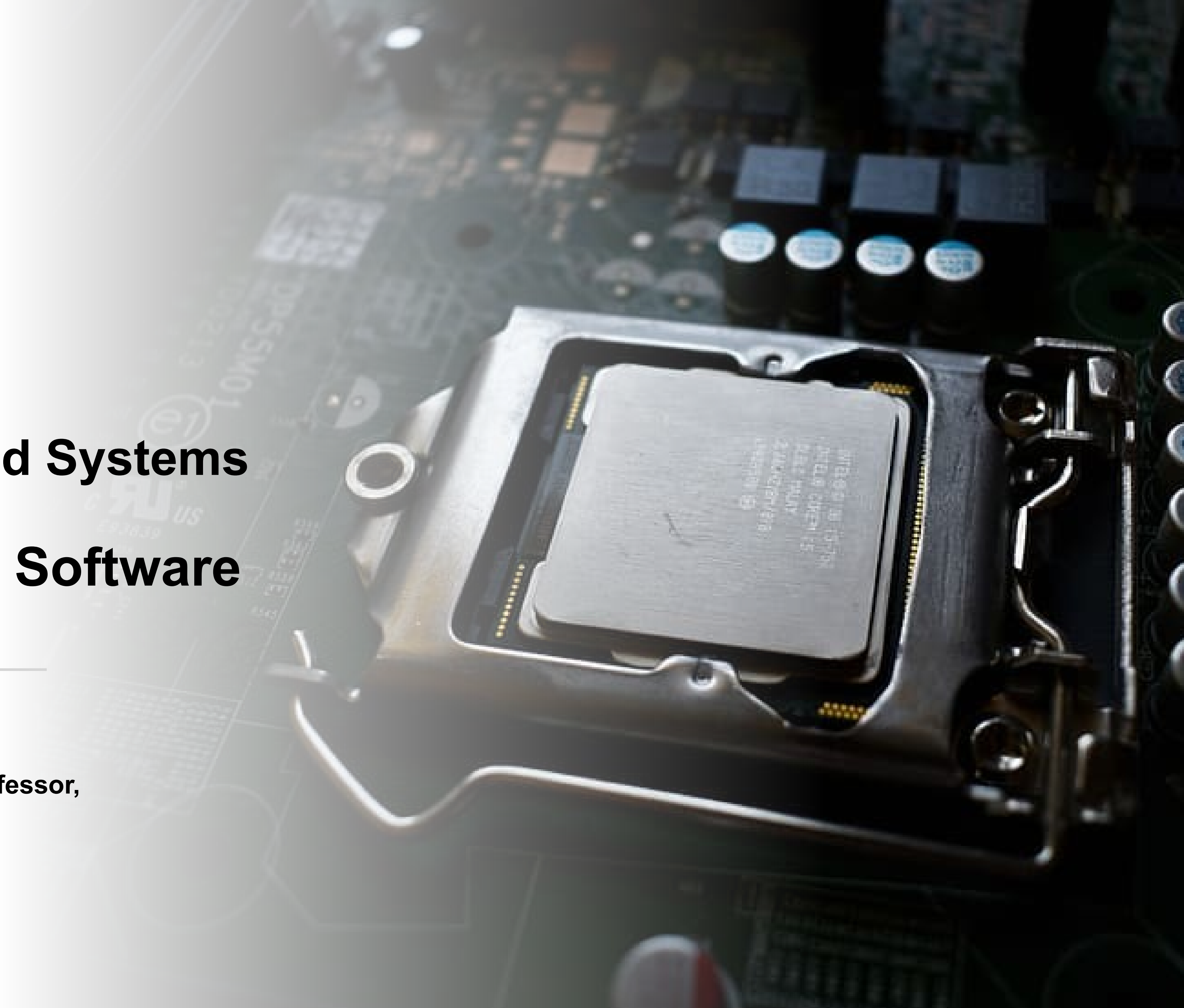
EECS 388:

Embedded Systems

Lecture 2: Embedded Software Development



Instructor: Tamzidul Hoque, Assistant Professor, Dept. of EECS, University of Kansas (hoque@ku.edu), office: Eaton 2038

Announcements

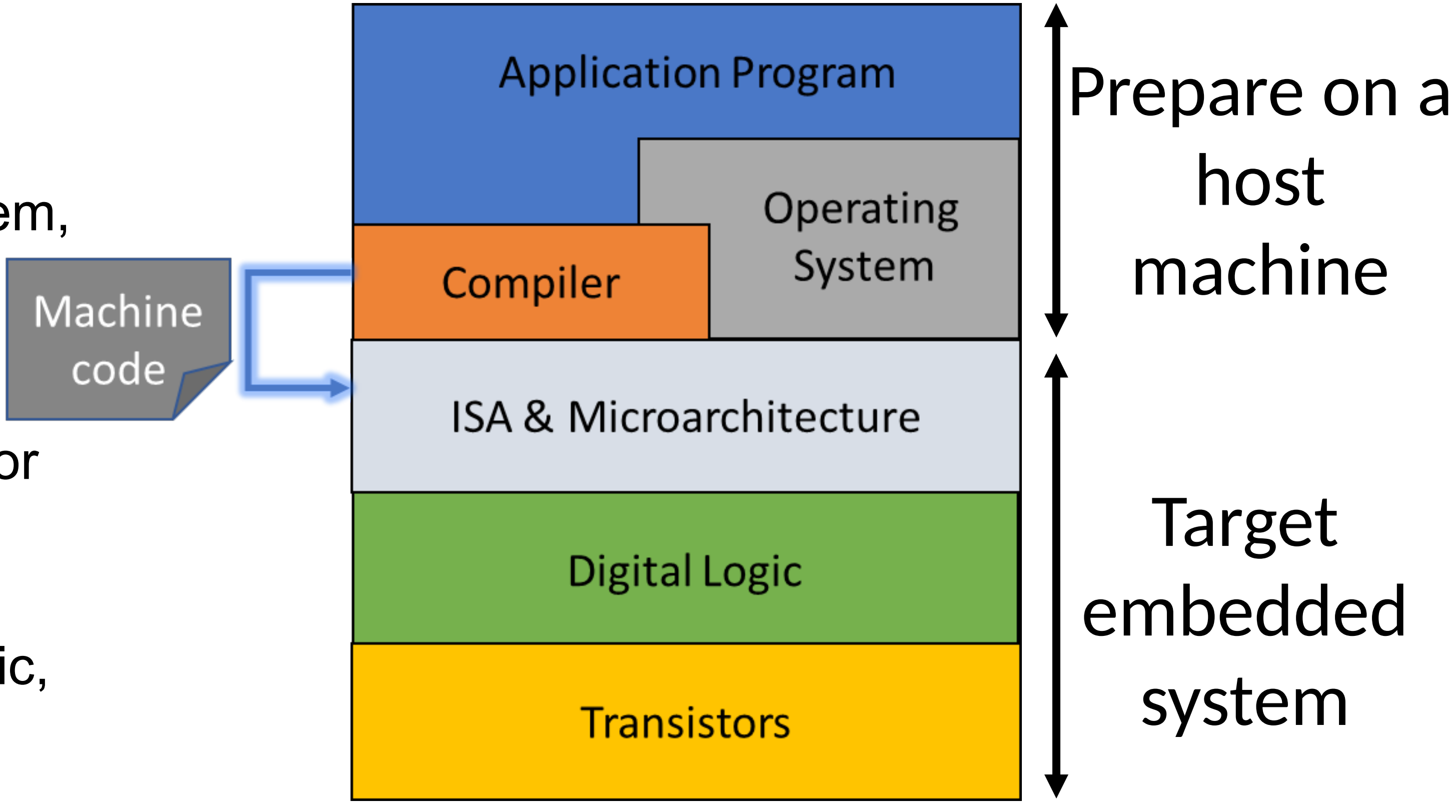
- **Labs and office hours start from next week**
- **Lab section instructors:**

Time		GTA NAME
Tu 09:00 - 10:50 AM EATN 3002 - LAWRENCE		Soma
Th 09:00 - 10:50 AM EATN 3002 - LAWRENCE		Ishraq
M 10:00 - 11:50 AM EATN 3002 - LAWRENCE		Ishraq
F 01:00 - 02:50 PM EATN 3002 - LAWRENCE		Soma
M 03:00 - 04:50 PM EATN 3002 - LAWRENCE		Ahsan
W 03:00 - 04:50 PM EATN 3002 - LAWRENCE		Ahsan

Absence in lab without valid reason will result in 30% penalty.

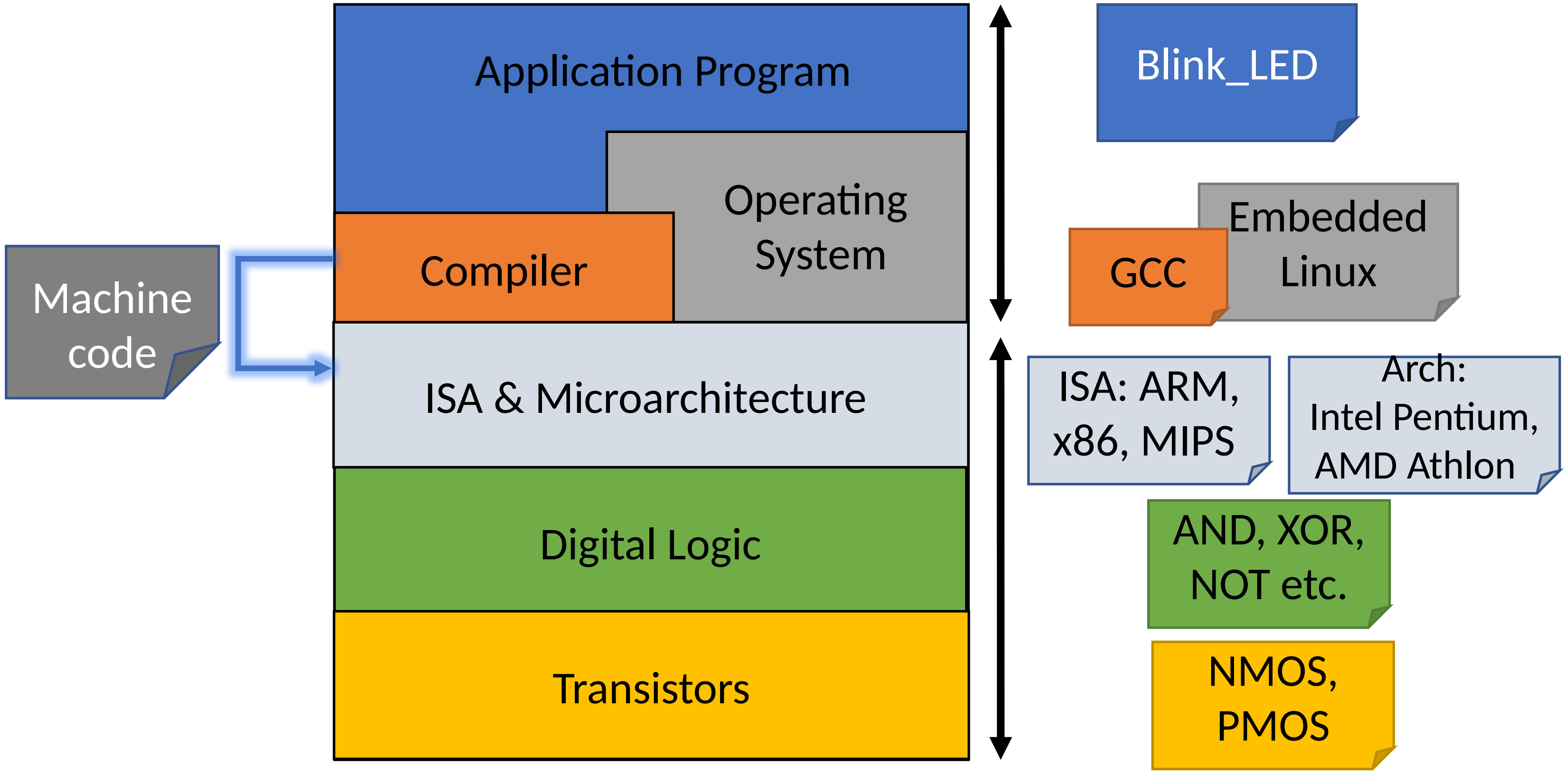
Context

- We can look at embedded systems from different abstraction
- **Software Abstraction:**
  - Application programs, operating system, compiler.
- **Hardware:**
  - Primary component: microprocessor or microcontroller
  - Abstractions: microarchitecture and instruction set architecture, digital logic, transistor



