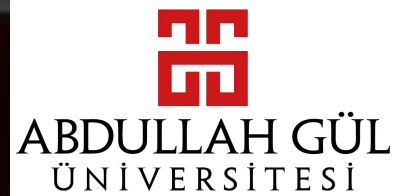


EE 304 Embedded Systems

Mehmet Bozdal



ARM
STM32F407IGT6
VQ337424 AA052
TWN HP 431

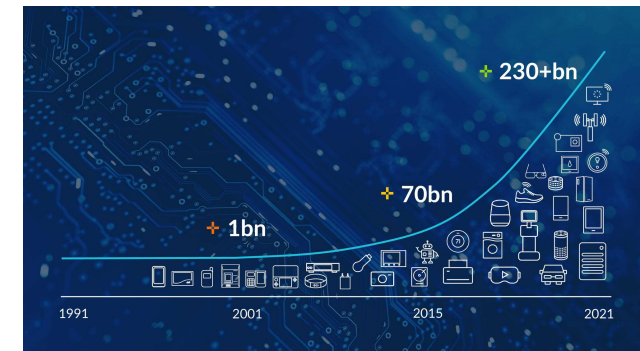


Agenda

- Arm and Cortex-M3
- Registers
- Assembly Syntax
 - Symbol
 - Instruction
 - Directive

Arm

- Arm is a semiconductor and software design company based in Cambridge, England.
- They don't manufacture silicon so it is a fabless company.
- Develops the architectures and licenses them to other companies.
- Low costs, minimal power consumption, and lower heat generation.
- Arm processors are desirable for light, portable, battery-powered devices, including smartphones, laptops and tablet computers, and embedded systems.



Arm Cortex Processor Family

Cortex-M Series

- Cortex-M processors strike a balance between performance, cost-effectiveness, and energy efficiency.
- They are well-suited for a wide spectrum of microcontroller applications.
- Explain their suitability for various microcontroller applications.
- Typical use cases encompass home appliances, robotics, industrial control, smartwatches, and IoT devices.
- *Throughout this course, our primary focus will be on the STM32F107 microcontroller boards, which are based on the Cortex-M3 architecture.*

Cortex-A Series

- The Cortex-A series processors are engineered for high-performance applications.
- They support for full-fledged operating systems like Linux, iOS, and Android.

