C programming language pt5

Compilation process (build process):

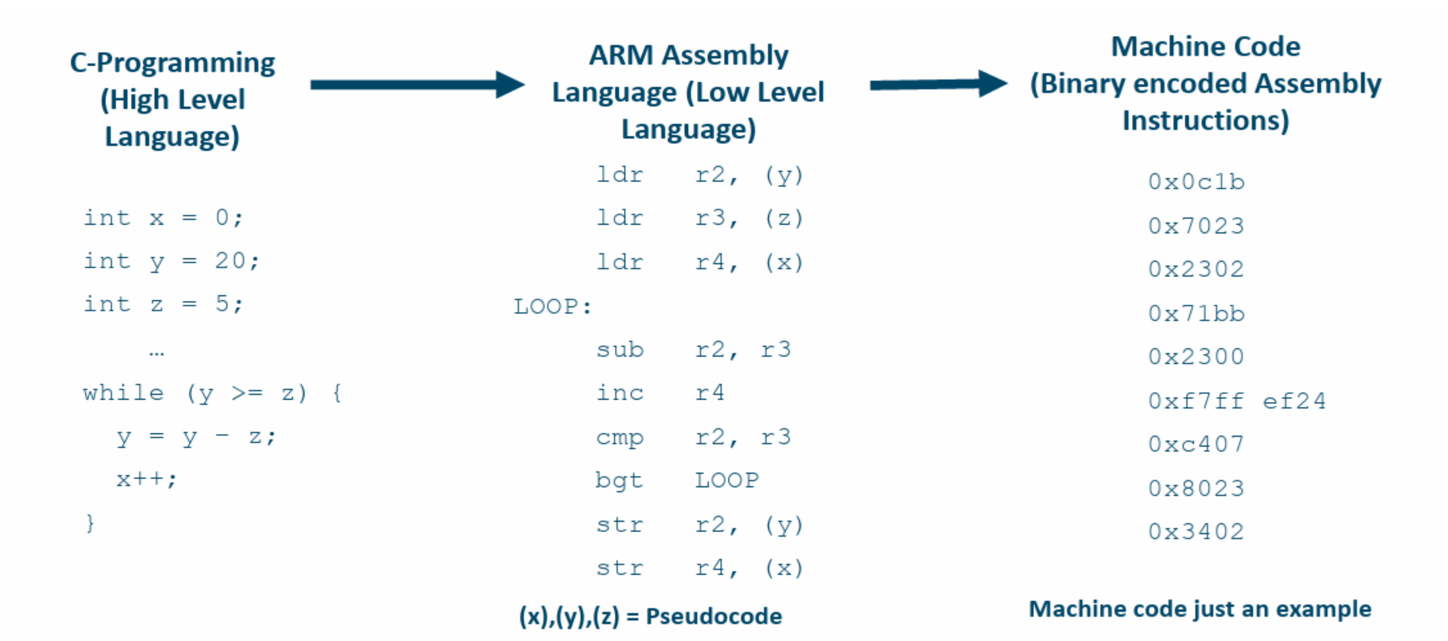
Native compiler:

The code generation and running executable happened in the same platform (host = target).

Cross compiler:

The code generation and running executable happened in different platforms (host ≠ target).

Compilation:

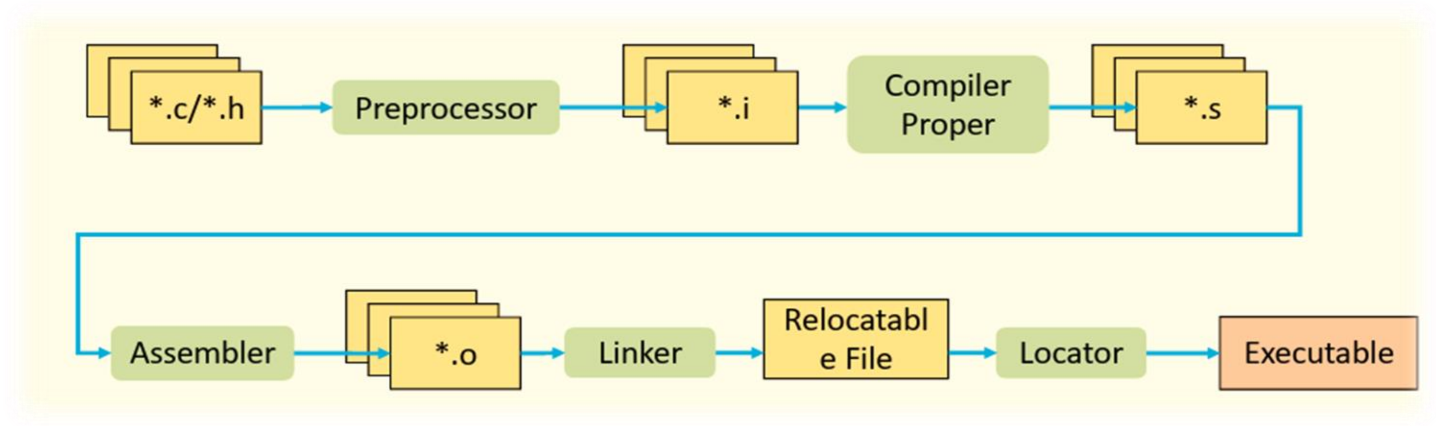

C-Programming (High Level Language)
ARM Assembly Language (Low Level Language)
Machine Code (Binary encoded Assembly Instructions)
(x),(y),(z) = Pseudocode
Machine code just an example

```
int x = 0;
int y = 20;
int z = 5;
    …
while (y >= z) {
  y = y - z;
  x++;
}
```

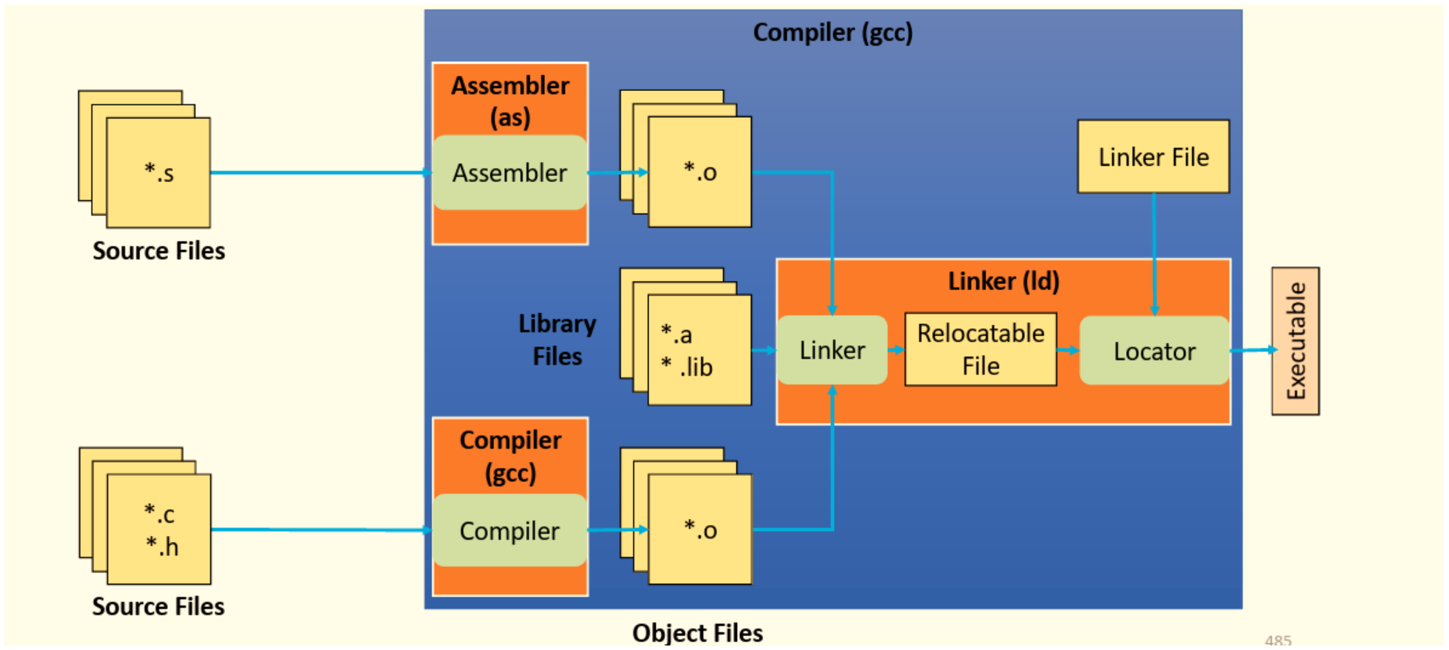
```
     ldr    r2, (y)
     ldr    r3, (z)
     ldr    r4, (x)
LOOP:
     sub    r2, r3
     inc    r4
     cmp    r2, r3
     bgt    LOOP
     str    r2, (y)
     str    r4, (x)
```

```
0x0c1b
0x7023
0x2302
0x71bb
0x2300
0xf7ff ef24
0xc407
0x8023
0x3402
```