ISA: Instruction Set Architecture

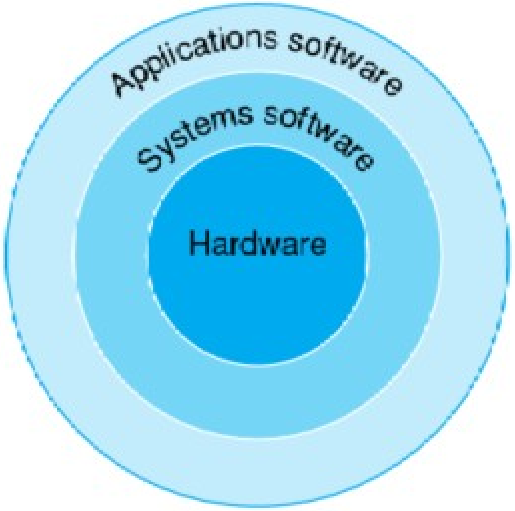
Examples

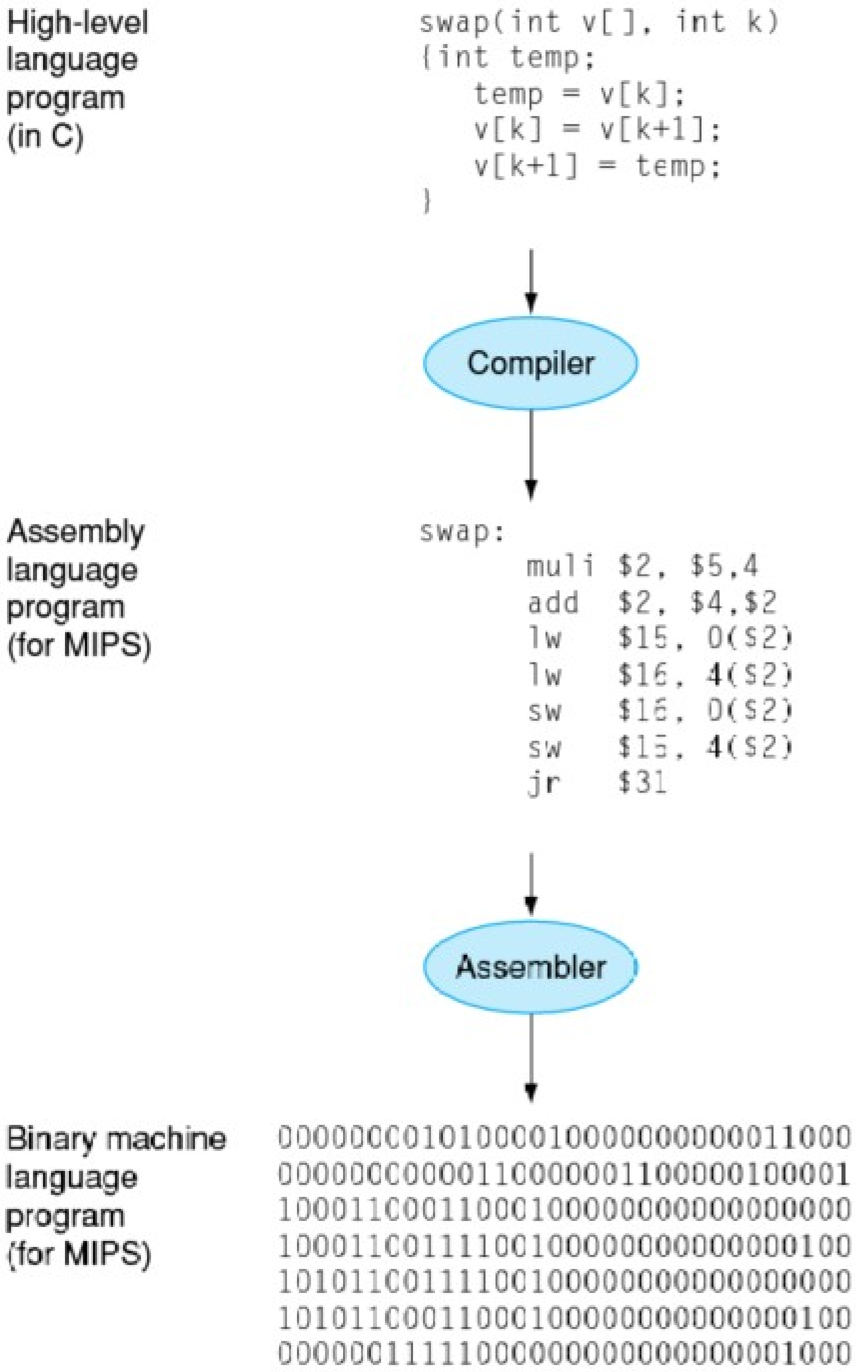


ISA: Instruction Set Architecture

Example

- **Application software:** Word processor, Internet browser
- **System Software:** Operating system, compiler
  - **OS**: A supervising software that manages hardware resources to run the application software.
  - **Compiler**: Translates programs to machine executable binary
- **Hardware:** Processor, memory, IO





High-level language program (in C)

```
swap(int v[], int k)
{int temp;
   temp = v[k];
   v[k] = v[k+1];
   v[k+1] = temp;
}
```

Assembly language program (for MIPS)

```
swap:
      muli $2, $5,4
      add  $2, $4,$2
      lw   $15, 0($2)
      lw   $16, 4($2)
      sw   $16, 0($2)
      sw   $15, 4($2)
      jr   $31
```

Binary machine language program (for MIPS)

```
00000000101000010000000000011000
00000000000110000001100000100001
10001100011000100000000000000000
10001100111100100000000000000100
10101100111100100000000000000000
10101100011000100000000000000100
00000011111000000000000000001000
```

Instruction Set Architecture (ISA)

- ISA acts as an interface between the hardware and the software
- Provides a model/abstraction of the hardware that can be controlled by writing programs in assembly language
- ISA can be considered as a manual for the assembly programmer.
- The ISA specifies the:
  - memory organization,
  - register set, and
  - instruction set (opcodes, data types, and addressing modes)
- ARM, x86, MIPS, SPARC, and PowerPC

More reading: ***1.7.4 The ISA:*** *Introduction to Computing Systems, 3/e Yale N. Patt,*

Analogy

- ISA of a CPU lets the programmer knows the required information to control the hardware by writing a program

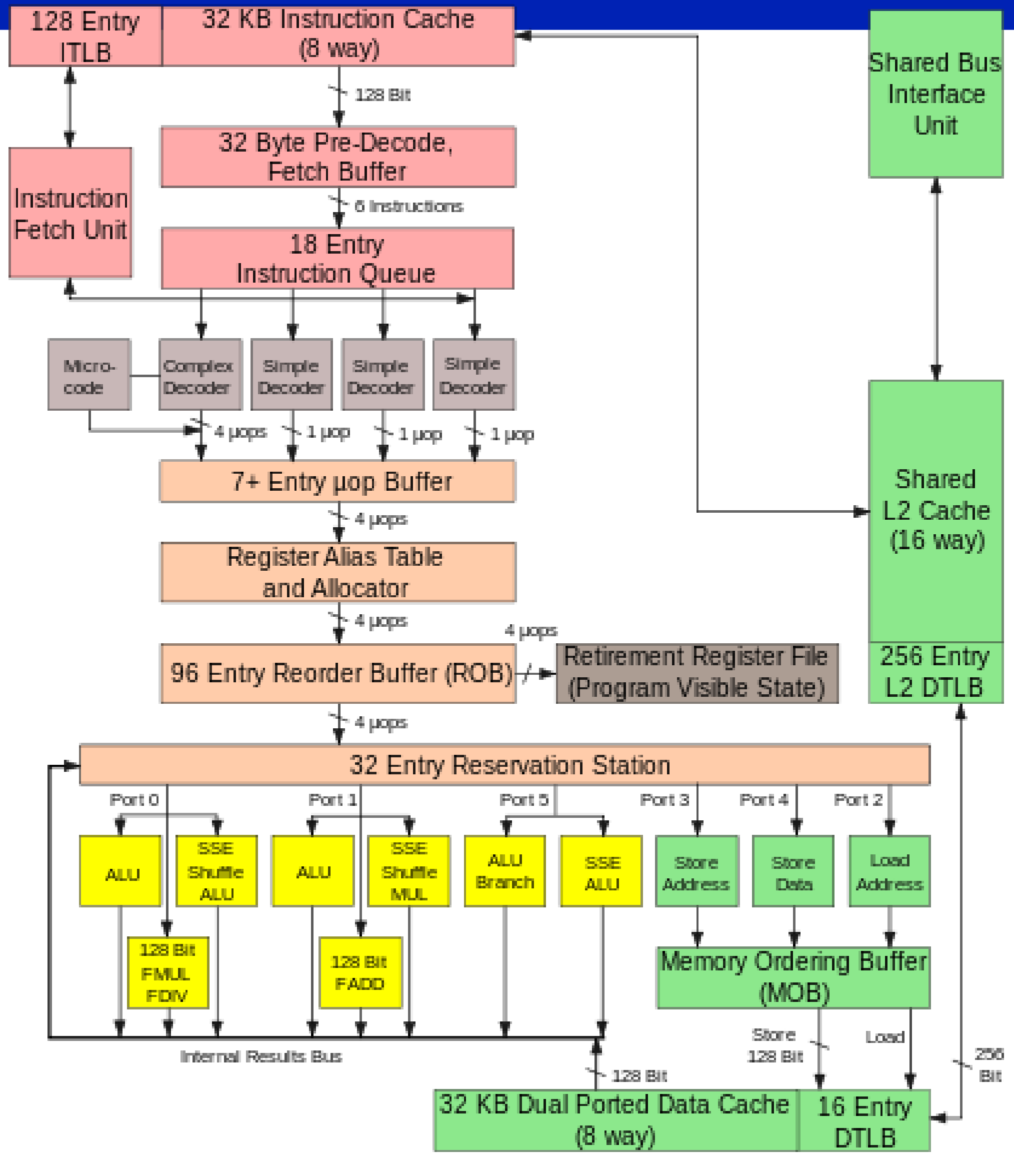
- ISA of a car describes what the driver needs to know to make the car carry out the driver’s wishes
  - How to use the gas and break paddle
  - How to use the steering, indicators, and horn