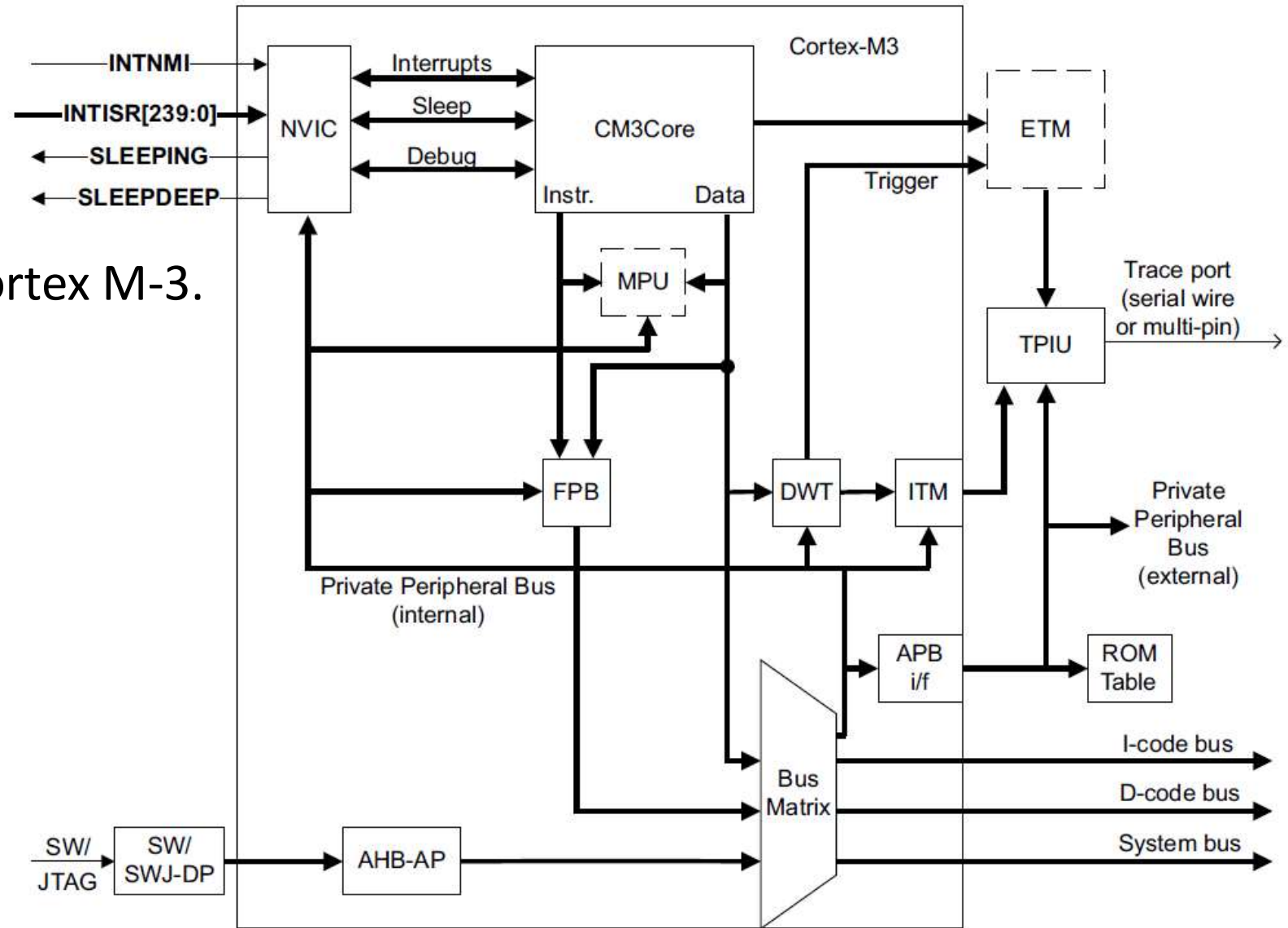
Cortex-R Series

- Cortex-R processors are purpose-built for mission-critical real-time systems.
- They excel in environments demanding high reliability, fault-tolerance, and deterministic real-time responsiveness.

Cortex M-3

STM32F107 is Cortex M-3.

- Harvard
- Thumb-2 ISA



ARM Instruction Sets

- **Thumb**

- 16-bit instructions.
- Code density optimization.
- Limited operands and registers.
- Ideal for resource-constrained embedded systems.

- **ARM32**

- 32-bit instructions.
- Enhanced coding flexibility.
- Faster execution.
- Larger code size.
- Suitable for a wide range of applications.

- **Thumb-2**

- Combines 16-bit Thumb and select 32-bit ARM32 instructions.
- Balanced code density and performance.
- Ideal for various embedded and mobile platforms.
- Cortex-M3 is based on Thumb-2

- **ARM64**

- 64-bit instructions.
- High computational power.
- Enhanced memory addressability.
- Commonly used in desktops and servers.

ISA Defines

- Instructions supported
- Registers
- Data types
- Addressing modes
- Memory access

