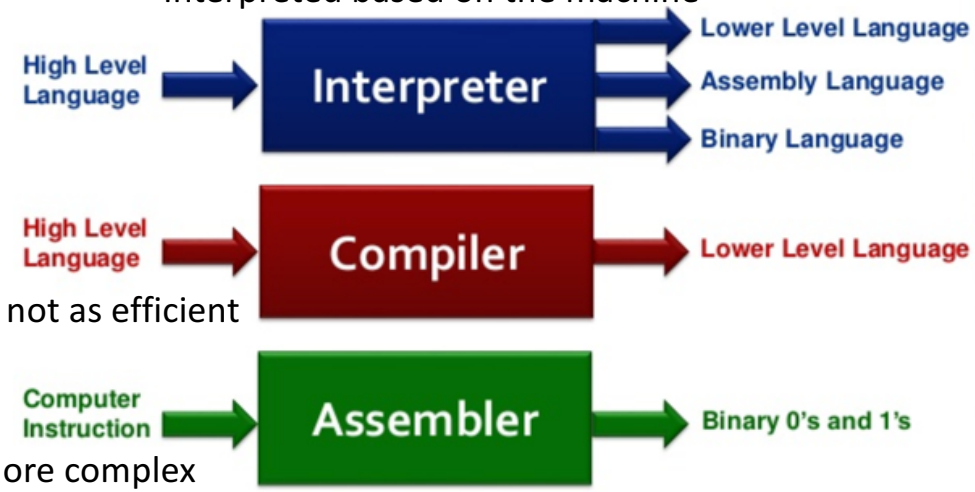
Chapters 3
ARM Assembly

Embedded Systems with ARM Cortext-M

Updated: Wednesday, February 7, 2018

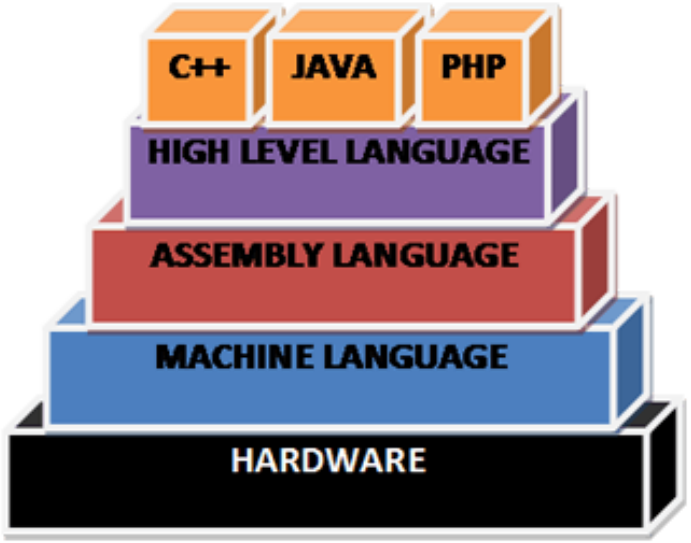
Programming languages - Categories