- Under the same ISA, different cars can be designed
  - Six vs eight cylinders ☾ ISA remains same

- When we need different ISA for a car?
  - Manual vs automatic transmission

Microarchitecture

- Diagrams that describe the interconnections of the microarchitectural elements
- Implementation of the ISA
- A given ISA may be implemented with different microarchitectures.
- x86-64 ISA implemented by Intel and AMD have different microarchitecture



Intel Core 2 Architecture

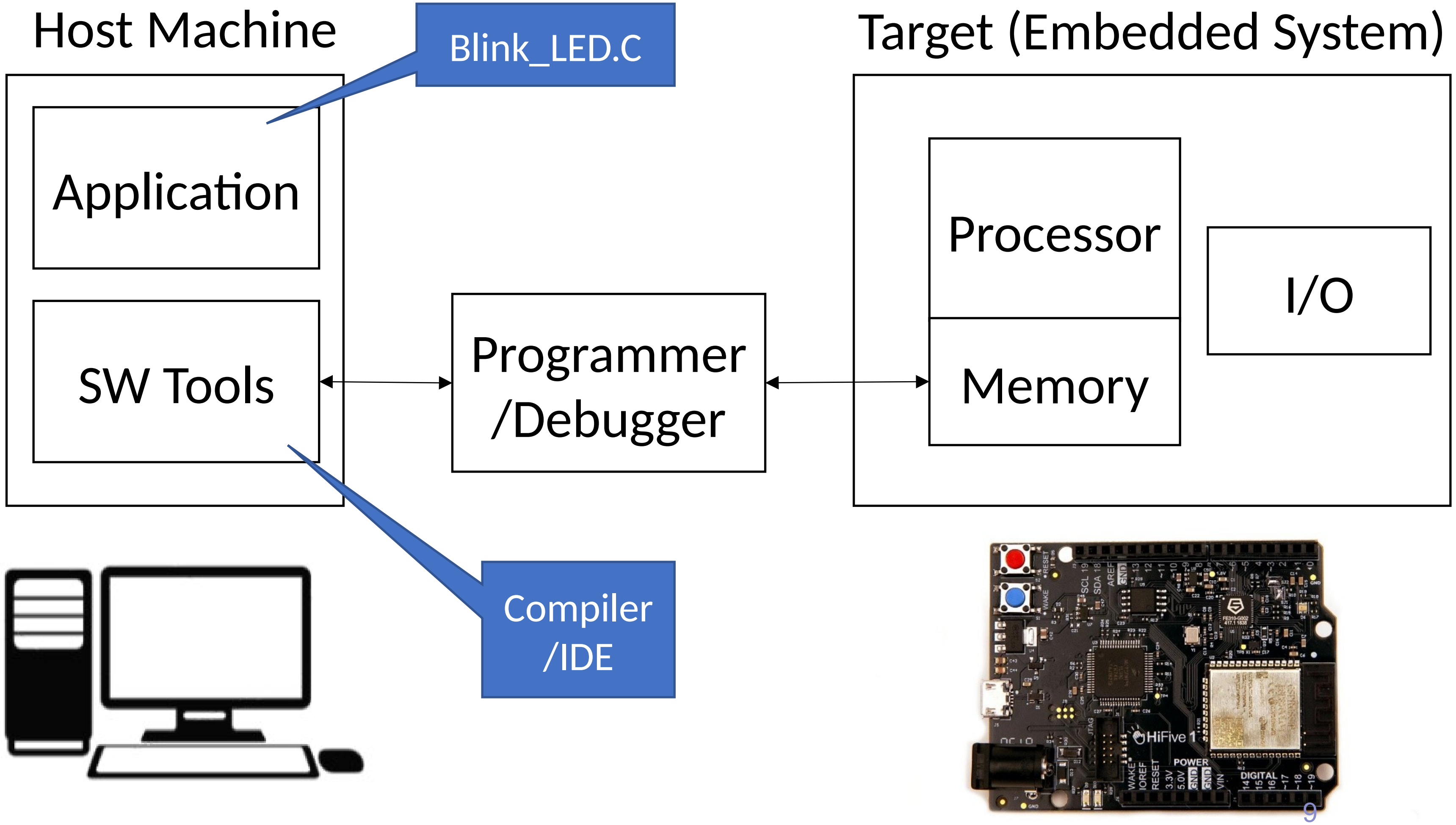
https://en.wikipedia.org/wiki/Microarchitecture

Embedded System Development Platform

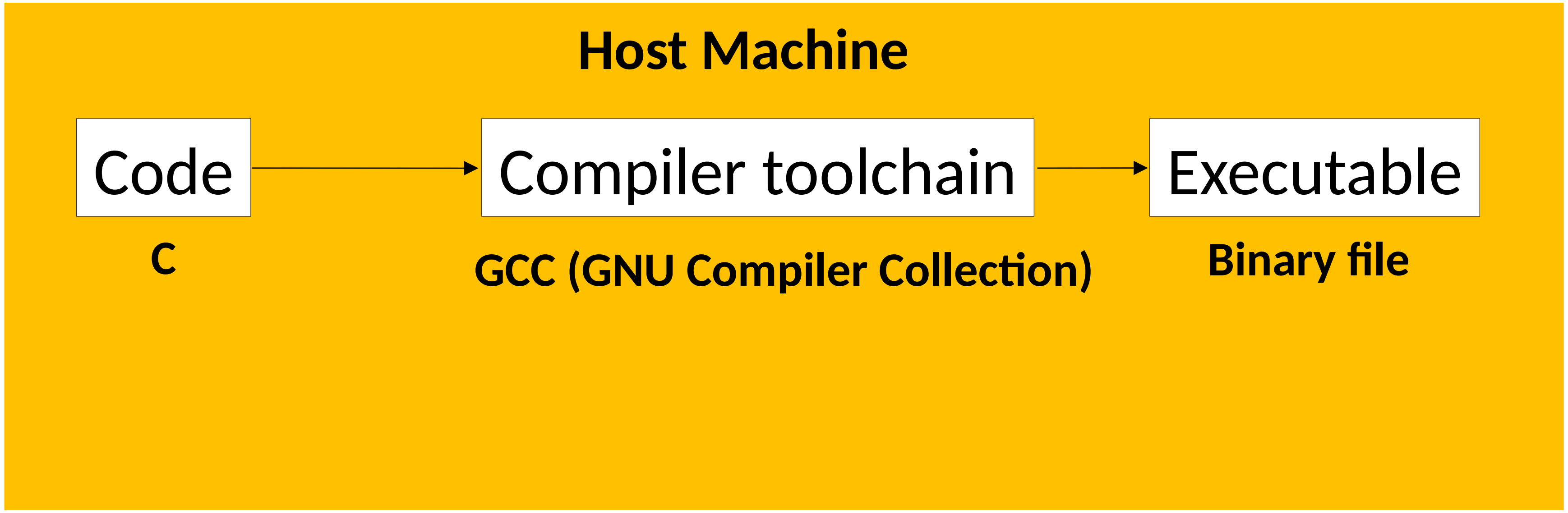
- When you want to program an embedded system, you first write the application program (in C) on a host machine.

- The host machine has a compiler that generates the machine code.

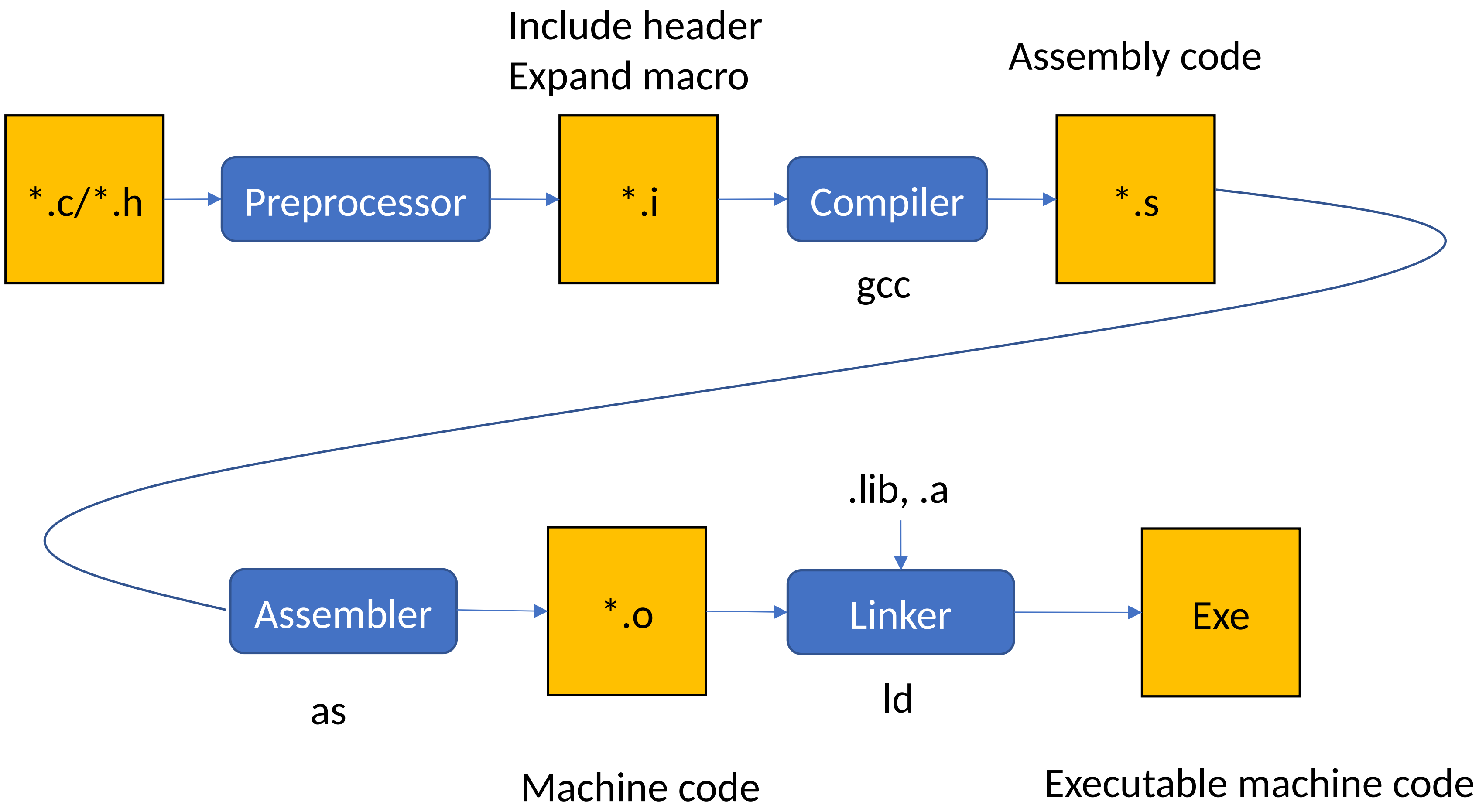
- This machine code will only run on this specific device or other devices with the same arch.