Cortex-M3 – Thumb-2

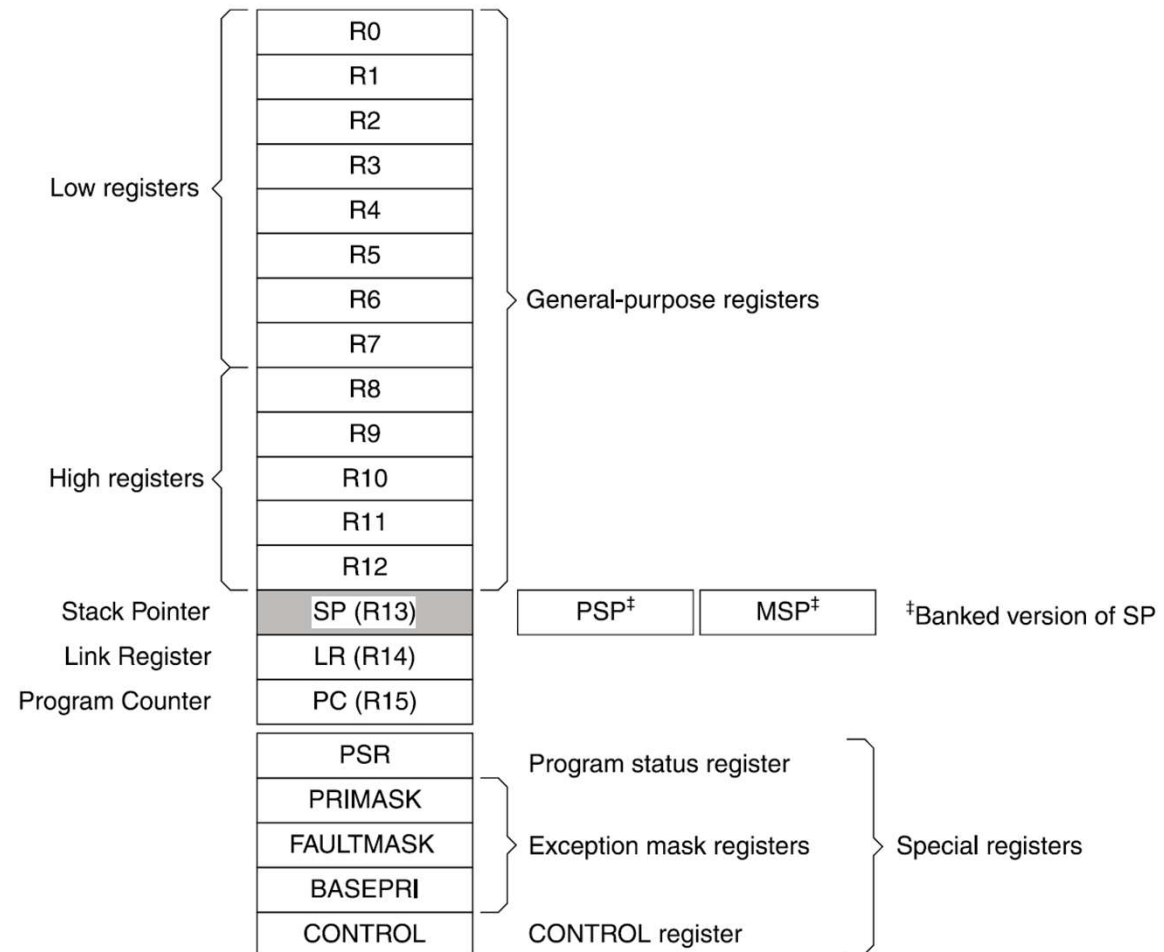
Cortex-M3 is a RISC machine which follows a load/store architecture. Instructions can be classified in three categories:

- **Memory Access:** These instructions facilitate data transfer between memory and registers.
- **Register-to-Register Operations:** Instructions dedicated to ALU (Arithmetic Logic Unit) operations between registers.
- **Control Flow Operations:** Instructions that enable programming language control flow, including 'if,' procedure calls, and etc.