Toolchain:



Compilation with linking:



Extensions:

Assembly Files	→	.s Extension
Object Files	→	.o Extension
Library Files	→	.a Extension (with .h)
Executable File	→	Extension Varies

1: Preprocessor (cpp):

```
cpp file_name.c > file_name.i
```

2: compiler (gcc,g++):

```
gcc -S file_name.i
```

3: assembler (as)

```
as -o file_name.s file_name.o
```

4: linker (ld)

```
ld -o file_name.exe file_name.o ...libraries...
```

You can compile file directly using (use -v to show some details):

```
gcc -o file_name.exe file_name.c
gcc -v -o file_name.exe file_name.c
```

Preprocessor directives:

Every line starting with hash # is preprocessor directive whether it is library header file or user defined header file.

If user defined header file is not in the same folder, you should write its path (you can type path from current file).

```
#include "C:\dir_name1\dir_name\file_name"
```

It is better to assign some values in macros or parametrized macro.

```
#define PI 3.14159
```

```
#define add(num1,num2) (num1+num2)
```

function like macro

```
/**
 * @brief Macros vs Functions
 * Macros :
 * 1) Macros are pre-processed -> Processed before your program compiles.
 * 2) No type checking (incompatible operand) is done -> Errors/side-effects in some cases.
 * 3) Macros do not check for compilation error (if any).
 * 4) Difficult to debug as they cause simple replacement.
 * 5) Using Macro increases the code length
 * 6) Speed of Execution using Macro is Faster
 * 7) Macros are useful when small code is repeated many times
 * 8) Macro does not check any Compile-Time Errors
 *
 * Functions :
 * 1) Functions are not preprocessed but "compiled".
 * 2) There is a type checking -> Warning if issue found
 * 3) Check for compilation error (if any).
 * 4) Easy to debug and the the stack frame located in the stack memeory.
 * 5) Using Function keeps the code length unaffected
 * 6) Speed of Execution using functions is slower than macros
 * 7) Functions are useful when large code is to be written
 * 8) Macro does not check any Compile-Time Errors
 *
 */
```

Conditional directive:

```
#if expression1
//code
#elif expression2
//code
#else
//code
#endif
```

File guard (in header file):

```
#ifndef expression
#define expression
//code
#endif
```

```
#ifdef expression

#error "text"

#endif
```

```
#line newline_number "file_name"
//code
```

Pragma: compiler dependent

```
#pragma once
```

like file guard

```
#pragma GCC poison identifier_name
```

prevent user from using specific word.

```
#pragma GCC warning "text"
```

throw warning.

```
#pragma GCC error "text"
```

throw error.

```
#if defined expresstion
//code
#else
//code
#endif
```

you can use operators with it.

String operator:

```
#define name(name) #name
```

replace with a string