Due to the limited resource of the target, host machine usually contains our build environment



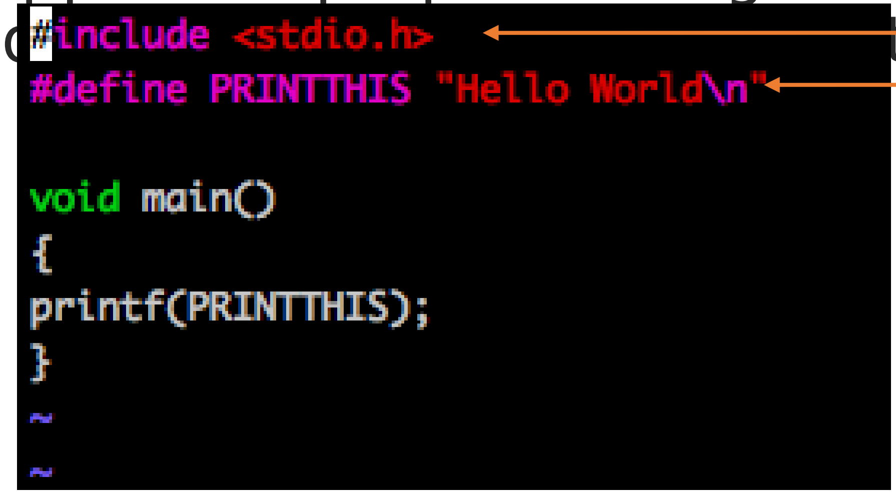
Compiler Toolchain



More reading: ***11.4.3 The Compilation Process:*** *Introduction to Computing Systems, 3/e Yale N. Patt,*

Preprocessing

- Preprocessors enables the inclusion of header files, macro expansions etc.
- Sometimes it is a separate program invoked by the compiler as the first part of translation.
- What happens in preprocessing: Removal of Comments, Expansi the included header files



```
#include <stdio.h>
#define PRINTTHIS "Hello World\n"

void main()
{
printf(PRINTTHIS);
}
~
~
```

Header file

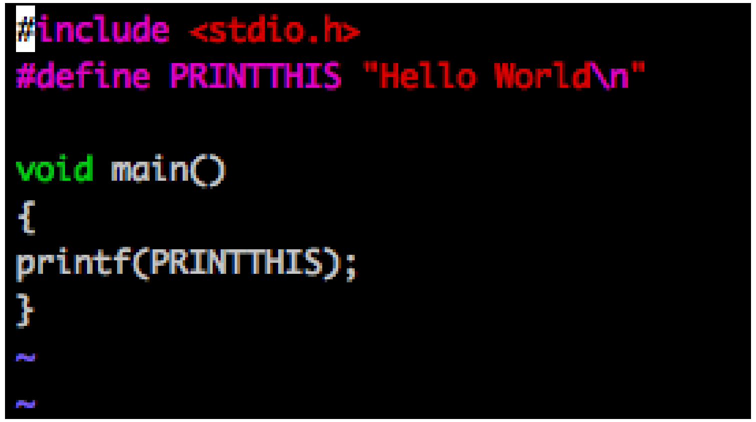
Macros

- All preprocessing directives begin with a # symbol in your program
- The first directives (#include <stdio.h>) requests a header file (stdio.h), to be included into the source code
- Macro ☾ fragments of codes that is given a name

Source: Link

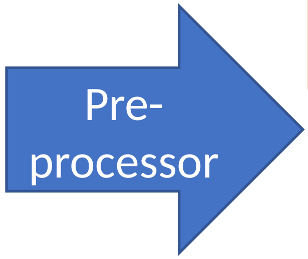
Example of Preprocessing Step

- The header file and macro definition are expanded after preprocessing

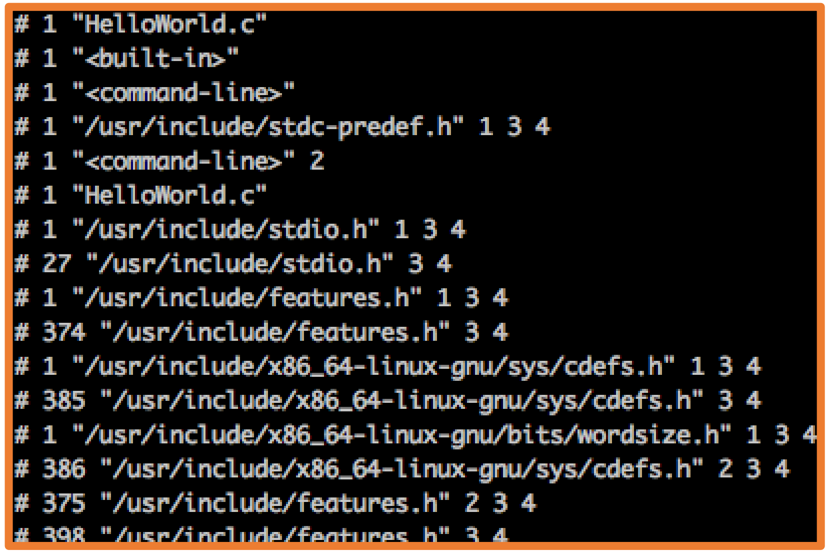


```
#include <stdio.h>
#define PRINTTHIS "Hello World\n"

void main()
{
printf(PRINTTHIS);
}
~
~
```

Input C code

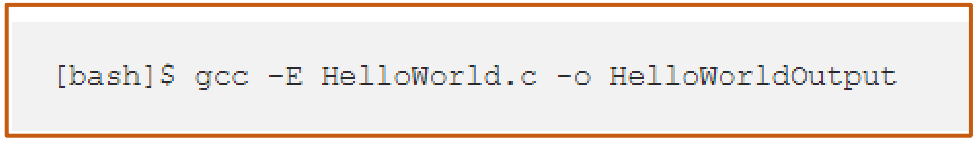


Pre-processor



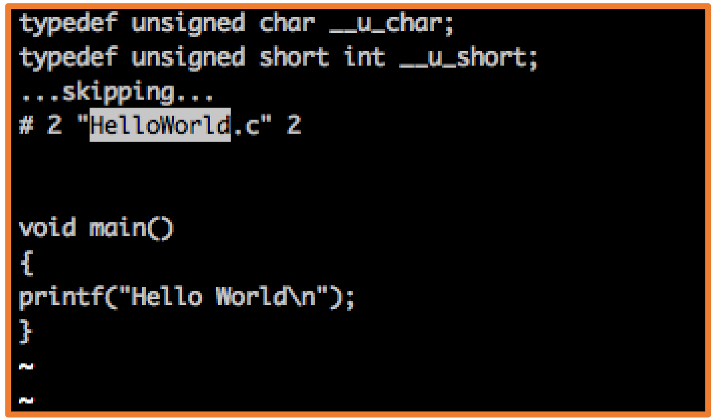
```
# 1 "HelloWorld.c"
# 1 "<built-in>"
# 1 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 1 "<command-line>" 2
# 1 "HelloWorld.c"
# 1 "/usr/include/stdio.h" 1 3 4
# 27 "/usr/include/stdio.h" 3 4
# 1 "/usr/include/features.h" 1 3 4
# 374 "/usr/include/features.h" 3 4
# 1 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 1 3 4
# 385 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 3 4
# 1 "/usr/include/x86_64-linux-gnu/bits/wordsize.h" 1 3 4
# 386 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 2 3 4
# 375 "/usr/include/features.h" 2 3 4
# 398 "/usr/include/features.h" 3 4
```

Expanded <stdio.h>



```
[bash]$ gcc -E HelloWorld.c -o HelloWorldOutput
```

Pre-processing is done by GCC's "-E" flag

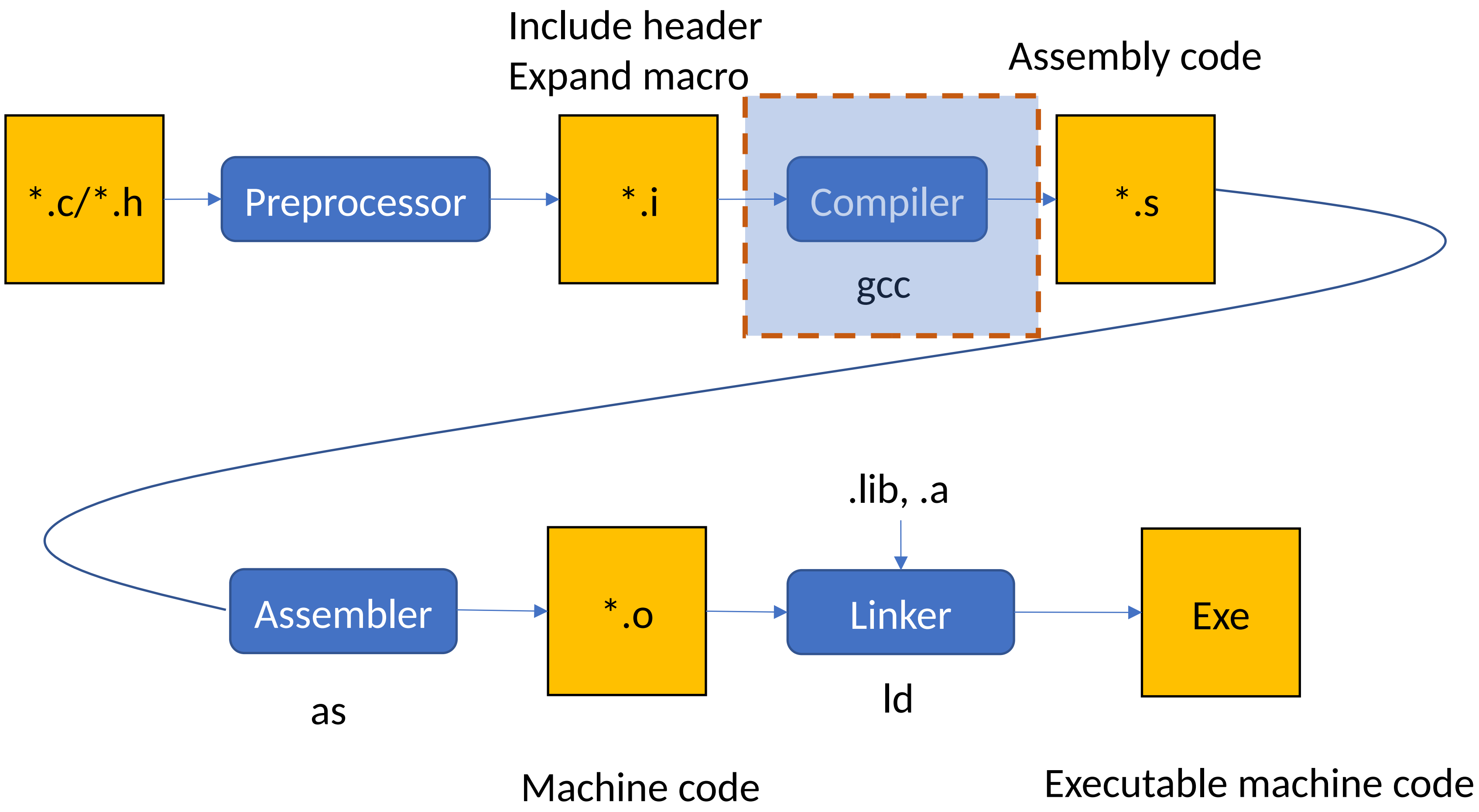


```
typedef unsigned char __u_char;
typedef unsigned short int __u_short;
...skipping...
# 2 "HelloWorld.c" 2


void main()
{
printf("Hello World\n");
}
~
~
```

