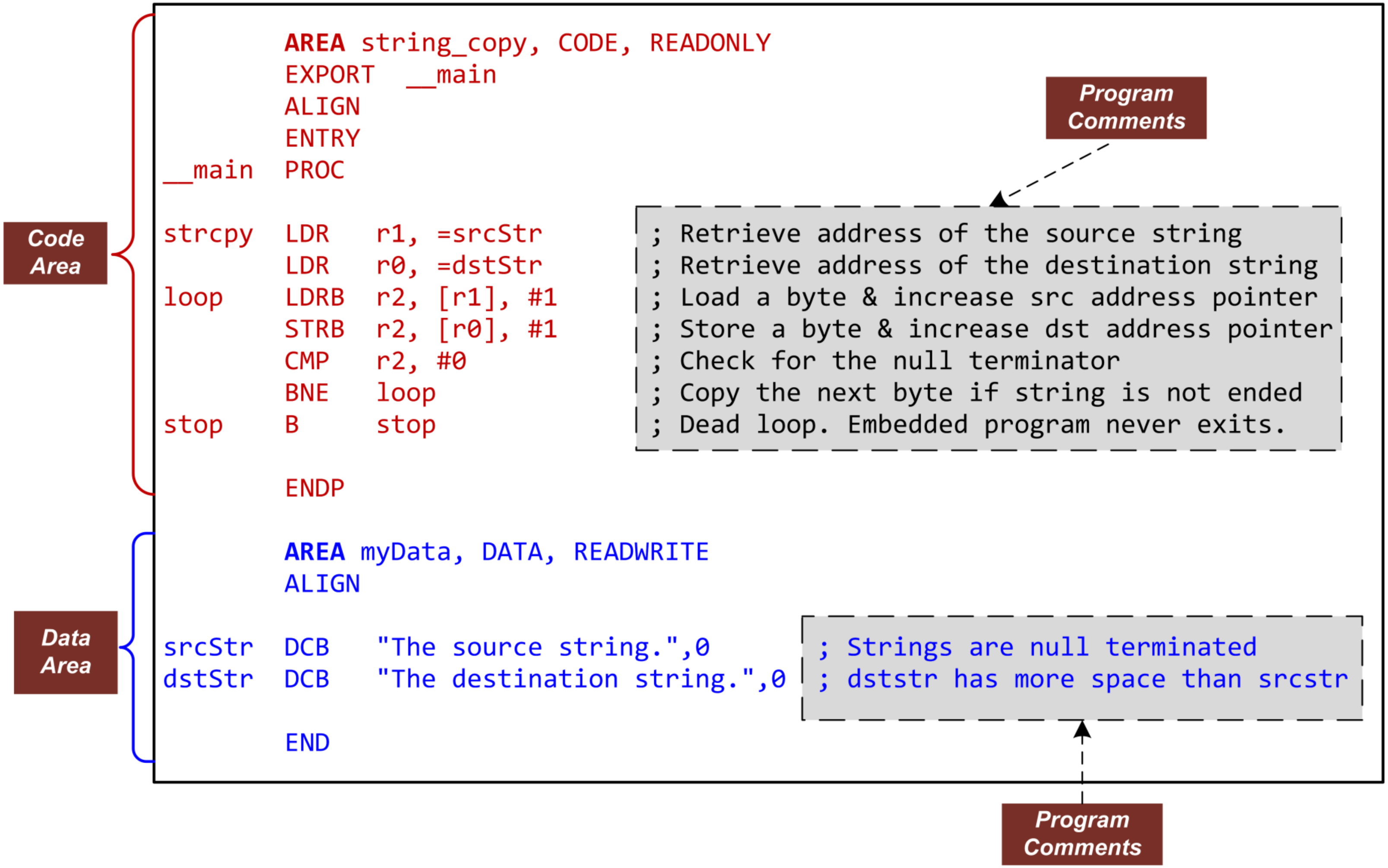
First Assembly



Code Area
Data Area

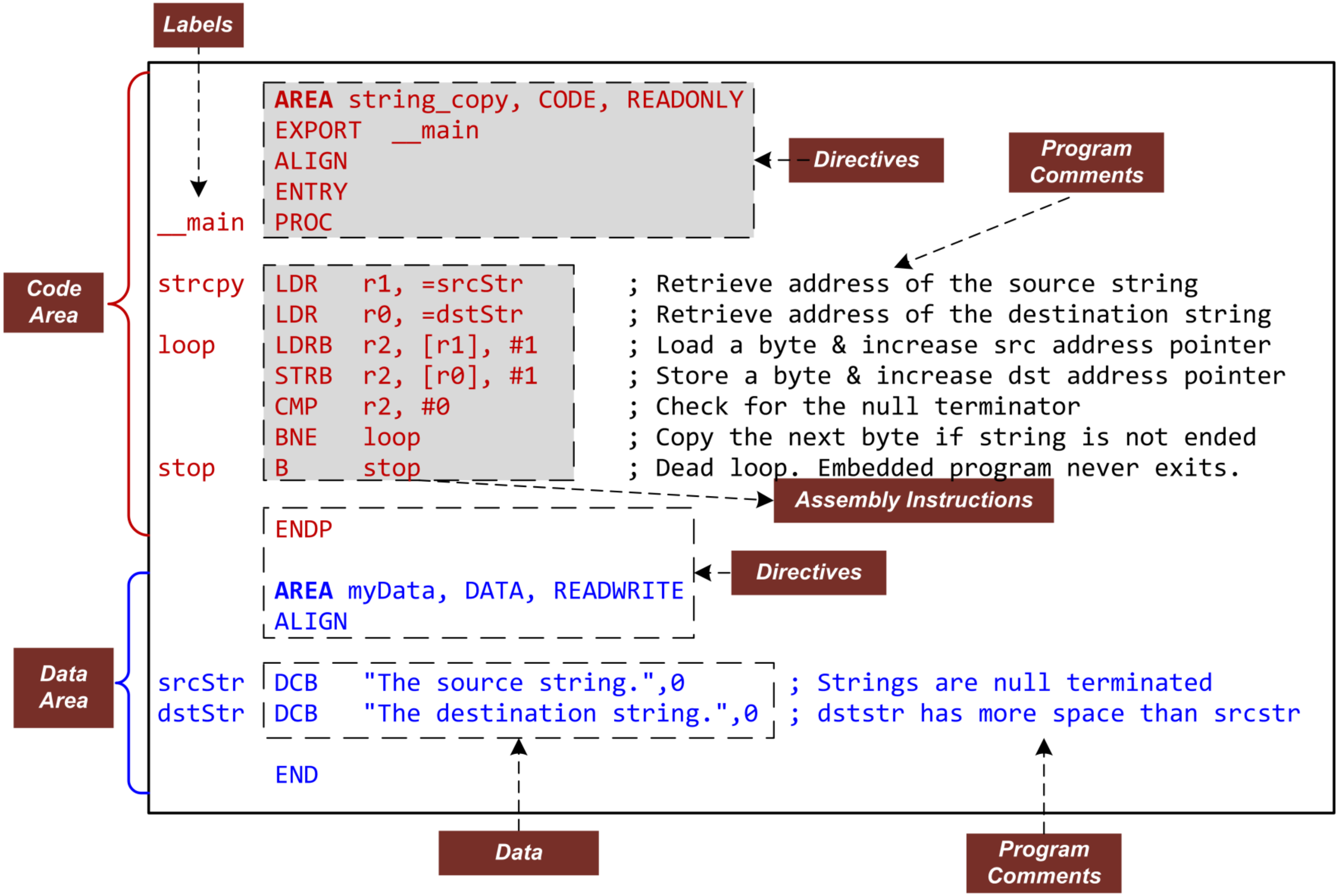
```
        AREA string_copy, CODE, READONLY
        EXPORT  __main
        ALIGN
        ENTRY
__main  PROC

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
```

First Assembly



```
        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
```

First Assembly



```
        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
```

Assembly Directives

- Directives are not real instruction commands. Instead, they are used to provide key information for compilation.

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 and a data area may be read from and written to.

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 first instruction to be executed within an application.
- There must be one and only 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 subroutine within another subroutine.

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 IMPORT is similar to the "extern" keyword in C.

Directive: Data Allocation

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
SPACE	Defined Zeroed Bytes	Reserve a number of zeroed bytes
FILL	Defined Initialized Bytes	Reserve and fill each byte with a value

Directive: Data Allocation

```
 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

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 of a word
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.