Registers

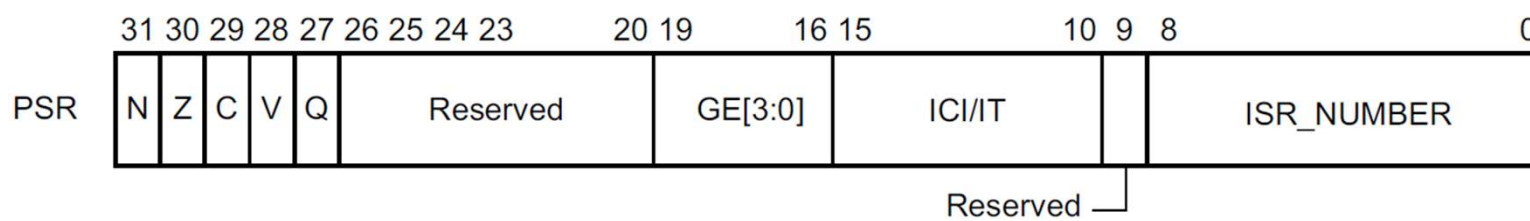
- R0-R12 are 32-bit general-purpose registers for data operations.
- The Stack Pointer (SP) is register R13. It holds the address of the top of the stack.
- The Link Register (LR) is register R14. It stores the return information for subroutines, function calls, and exceptions.
- The Program Counter (PC) is register R15. It contains the current program address.

More: STM32F1xx Cortex®-M3 Programming manual



ai15996

Registers - Program status register



Gives information about the most recent operation

Registers - Program status register

N: Negative or less than flag:

- 0: Operation result was positive, zero, greater than, or equal
 - 1: Operation result was negative or less than.
-

Z: Zero flag:

- 0: Operation result was not zero
 - 1: Operation result was zero.
-

C: Carry or borrow flag:

- 0: Add operation did not result in a carry bit or subtract operation resulted in a borrow bit
 - 1: Add operation resulted in a carry bit or subtract operation did not result in a borrow bit.
-

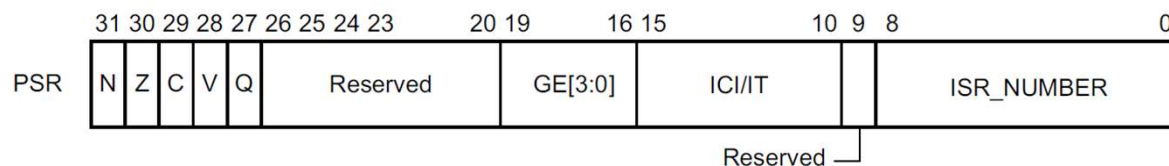
V: Overflow flag:

- 0: Operation did not result in an overflow
 - 1: Operation resulted in an overflow.
-

Q: DSP overflow and saturation flag: Sticky saturation flag.

- 0: Indicates that saturation has not occurred since reset or since the bit was last cleared to zero
 - 1: Indicates when an SSAT or USAT instruction results in saturation, or indicates a DSP overflow.
- This bit is cleared to zero by software using an MRS instruction.

Registers - Program status register



Gives information about the most recent operation

ISR_NUMBER:

This is the number of the current exception:

- 0: Thread mode
- 1: Reserved
- 2: NMI
- 3: Hard fault
- 4: Memory management fault
- 5: Bus fault
- 6: Usage fault
- 7: Reserved
-
- 10: Reserved
- 11: SVCall
- 12: Reserved for Debug
- 13: Reserved
- 14: PendSV
- 15: SysTick
- 16: IRQ0⁽¹⁾
-
- 255: IRQ240⁽¹⁾