Interpreted based on the machine

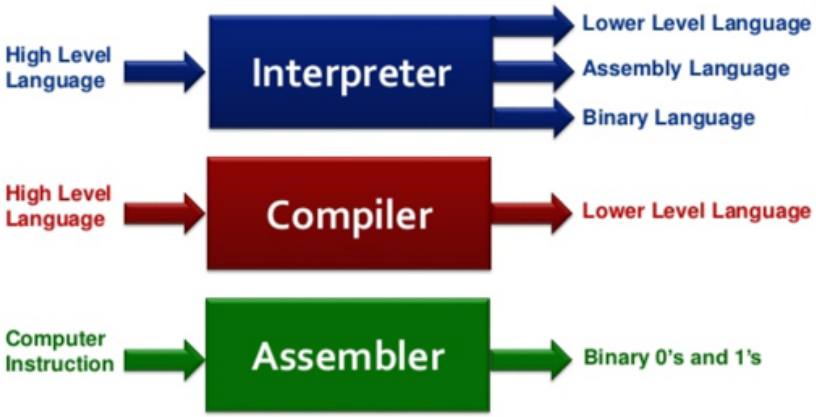


Less complex, not as efficient

Efficient, more complex



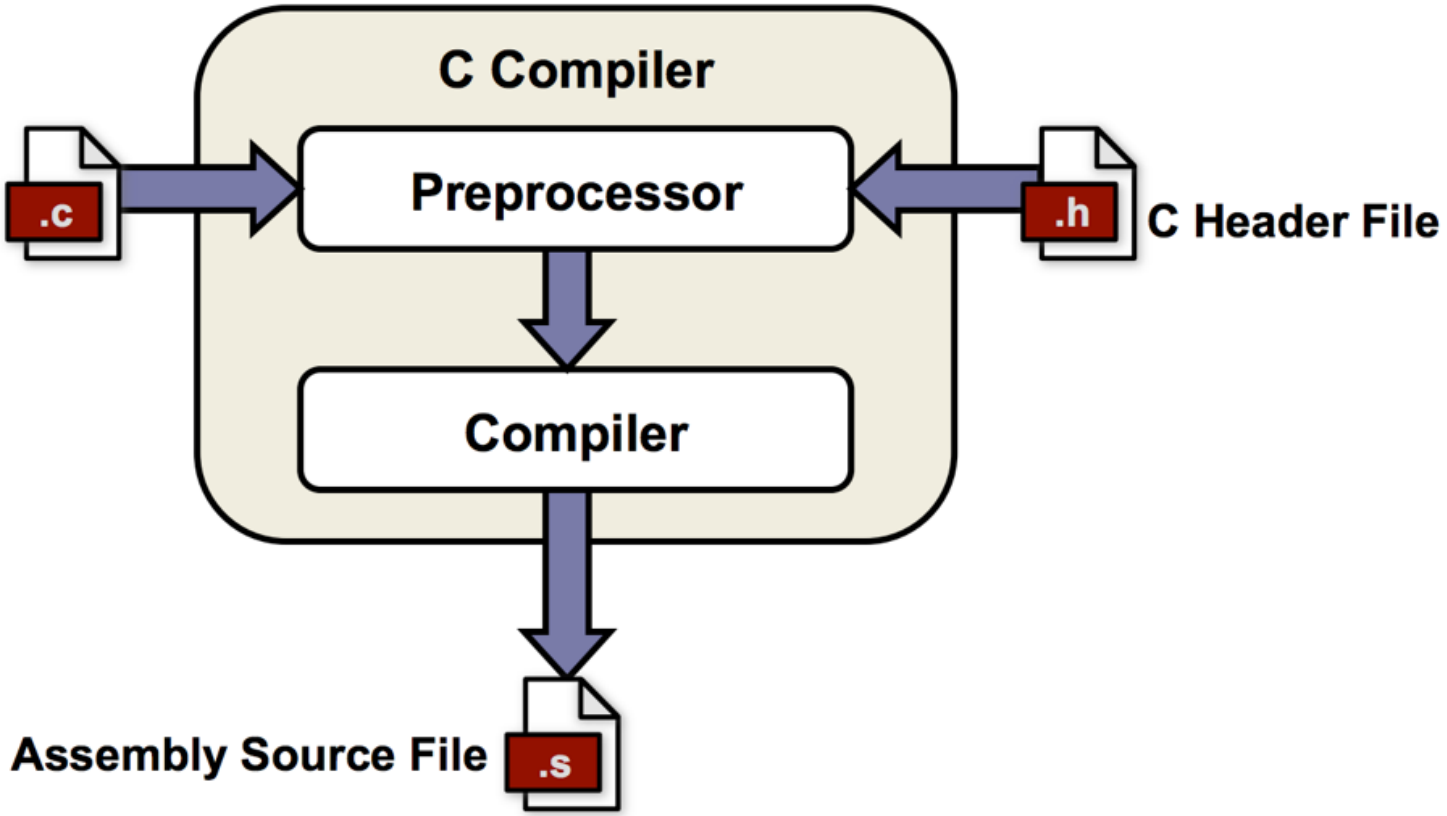
Programming languages -
Interpreter Vs. Compiler