Expanded macro

Source: Link

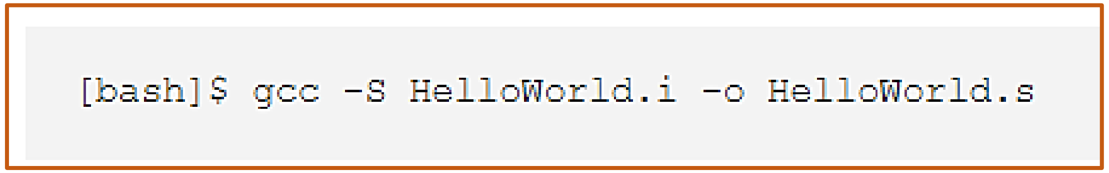
Compiler Toolchain



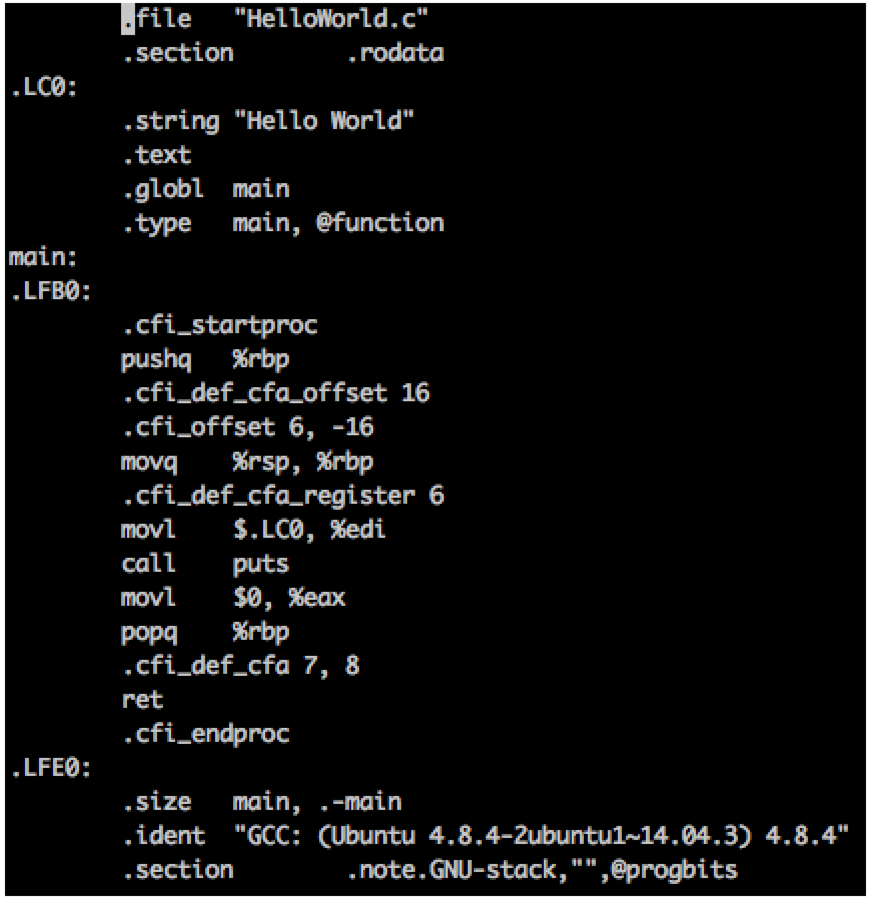
More reading: ***11.4.3 The Compilation Process:*** *Introduction to Computing Systems, 3/e Yale N. Patt,*

Compilation

- Takes the Preprocessed file as input, compiles it and produces an intermediate compiled output.
- The output file for this stage produces machine dependent Assembly code.
- Invoked by –S flag



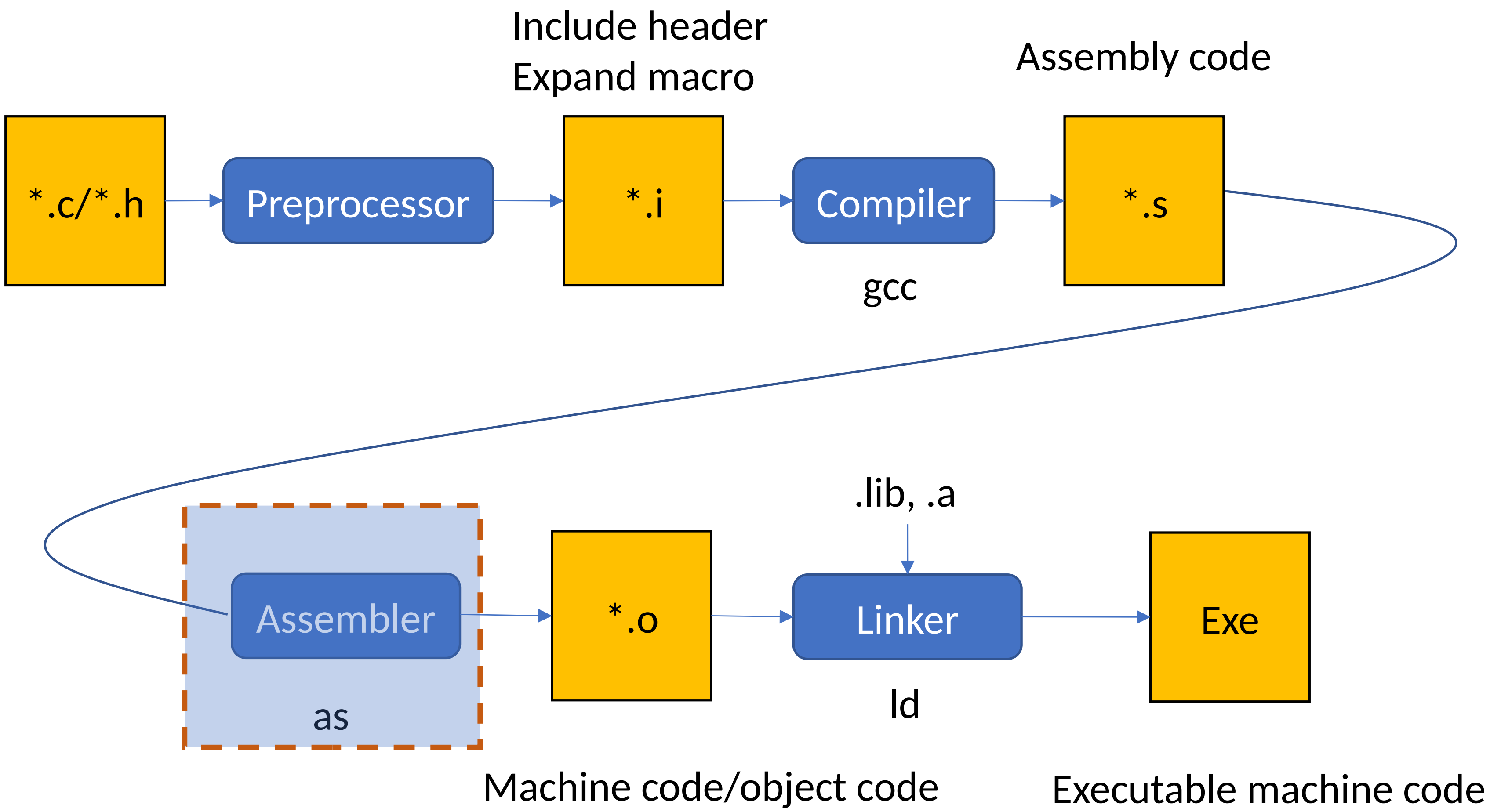
```
[bash]$ gcc -S HelloWorld.i -o HelloWorld.s
```



```
        .file   "HelloWorld.c"
        .section        .rodata
.LC0:
        .string "Hello World"
        .text
        .globl  main
        .type   main, @function
main:
.LFB0:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        movl    $.LC0, %edi
        call    puts
        movl    $0, %eax
        popq    %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE0:
        .size   main, .-main
        .ident  "GCC: (Ubuntu 4.8.4-2ubuntu1~14.04.3) 4.8.4"
        .section        .note.GNU-stack,"",@progbits
```

Source: Link

Compiler Toolchain

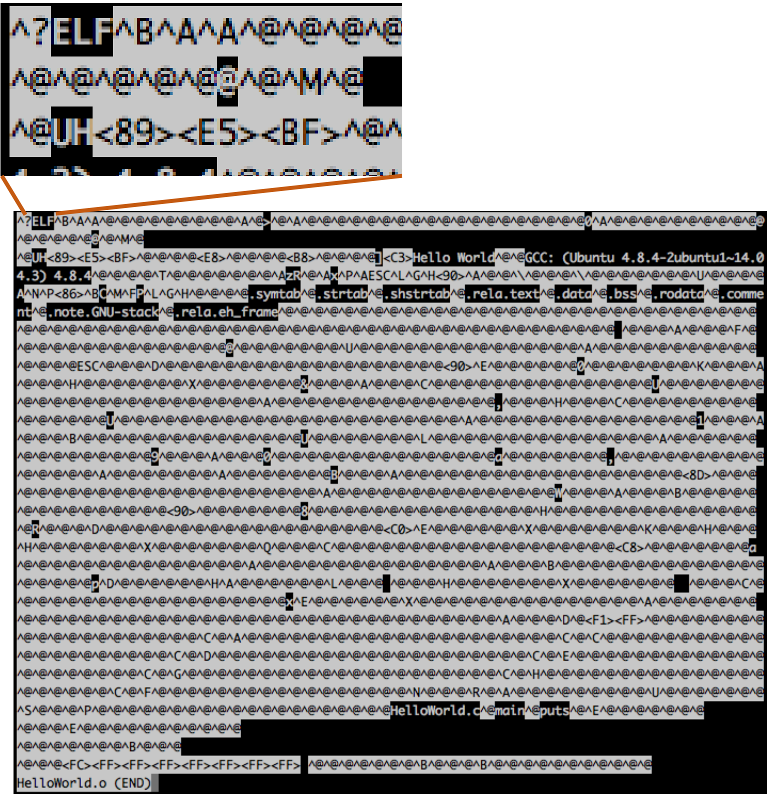


More reading: ***11.4.3 The Compilation Process:*** *Introduction to Computing Systems, 3/e Yale N. Patt,*

Assembly

- Converts assembly code into object code (binary file unreadable by texteditor).
- Object code is a portion of machine code that hasn’t yet been linked into a complete program.
- Assembler leaves the addresses of the external functions (if any) undefined, to be filled in later by the Linker.
- After the linking is done, we get an executable file that can be executed on the CPU

ELF: executable and linkable format.