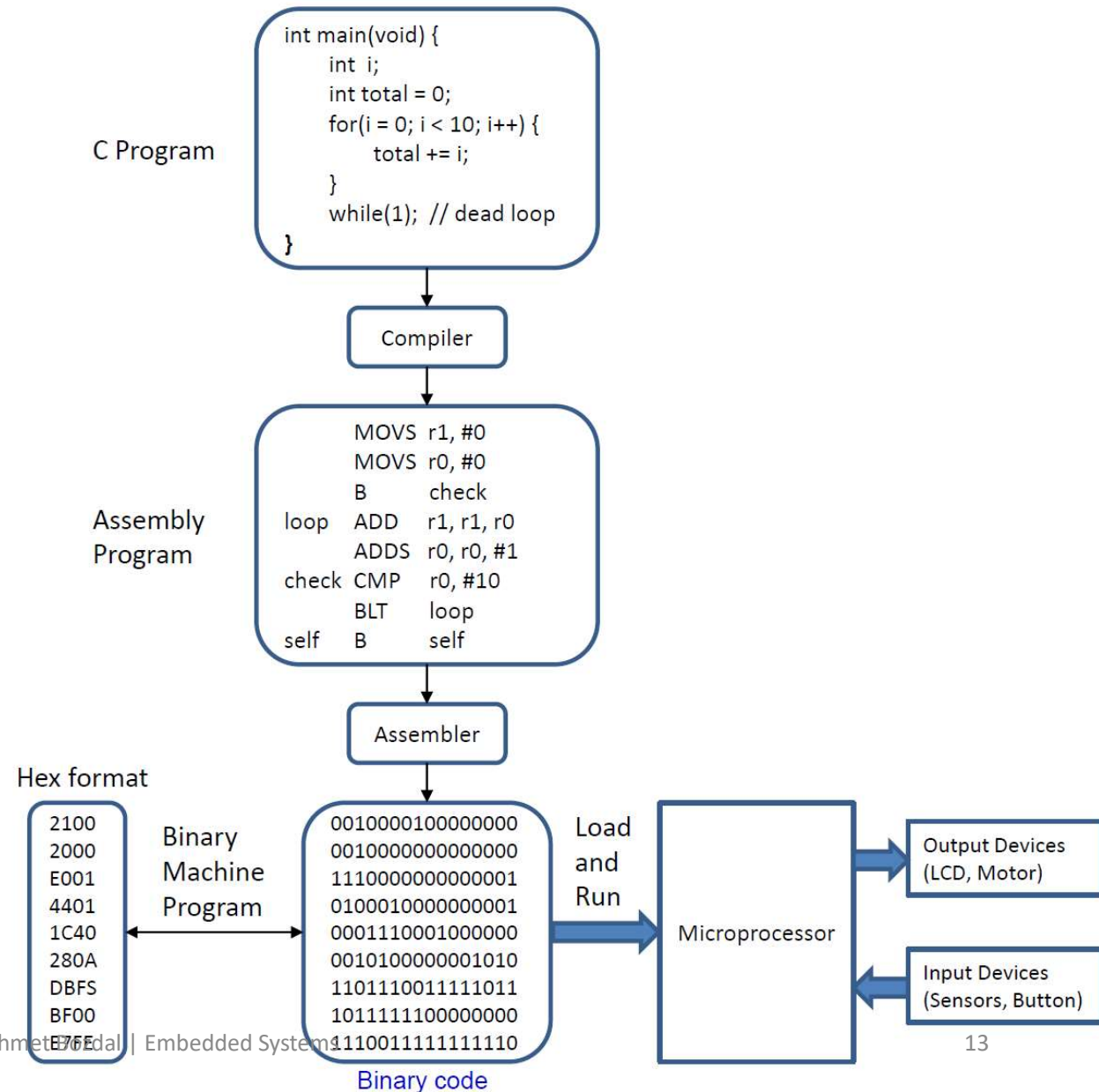
Levels of Program Code

High level languages (e.g. C/C++) are easy to code for programmers.

Assembly is just a more readable version of machine language (machine language in symbolic form instead of binary form).

Machine language is the only language that is ever actually executed by a computer.

Because .hex is much more readable and useful than binary –hex code is often used and shown.



Why assembly?