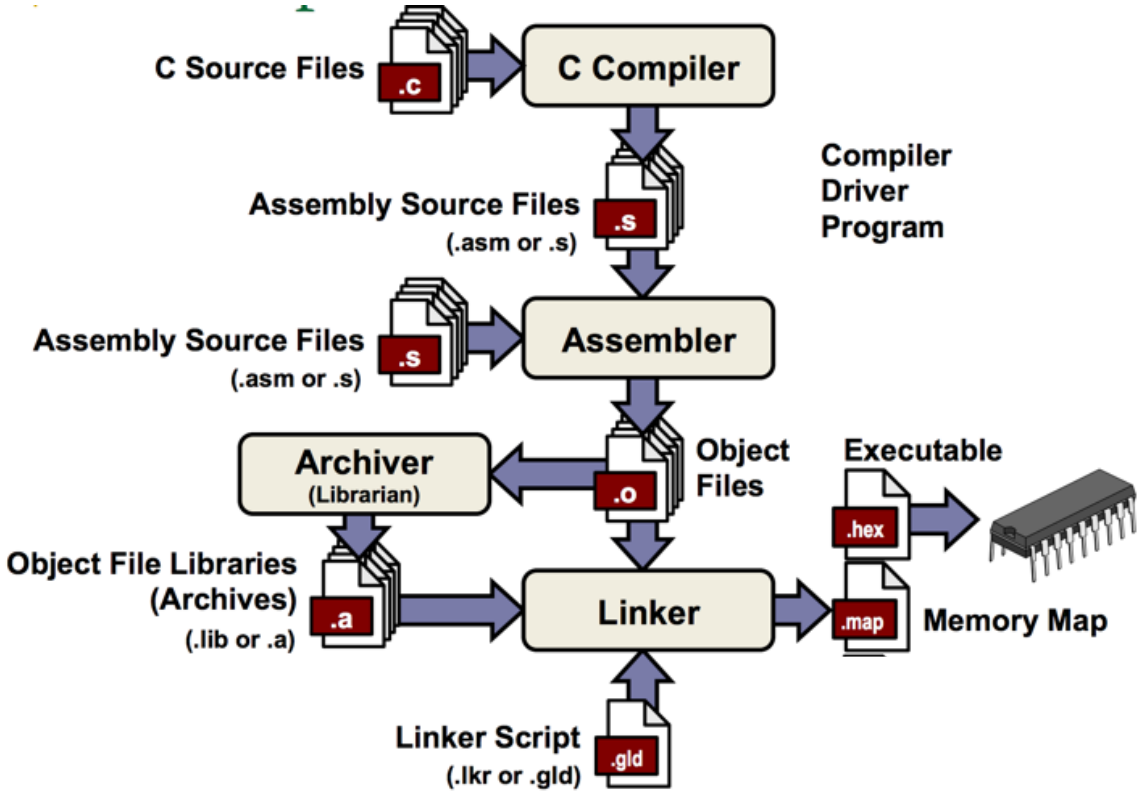
Interpreter	Compiler
Translates program one statement at a time.	Scans the entire program and translates it as a whole into machine code.
It takes less amount of time to analyze the source code but the overall execution time is slower.	It takes large amount of time to analyze the source code but the overall execution time is comparatively faster.
No intermediate object code is generated, hence are memory efficient.	Generates intermediate object code which further requires linking, hence requires more memory.
Continues translating the program until the first error is met, in which case it stops. Hence debugging is easy.	It generates the error message only after scanning the whole program. Hence debugging is comparatively hard.
Programming language like Python, Ruby use interpreters.	Programming language like C, C++ use compilers.

Assembler convers instructions into Machine Language 1s and 0s.