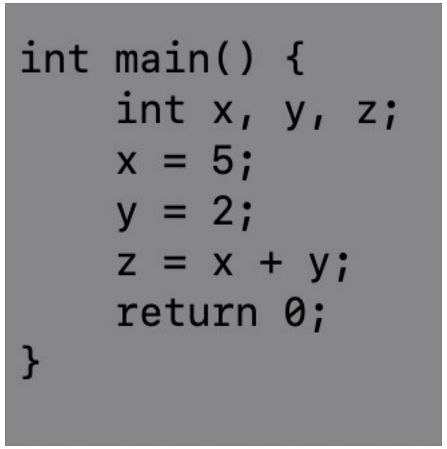
```
^?ELF^B^A^A^@^@^@^@
^@^@^@^@^@@^@^M^@
^@UH<89><E5><BF>^@^
```

```
^?ELF^B^A^A^@^@^@^@^@^@^@^@^@^A^@>^@^A^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@0^A^@^@^@^@^@^@^@^@^@^@@
^@^@^@^@^@@^@^M^@
^@UH<89><E5><BF>^@^@^@^@<E8>^@^@^@^@<B8>^@^@^@^@]<C3>Hello World^@^@GCC: (Ubuntu 4.8.4-2ubuntu1~14.0
4.3) 4.8.4^@^@^@^@^T^@^@^@^@^@^@^@^AzR^@^Ax^P^AESC^L^G^H<90>^A^@^@^\^@^@^@^\^@^@^@^@^@^@^@^U^@^@^@^@
A^N^P<86>^BC^M^FP^L^G^H^@^@^@^@.symtab^@.strtab^@.shstrtab^@.rela.text^@.data^@.bss^@.rodata^@.comme
nt^@.note.GNU-stack^@.rela.eh_frame^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@
^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@ ^@^@^@^A^@^@^@^F^@
^@^@^@^@^@^@^@^@^@^@^@^@^@^@@^@^@^@^@^@^@^@^U^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^A^@^@^@^@^@^@^@^@^@^@^@
^@^@^@^@ESC^@^@^@^D^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@<90>^E^@^@^@^@^@^@0^@^@^@^@^@^@^@^K^@^@^@^A
^@^@^@^H^@^@^@^@^@^@^@^X^@^@^@^@^@^@^@&^@^@^@^A^@^@^@^C^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@U^@^@^@^@^@^@^@
^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^A^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@,^@^@^@^H^@^@^@^C^@^@^@^@^@^@^@^@^@
^@^@^@^@^@^@U^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^A^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@1^@^@^@^A
^@^@^@^B^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@U^@^@^@^@^@^@^@^L^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^A^@^@^@^@^@^@
^@^@^@^@^@^@^@^@^@9^@^@^@^A^@^@^@0^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@a^@^@^@^@^@^@^@,^@^@^@^@^@^@^@^@^@^@
^@^@^@^@^@^A^@^@^@^@^@^@^@^A^@^@^@^@^@^@^@B^@^@^@^A^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@<8D>^@^@^@
^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^A^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@W^@^@^@^A^@^@^@^B^@^@^@^@^@
^@^@^@^@^@^@^@^@^@^@<90>^@^@^@^@^@^@^@8^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^H^@^@^@^@^@^@^@^@^@^@^@^@^@^@
^@R^@^@^@^D^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@<C0>^E^@^@^@^@^@^@^X^@^@^@^@^@^@^@^K^@^@^@^H^@^@^@
^H^@^@^@^@^@^@^@^X^@^@^@^@^@^@^@^Q^@^@^@^C^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@<C8>^@^@^@^@^@^@^@a
^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^A^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^A^@^@^@^B^@^@^@^@^@^@^@^@^@^@^@^@^@^@
^@^@^@^@^@p^D^@^@^@^@^@^@^H^A^@^@^@^@^@^@^L^@^@^@ ^@^@^@^H^@^@^@^@^@^@^@^X^@^@^@^@^@^@^@ ^@^@^@^C^@
^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@x^E^@^@^@^@^@^@^X^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^A^@^@^@^@^@^@^@
^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^A^@^@^@^D^@<F1><FF>^@^@^@^@^@^@^@^@^@
^@^@^@^@^@^@^@^@^@^@^@^C^@^A^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^C^@^C^@^@^@^@^@^@^@^@^@^@
^@^@^@^@^@^@^@^@^@^@^@^C^@^D^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^C^@^E^@^@^@^@^@^@^@^@^@^@^@^@
^@^@^@^@^@^@^@^@^@^C^@^G^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^C^@^H^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@
^@^@^@^@^@^@^C^@^F^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^N^@^@^@^R^@^A^@^@^@^@^@^@^@^@^@^U^@^@^@^@^@^@^@
^S^@^@^@^P^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@HelloWorld.c^@main^@puts^@^E^@^@^@^@^@^@
^@^@^@^E^@^@^@^@^@^@^@^@^@^@^@
^@^@^@^@^@^@^@^B^@^@^@
^@^@^@<FC><FF><FF><FF><FF><FF><FF><FF> ^@^@^@^@^@^@^@^B^@^@^@^B^@^@^@^@^@^@^@^@^@^@^@
HelloWorld.o (END)
```

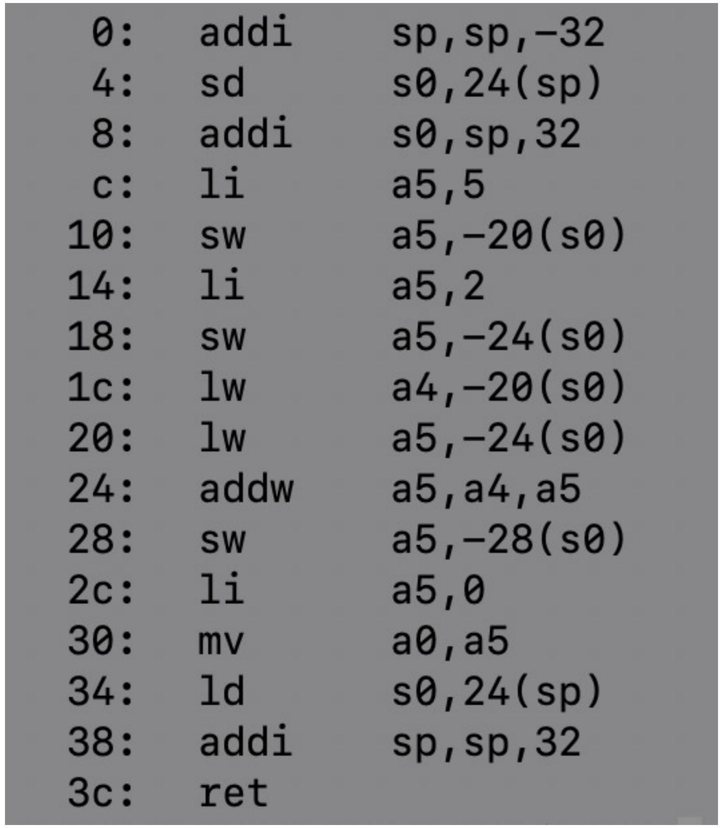
```
[bash]$ gcc -c HelloWorld.c -o HelloWorld.o
```

Source: Link

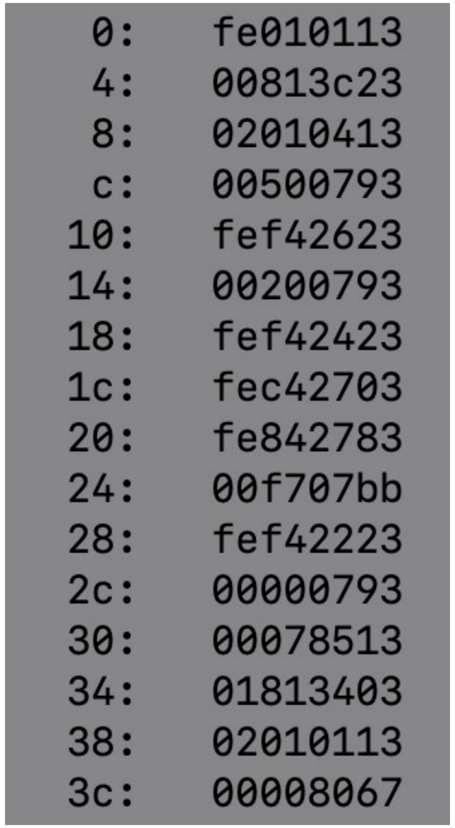
$$\text{C-code} \xrightarrow{\text{gcc -S}} \text{Assembly} \xrightarrow{\text{gcc -c}} \text{Binary}$$



```
int main() {
    int x, y, z;
    x = 5;
    y = 2;
    z = x + y;
    return 0;
}
```



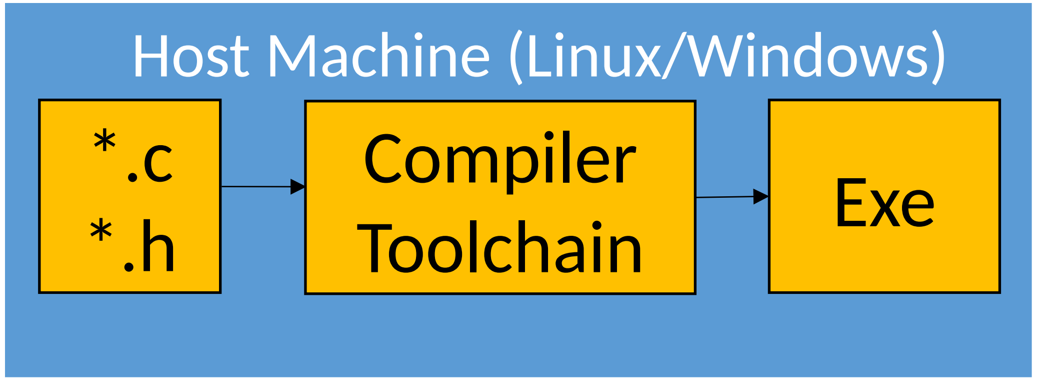
```
  0:   addi    sp,sp,-32
  4:   sd      s0,24(sp)
  8:   addi    s0,sp,32
  c:   li      a5,5
 10:   sw      a5,-20(s0)
 14:   li      a5,2
 18:   sw      a5,-24(s0)
 1c:   lw      a4,-20(s0)
 20:   lw      a5,-24(s0)
 24:   addw    a5,a4,a5
 28:   sw      a5,-28(s0)
 2c:   li      a5,0
 30:   mv      a0,a5
 34:   ld      s0,24(sp)
 38:   addi    sp,sp,32
 3c:   ret
```



```
  0:   fe010113
  4:   00813c23
  8:   02010413
  c:   00500793
 10:   fef42623
 14:   00200793
 18:   fef42423
 1c:   fec42703
 20:   fe842783
 24:   00f707bb
 28:   fef42223
 2c:   00000793
 30:   00078513
 34:   01813403
 38:   02010113
 3c:   00008067
```