- High-quality optimizing compilers for high-level languages
- ARM architecture is specifically optimized for high-level languages

However,

- Assembly language makes it possible to manipulate hardware directly
 - writing low-level software such as bootloaders and operating system kernels
- Understanding processor and memory function
- It eases debugging
- Reverse engineering software

Key Components of Assembly Programs

{symbol} {instruction|directive|pseudo-instruction} {;comment}

Key Components of Assembly Programs

{*symbol*} {*instruction|directive|pseudo-instruction*} {;*comment*}

- *symbol* is usually a label. In instructions and pseudo-instructions it is always a label. In some directives it is a symbol for a variable or a constant. The description of the directive makes this clear in each case.
- *symbol* must begin in the first column. It cannot contain any white space character such as a space or a tab unless it is enclosed by bars (|).
- Labels are symbolic representations of addresses. You can use labels to mark specific addresses that you want to refer to from other parts of the code.

Key Components of Assembly Programs

`{symbol} {instruction|directive|pseudo-instruction} {;comment}`


- *instructions* are the fundamental operations that the processor can execute (e.g., MOV, ADD, SUB).

Syntax of Assembly Language

<i>Label</i>	<i>Opcode</i>	<i>Operands</i>	<i>Comment</i>
Func	MOV	r0, #100	; this sets R0 to 100
		BX LR	; this is a function return

Syntax of Assembly Language

<i>Label</i>	<i>Opcode</i>	<i>Operands</i>	<i>Comment</i>
Func	MOV	r0, #100	; this sets R0 to 100
	BX	LR	; this is a function return



The **label field** is optional and starts in the first column and is used to identify the **position in memory** of the current instruction. You must choose a unique name for each label.

```
Func MOV r0, #100
      BX lr
```

Syntax of Assembly Language

<i>Label</i>	<i>Opcode</i>	<i>Operands</i>	<i>Comment</i>
Func	MOV	r0, #100	; this sets R0 to 100
		BX LR	; this is a function return

The **opcode field** specifies the processor command to execute. Mnemonic is a symbolic name for the opcode.

Syntax of Assembly Language

<i>Label</i>	<i>Opcode</i>	<i>Operands</i>	<i>Comment</i>
Func	MOV	r0, #100	; this sets R0 to 100
		BX LR	; this is a function return



The **operand field** specifies where to find the data to execute the instruction. Thumb instructions have 0, 1, 2, 3, or 4 operands, separated by commas.

```
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
```

Syntax of Assembly Language

<i>Label</i>	<i>Opcode</i>	<i>Operands</i>
Func	MOV	r0, #100
		BX LR

Comment

```
; this sets R0 to 100  
; this is a function return
```

The **comment field** is also optional and is ignored by the assembler, but it allows you to describe the software making it easier to understand.

A semicolon must separate the operand(s) and comment field.

Syntax - directive

`{symbol} {instruction|directive|pseudo-instruction} {;comment}`

- Directives provide important information to the assembler that either affects the assembly process or affects the final output image.
- They are used to provide key information to compile the source program, such as declaring constants and symbolic names, defining data layout, allocating memory space, and specifying the program structure and entry point.

Syntax - directive

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 memory boundary
DCB	Allocate one or more bytes (8 bits) of data
DCW	Allocate one or more halfwords(16 bits) of data
DCD	Allocate one or more words (32 bits) of data
DCFS	Allocate single-precision (32 bits) floating-point numbers
DCFB	Allocate double-precision (64 bits) floating-point numbers
SPACE	Allocate a zeroed block of memory
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
PROC	Declare the start of a procedure
ENDP	Designate the end of a procedure
END	Designate the end of a source file

Syntax - Directives

- Instructions (e.g. ADD, MOV) tell the CPU what to do
- Assembler directives tell the assembler what to do
 - AREA
 - ENTRY and END
 - PROC and ENDP
 - IMPORT and EXPORT
- Define Constant Data - DCD, DCW, DCB

Directive - AREA

- The AREA directive signals the start of a new section, either for code or data.
- It specifies the memory locations where programs, subroutines or data reside.
- Each area is a fundamental, self-contained unit. Processed independently by the linker.
- **An assembly program must have at least one code area**

AREA {sectionName} {attribute1} {, attribute2}...