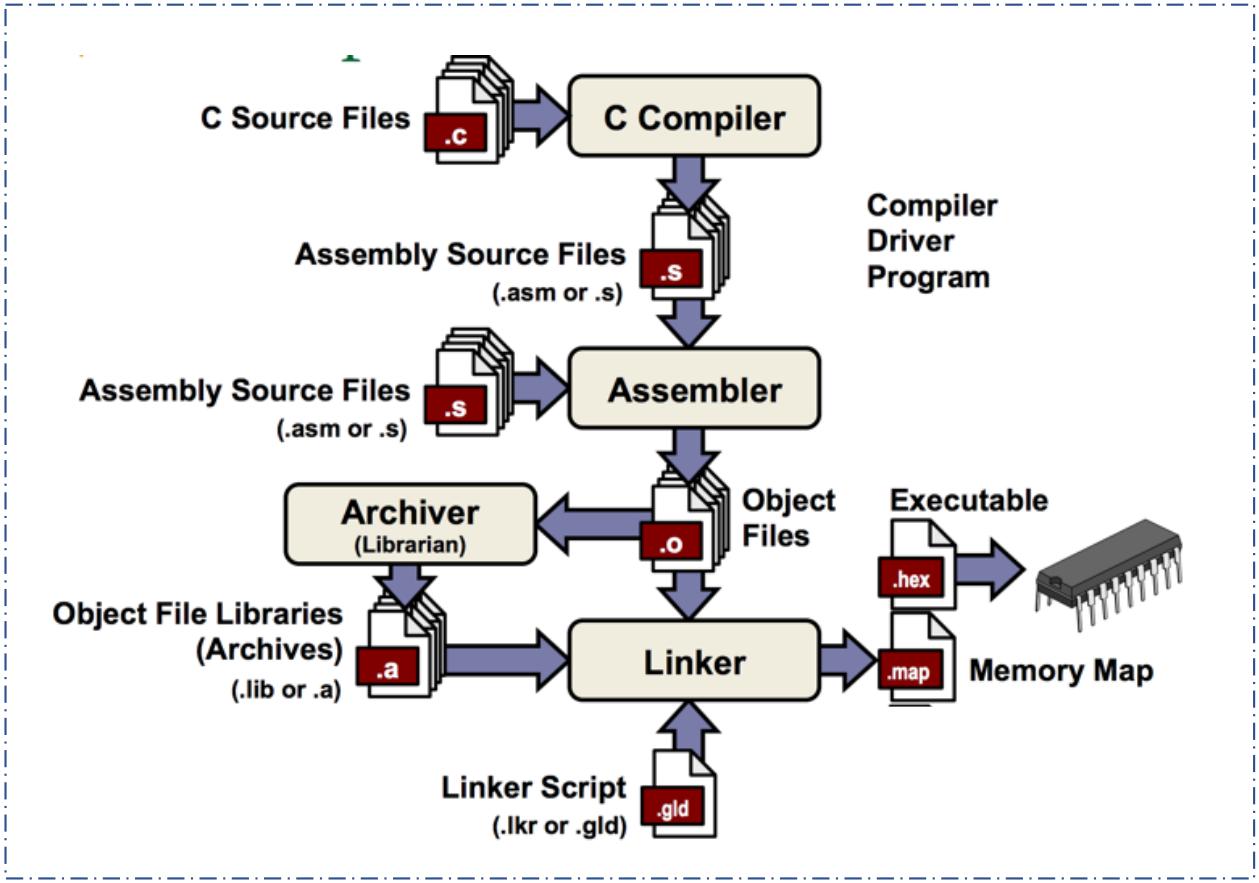
C Compiler


C Compiler
Preprocessor
Compiler
.c
.h
C Header File
Assembly Source File
.s

Assemblers and C Compilers



Assemblers and C Compilers



IDE - Integrated Development Environment

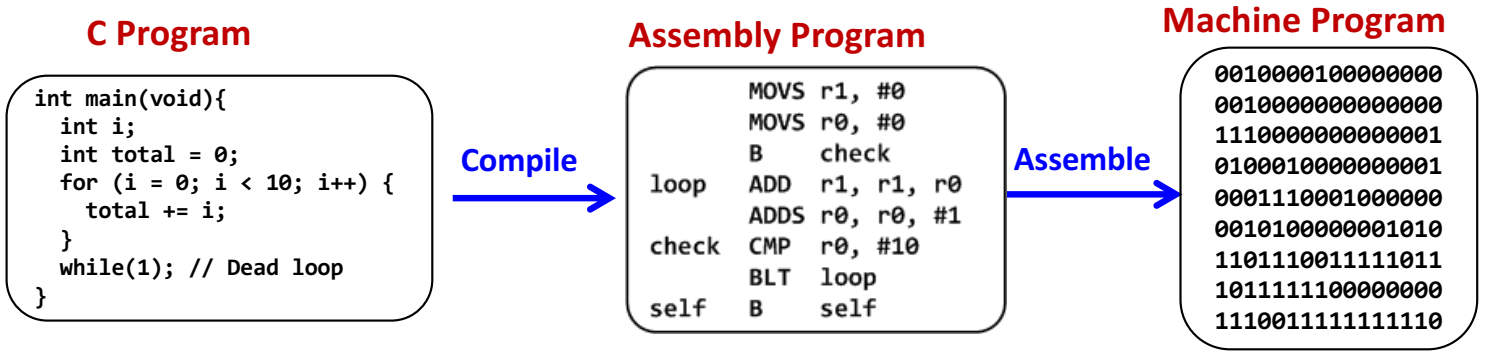
ARM Assembly

- Modern ARM processors have several instruction sets:
- The fully-featured 32-bit ARM instruction set,
- The more restricted, but space efficient, 16-bit *Thumb* instruction set,
- The newer mixed 16/32-bit *Thumb-2* instruction set,
- *Jazelle DBX* for Java byte codes,
- The *NEON* 64/128-bit SIMD instruction set,
- The *VFP* vector floating point instruction set.
- → Thumb-2 is the progression of Thumb (strictly it is Thumb v3). It improves performance whilst keeping the code density tight by allowing a mixture of 16- and 32-bit instructions.

http://www.davespace.co.uk/arm/introduction-to-arm/instruction-sets.html

Levels of Program Code

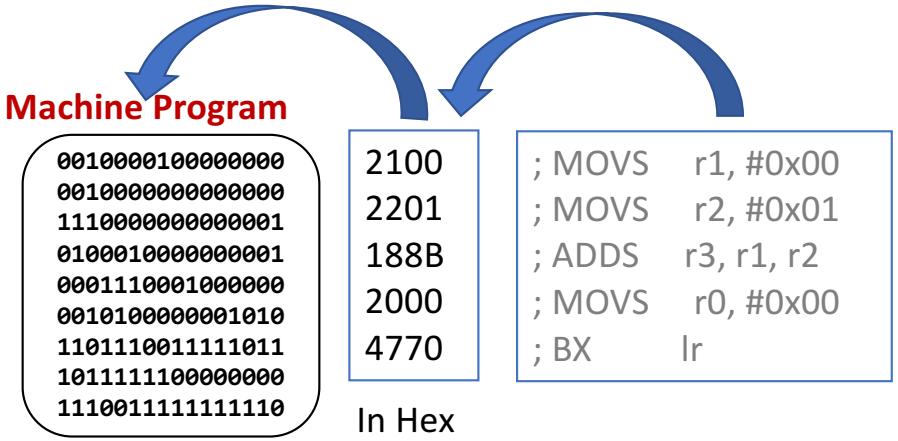
C Code → Assembly → Machine Language



- **High-level language**
  - Level of abstraction closer to problem domain
  - Provides for productivity and portability
- **Assembly language**
  - Textual representation of instructions
- **Hardware representation**
  - Binary digits (bits)
  - Encoded instructions and data