Native Compilation

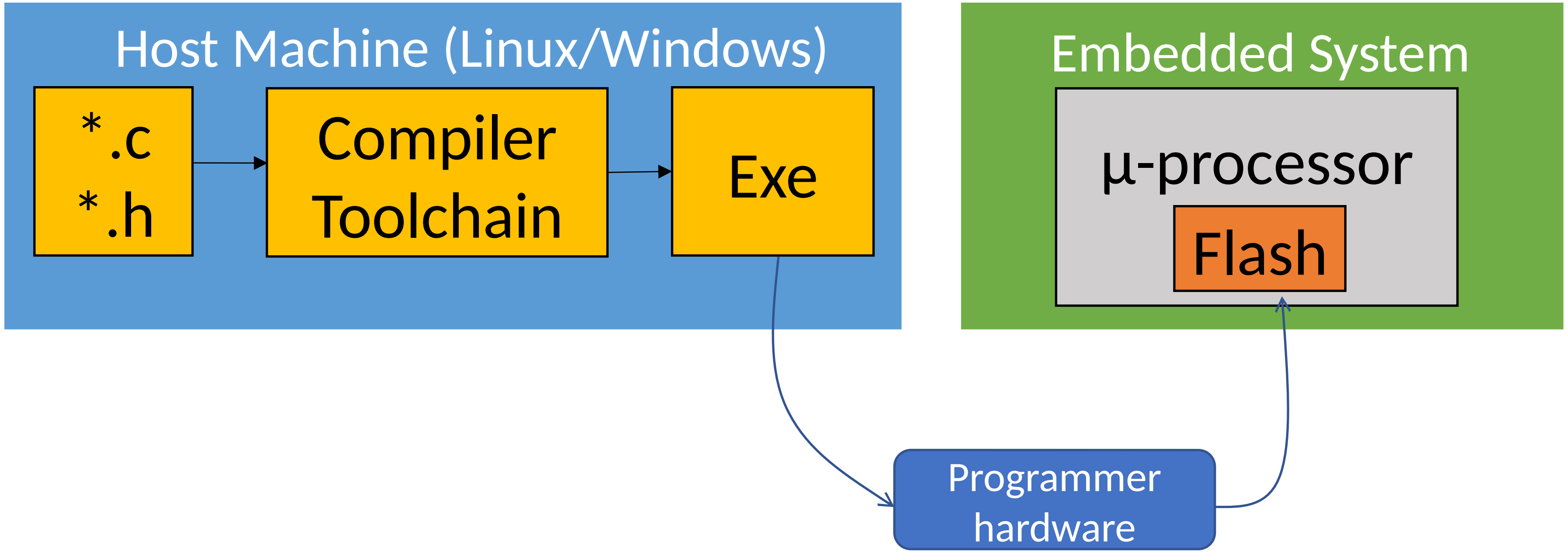
- Compile and run on one system



No extra hardware needed for sending/programming the .exe

Cross Compilation

- Compile on one system and run on another system



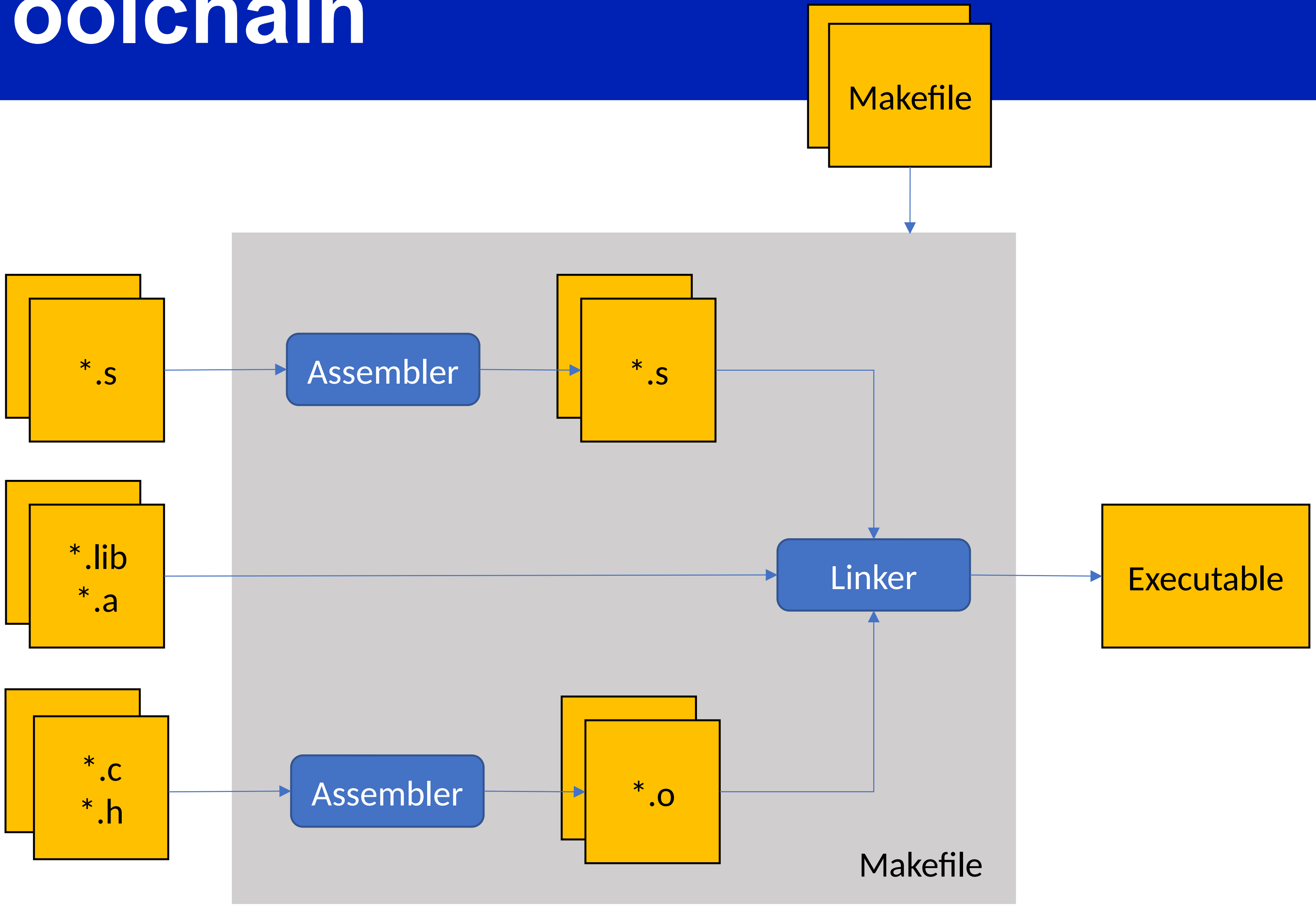
- Building can be too complex
    - Many gcc flags and commands
        - Linux has over 40k+ source code files!
    - Dependencies
    - Many source files
    - Many supported platforms

- Building manually is
    - Not scalable
    - Time consuming
    - Error prone

GNU Make

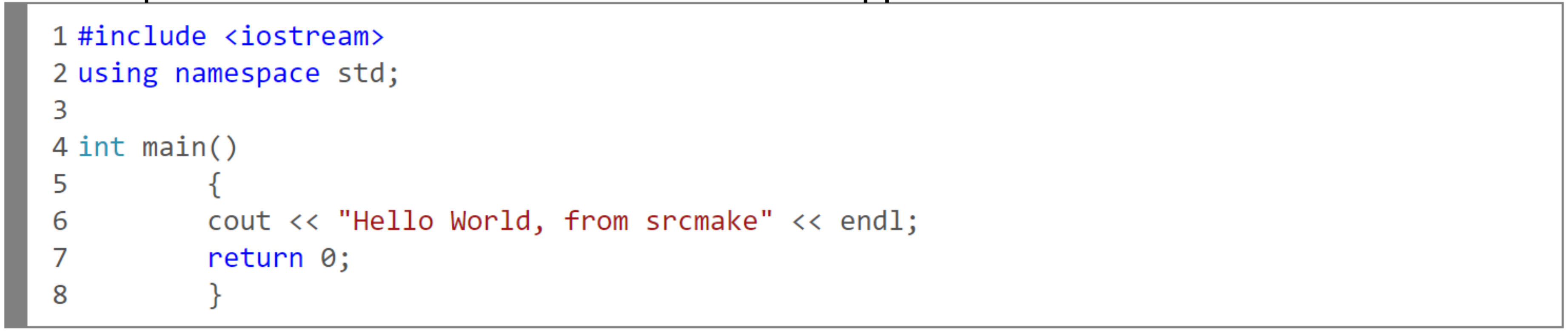
- “GNU Make is a tool which controls the generation of executables and other non-source files of a program from the program's source files”
  - Preprocessing
  - Compiling
  - Assembling
  - Linking

GNU Toolchain



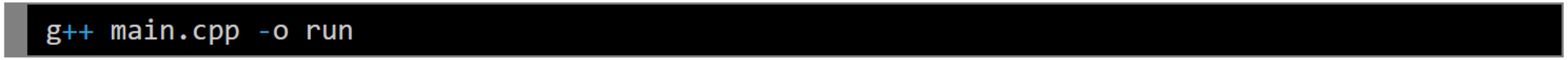
A Simple Makefile:

- Simple Hello World code in a file named main.cpp.



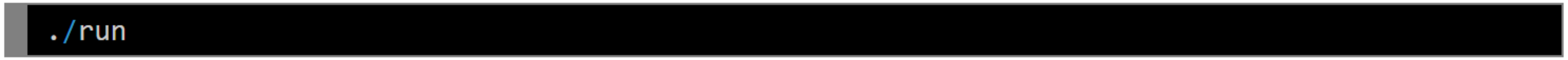
```cpp
#include <iostream>
using namespace std;

int main()
        {
        cout << "Hello World, from srcmake" << endl;
        return 0;
        }
```

- To compile this, we'd use the following command in a terminal:



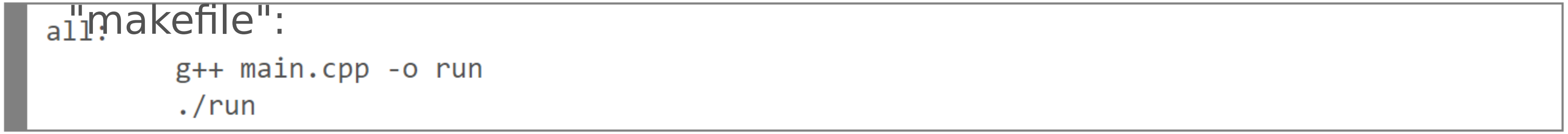
```
g++ main.cpp -o run
```

- To use run, we'd then do the following:



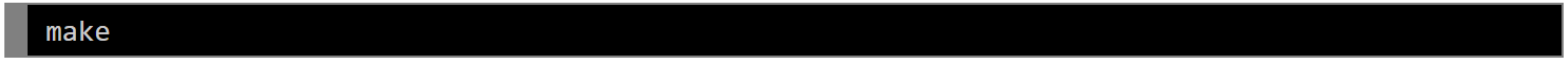
```
./run
```

- To create a make file, we just need to write the following into a file named "makefile":



```make
all:
	g++ main.cpp -o run
	./run
```

- To use our make file, all we need to do is type "make" into the terminal.