Key attribute

- CODE: area includes only instructions
- DATA: area includes only data
- READONLY: the default for CODE areas
- READWRITE: the default for DATA areas

Directive - AREA

<code>AREA myData, DATA, READWRITE</code>	; Define a data section
<code>Array DCD 1, 2, 3, 4, 5</code>	; Define an array with five integers
<code>AREA myCode, CODE, READONLY</code>	; Define a code section
<code>EXPORT __main</code>	; Make __main visible to the linker
<code>ENTRY</code>	; Mark the entrance to the entire program
<code>__main PROC</code>	; PROC marks the begin of a subroutine
<code>...</code>	; Assembly program starts here.
<code>ENDP</code>	; Mark the end of a subroutine
<code>END</code>	; Mark the end of a program

Directive – ENTRY & END

ENTRY

- The ENTRY directive designates the first instruction to be executed in an application.
- An application should have one and only one ENTRY directive, regardless of the number of source files.
- Omission of the ENTRY directive results in a linker error.
- Presence of multiple ENTRY directives triggers an assembler error.

END

- Informs the assembler that the end of the source file has been reached.
- Each assembly program must end with this directive.

Directive - ENTRY,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

Directive – PROC & ENDP

- 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.
- An assembly program must have at least one subroutine named `_main`.

Directive – PROC & 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

Directive – Define Constant (Data)

Entering fixed data into program memory. DCB,DCW,DCD

{Label} DC{B,W,D} expression

Data may include:

- Name
- Key identification
- Subroutine address
- Code conversion tables

Directive – Define Constant (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 Doubleword	Reserve 64-bit values
SPACE	Defined Zeroed Bytes	Reserve some zeroed bytes
FILL	Defined Initialized Bytes	Reserve and fill each byte with a value

Directive – Define Constant (Data)

- DCB allocates bytes (8-bit) of memory & initializes them.

MYVALUE DCB 5

FIBO DCB 1,1,2,3,5,8

MY_MSG DCB "Hello World!"

- DCW allocates a half-word(16-bit)

- MYVALUE DCW 25425

- DCD allocates a word of memory(32-bit)

- MYDATA DCD 0x200000, 0x30F5, 5000000

- SPACE allocates memory without initializing.

BETA SPACE 255;Allocate 255 bytes of zeroed memory space

Directive – Define Constant (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

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 initialized to binary value 2_01010101

octal DCB 8_73 ; Allocate a byte initialized to octal value 8_73

char DCB 'A' ; Allocate a byte initialized to ASCII character 'A'
```

Directive - EQU and RN

- 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.

```
COUNT EQU 0x25
GPIOA_ODR EQU 0x4001080C
MOV R1, #COUNT
```

- The RN directive gives a symbolic name to a specific register.

```
RESULT RN R2
MOV RESULT, #23
```

Directive - ALIGN

Alignment Requirement:

- Many processors require that the starting memory address of an instruction or a variable must be a multiple of 2^n .
- For example, an address aligned to a word boundary must be divisible by 4 (i.e., 2^2).

Consequences of Misalignment:

- When instructions or data are not appropriately aligned in memory, some processors generate a misalignment fault signal.
- Misaligned memory accesses can result in the processor aborting the memory access.

Impact on Performance:

- To improve performance, processors encourage memory alignment.
- Cortex-M processors allow unaligned memory accesses, but with a performance trade-off.
 - Multiple memory accesses may be required to fetch a misaligned data item or instruction.

Directive - IMPORT and EXPORT

- 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 - IMPORT and EXPORT

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

Directive – INCLUDE & GET

- GET directive includes a file within the file being assembled. The included file is assembled at the location of the GET directive.
- INCLUDE is a synonym for GET.
- GET is useful for including macro definitions, EQUs, and storage maps in an assembly.

AREA Example, CODE, READONLY

GET file1.s ; includes file1 if it exists

; in the current place.

GET c:\project\file2.s ; includes file2

GET c:\Program files\file3.s ; space is permitted

How do computers read code?

- https://www.youtube.com/watch?v=QXjU9qTsYCc&ab_channel=FrameofEssence