Levels of Program Code

C Code → Assembly → Machine Language

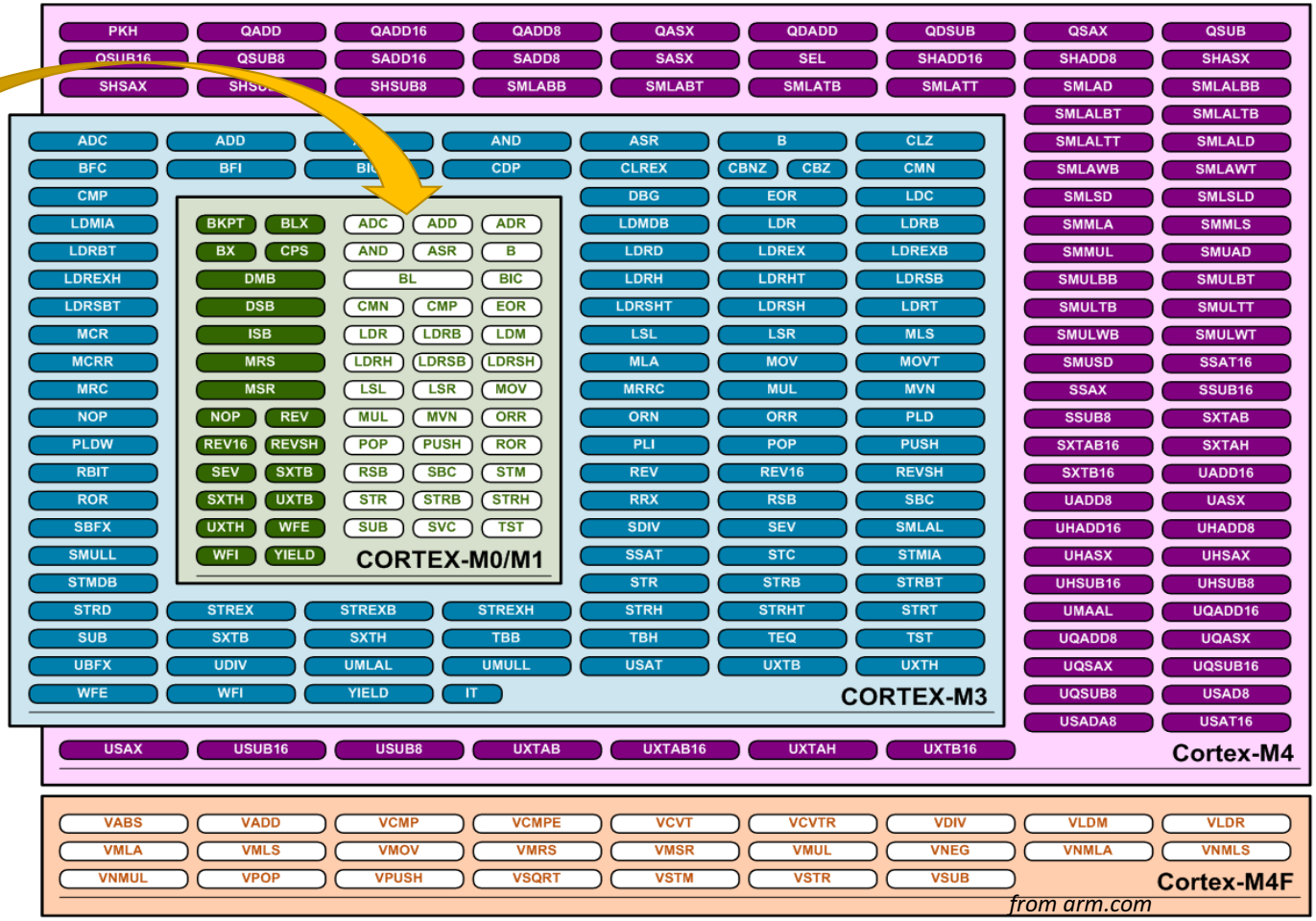


- **Hardware representation**
  - Binary digits (bits)
  - Encoded instructions and data

Assembly Instruction Sets for Cortex-M

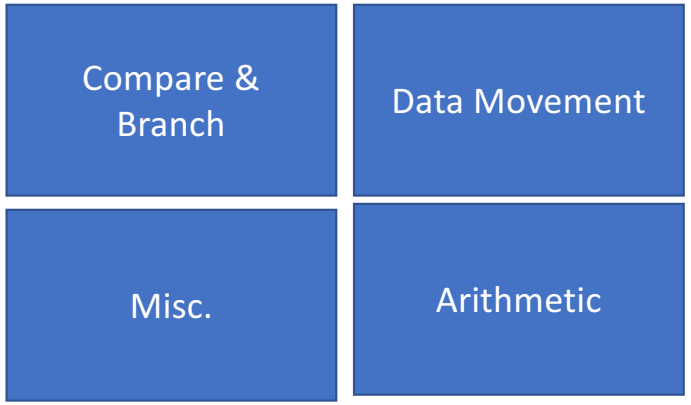
Examples:

- ADD
- AND
- CMP
- SUB
- MUL
- MOV
- etc.