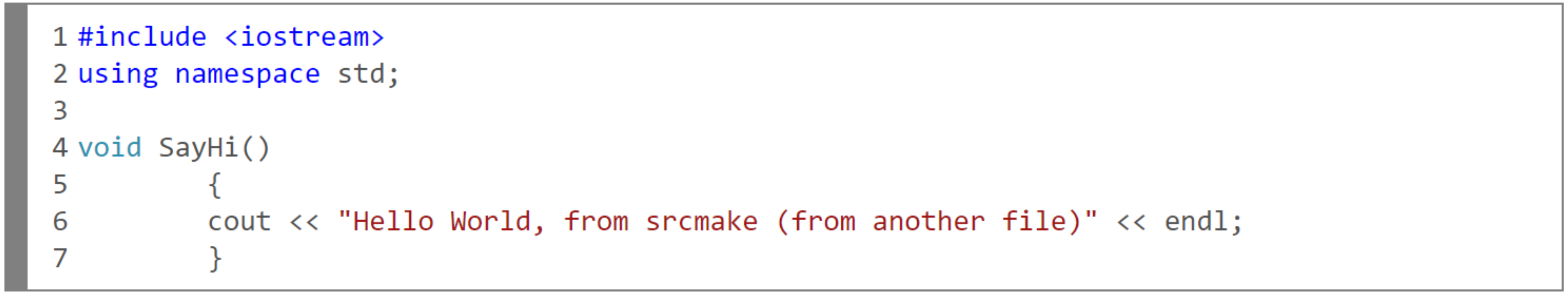


```
make
```

Source: link

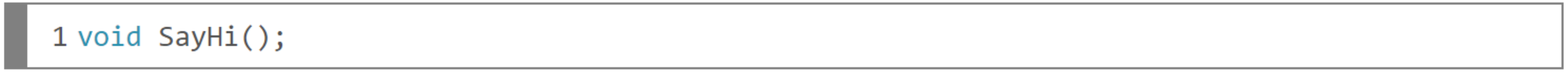
Example with Headers:

Say, we copy the code to a new file called **helper.cpp**, instead of the main.cpp



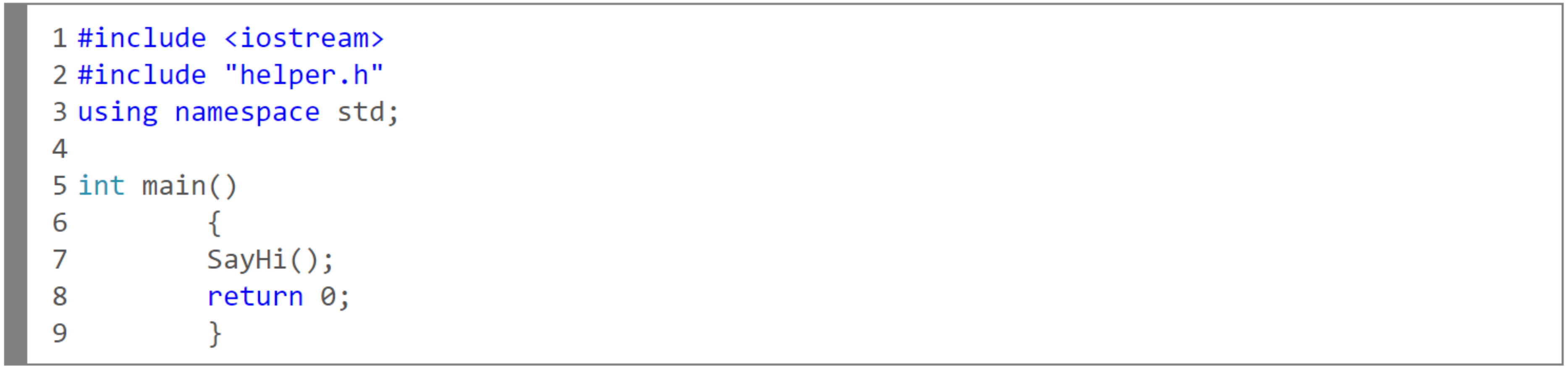
```cpp
#include <iostream>
using namespace std;

void SayHi()
        {
        cout << "Hello World, from srcmake (from another file)" << endl;
        }
```

we need a header file to match this .cpp file, so let's create a file called **helper.h**



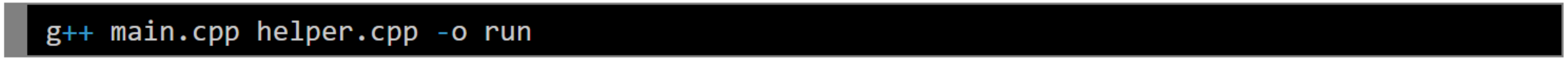
```cpp
void SayHi();
```

our **main.cpp** file will now include this header, and make use of the SayHi() function



```cpp
#include <iostream>
#include "helper.h"
using namespace std;

int main()
        {
        SayHi();
        return 0;
        }
```

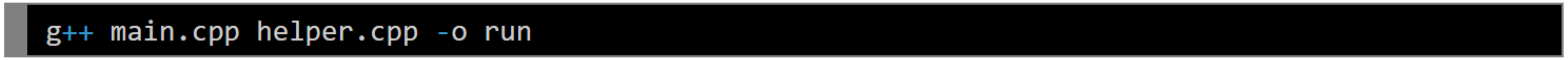
Now to compile this, we need the do the following:



```
g++ main.cpp helper.cpp -o run
```

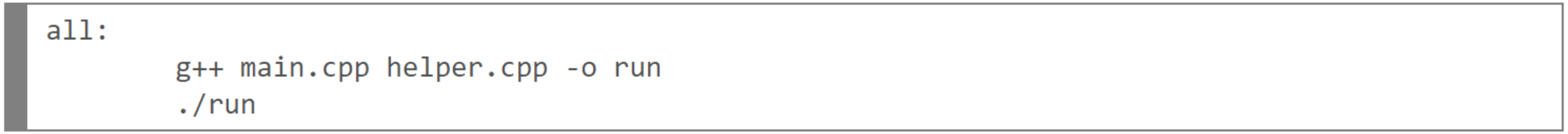
A Simple Makefile (Cont.):

Now to compile this, we need the do the following:



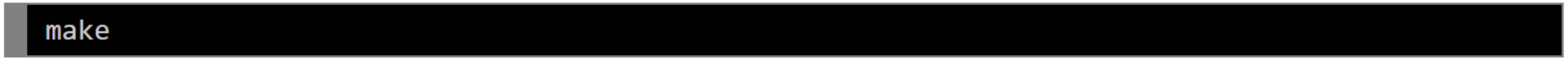
```
g++ main.cpp helper.cpp -o run
```

The make file that can be used here:



```
all:
        g++ main.cpp helper.cpp -o run
        ./run
```

Command that will run the makefile



```
make
```

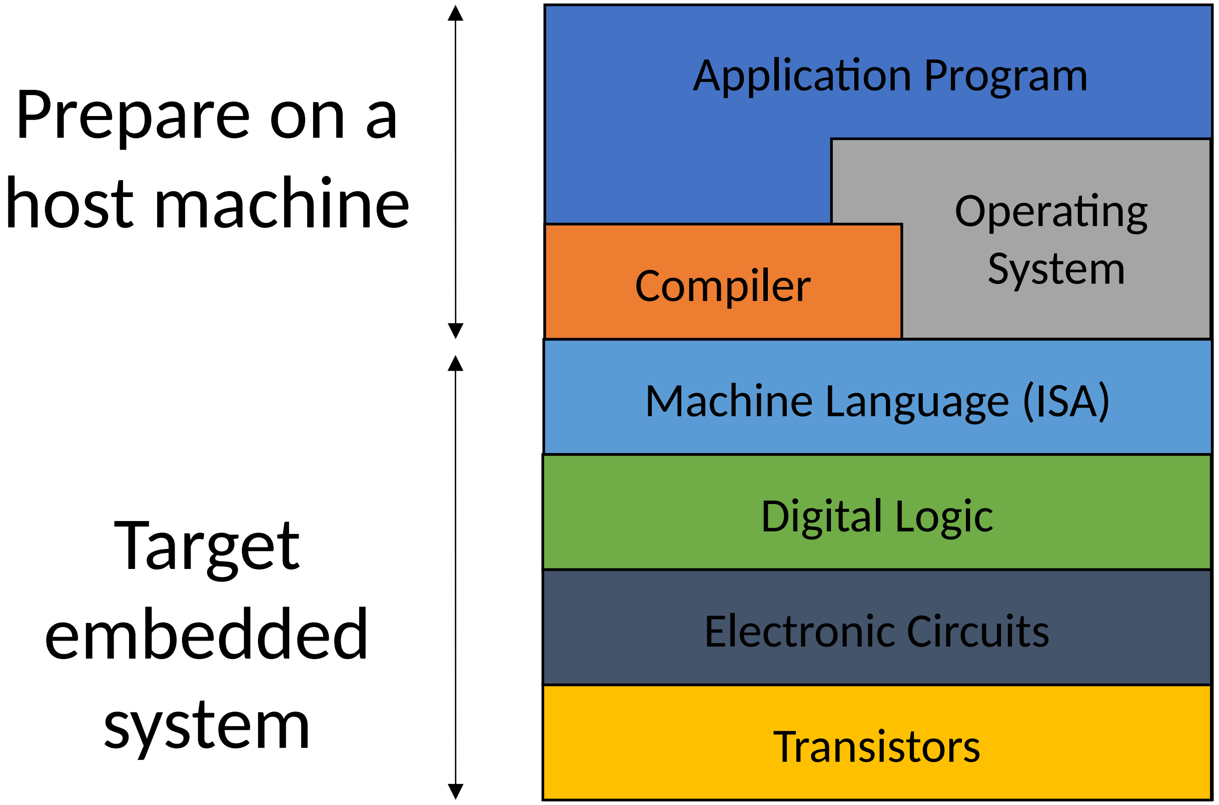
Integrated Development Environment (IDE)

- Autogenerate Makefiles
- Provide a very simple interface for developers (usually beginners)
    - Bad for maintainability and portability

- Professional software teams write their own makefile

- In the lab, for software development on the microcontroller, we will use Visual Studio Code (VSCode) and PlatformIO IDE combination

Recap



- Compiler toolchain
- Cross compilation
- GNU Make
- IDE