from arm.com

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

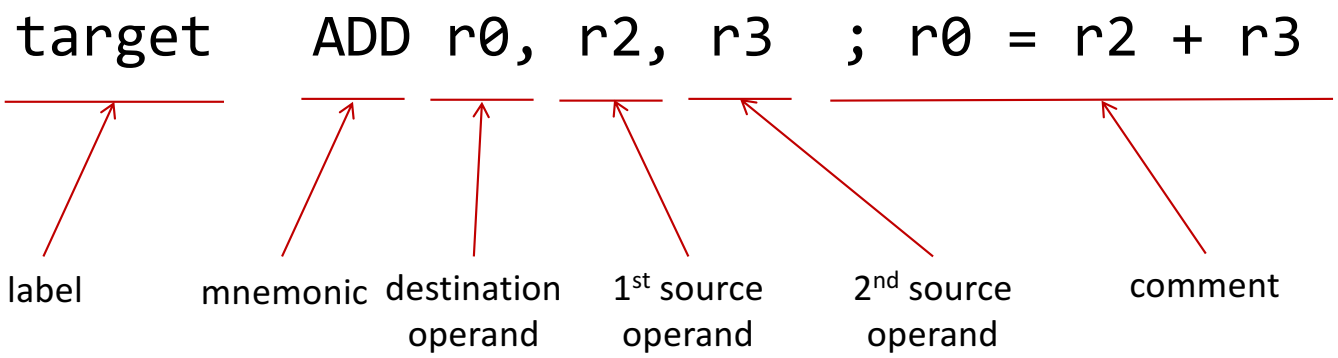
ARM Instruction Format

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

- Label is a reference to the **memory address** of this instruction.
- **Mnemonic** represents the operation to be performed (ADD, SUB, etc.).
- 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
```



ARM Instruction Format

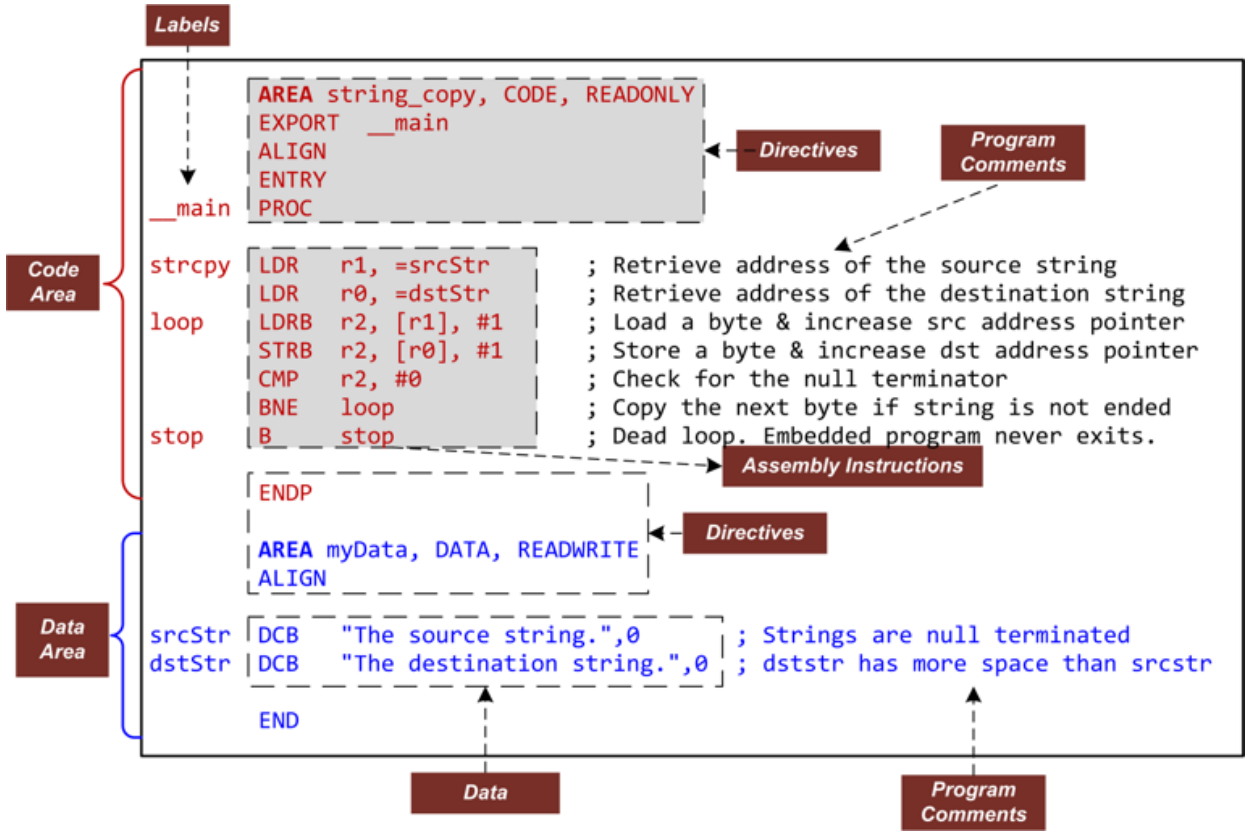
```
label        mnemonic operand1, operand2, operand3    ; comments
```

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
```

Remember:
R has two components:
- Register Address
- Register Content

First Assembly



Assembly Directives

```
INCLUDE constants.s        ; Load Constant Definitions
```

- Directives are **NOT** instruction; allocate space and define types in many cases. 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

Start of new data set; At least one code area is required per program; could be READONLY or READWRITE

Typically 2^N

For Example `INCLUDE constants.s ; Load Constant Definitions`

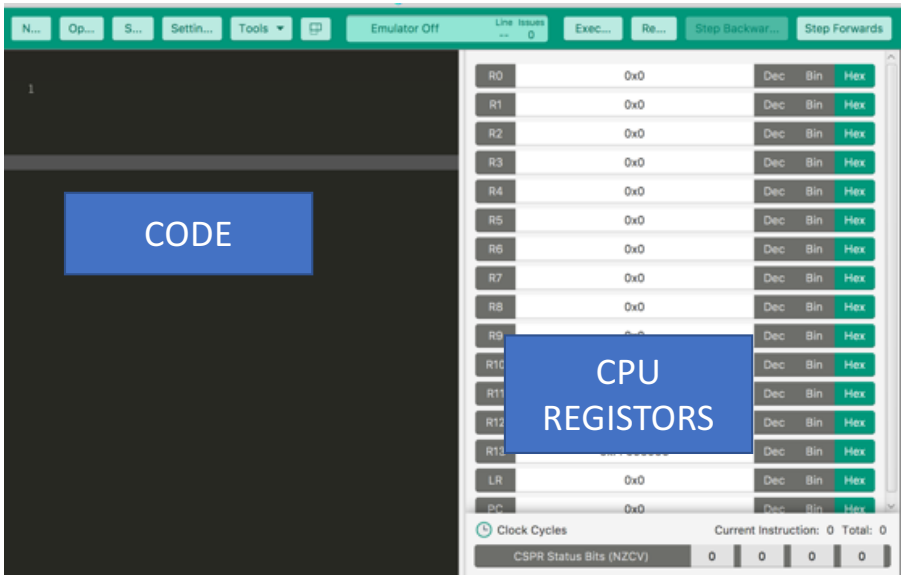
Refers to PROCEDURE and End of PROCEDURE; it defines the function; similar to main() in C

END of source file

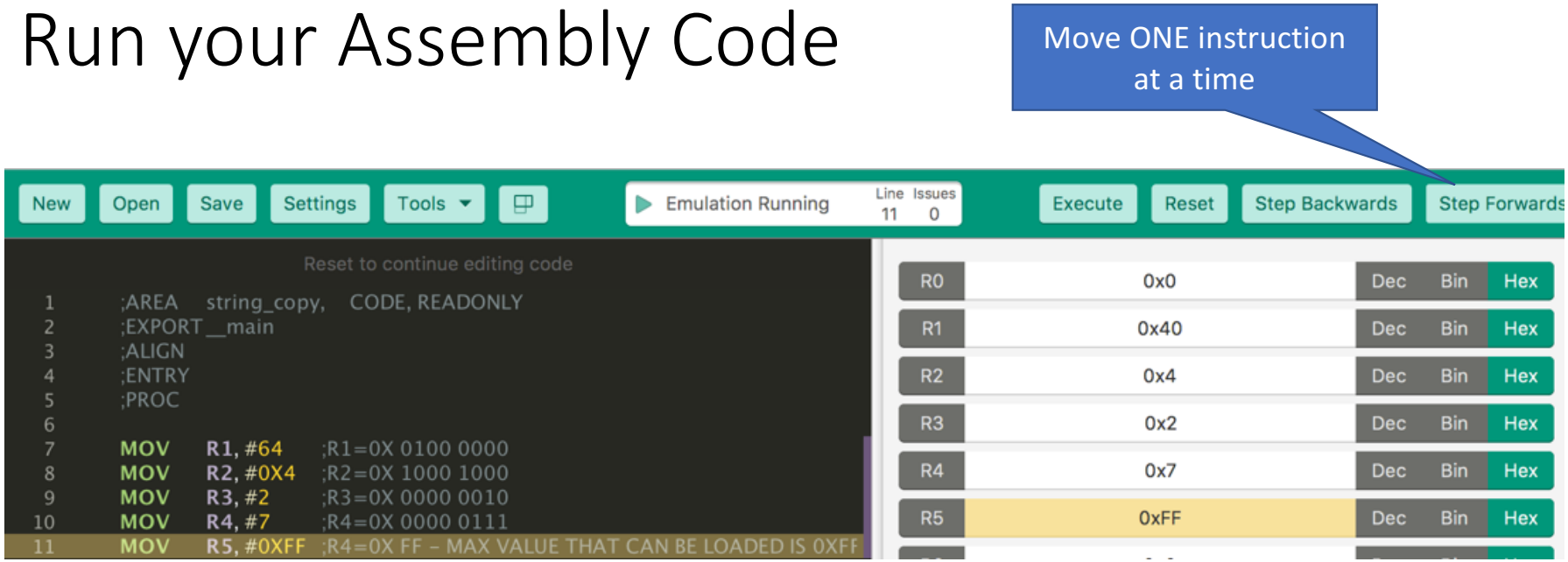
Let’s Practice First ….

Assembly Emulator

- Go to https://salmanarif.bitbucket.io/visual/index.html
- Download the appropriate version for your computer
- This is the list of supported instructions: https://salmanarif.bitbucket.io/visual/supported_instructions.html



Run your Assembly Code



What is your final PC value?

Try This & Answer the Questions…..

```
1  ; The purpose of this program is to add: aa+bb=cc
2  ; First we define the variables; The variables are stored in locations 0x100-0x108 in the MEMORY
3
4  ;load (32 bit) value 0x00000001 into memory location 0x100 (by default) call it aa
5  aa      DCD     0x00112233
6  bb      DCD     2               ;load (32 bit) value 0x00000002 into memory location 0x104 (by default)
7  cc      DCD     0               ;load (32 bit) value 0x00000000 into memory location 0x108 (by default)
8
9  main
10         LDR     r1, =aa         ;load memory address of aa into r1 After execution: PC = 0xC+4
11         LDR     r2, [r1]        ;load content of memory address in r1 into r2 (that is 0x0001)
12         LDR     r3, =bb
13         LDR     r4, [r3]
14         ADDS        r5, r2, r4  ;r2+r4 -> r5 and update flags
15         LDR     r6, =cc         ;load memory address of cc into r6
16         STR     r5, [r6]        ;store the sum into variable cc
17
18 ; this section shows that the sum is in fact loaded into cc
19         LDR     r7, =cc
20         LDR     r8, [r7]
21
22 ; let's check cc+1--> r9
23         ADDS        r9, r8, #1
24         END
```

Let’s Continue with Some Simple Commands….

Let's Take a MOV & MVN Commands

- MOV r0, #42
  - Move the constant 42 into register R0.
- MOV r2, r3
  - Move the contents of register R3 into register R2.
- MVN r1, r0
  - R1 = NOT(R0) = -43
- MOV r0, r0
  - A NOP (no operation) instruction.

- <operation>
- MOV – move
  - Rd := Operand2
- MVN – move NOT
  - Rd := 0xFFFFFFFF EOR Operand2

Arithmetic Operation

- <operation>
- ADD – Add
  - Rd := Rn + Operand2
- ADC – Add with Carry
  - Rd := Rn + Operand2 + Carry
- SUB – Subtract
  - Rd := Rn − Operand2
- SBC – Subtract with Carry
  - Rd := Rn − Operand2 − NOT(Carry)
- RSB – Reverse Subtract
  - Rd := Operand2 − Rn
- RSC – Reverse Subtract with Carry
  - Rd := Operand2 − Rn − NOT(Carry)

- ADD r0, r1, r2
  - R0 = R1 + R2
- SUB r5, r3, #10
  - R5 = R3 − 10
- RSB r2, r5, #0xFF00
  - R2 = 0xFF00 − R5

RSB and RSC subtract in reverse order (e.g. y - x not x - y).

Logical Instructions

- AND – *logical AND*
  - Rd := Rn AND Operand2
- EOR – *Exclusive OR*
  - Rd := Rn EOR Operand2
- ORR – *logical OR*
  - Rd := Rn OR Operand2
- BIC – *Bitwise Clear*
  - Rd := Rn AND NOT Operand2

- AND r8, r7, r2
  - R8 = R7 & R2
- ORR r11, r11, #1
  - R11 |= 1
- BIC r11, r11, #1
  - R11 &= ~1
- EOR r11, r11, #1
  - R11 ^= 1

Compare Instructions: <operation>{cond} Rn,Operand2

- <operation>
- CMP – *compare*
  - Flags set to result of (Rn − Operand2).
- CMN – *compare negative*
  - Flags set to result of (Rn + Operand2).
- TST – *bitwise test*
  - Flags set to result of (Rn AND Operand2).
- TEQ – *test equivalence*
  - Flags set to result of (Rn EOR Operand2).

- CMP r0, #42
  - Compare R0 to 42.
- CMN r2, #42
  - Compare R2 to -42.
- TST r11, #1
  - Test bit zero.
- TEQ r8, r9
  - Test R8 equals R9.
- SUBS r1, r0, #42
  - Compare R0 to 42, with result.

•CMP is like SUB.
•CMN is like ADD – subtract of a negative number is the same as add.
•TST is like AND.
•TEQ is like EOR – exclusive or of identical numbers gives result of zero.

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 first instruction to be executed** within an application 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: 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
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: 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

Pi      DCFS  3.14              ; Allocate a single-precision floating number

Pi      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.