CSE 3113: Microprocessor and Assembly Language Lab

Slide Credit
Professor Dr. Upama Kabir

Computer Science and Engineering, University of Dhaka,

Lab 1

Required Software Tools

1. Install Keil for windows

   Inside

   here choose MDK-Arm
2. Install ST-LINK debugger for windows

   https:

Why Keil MDK

Keil MDK (Microcontroller Development Kit) is the complete software development environment for a range of Arm Cortex-M based microcontroller devices. MDK includes:

1. μVision IDE with Integrated Debugger, Flash programmer and the Arm® Compiler toolchains.
2. STM32CubeMX exports μVision projects.
3. FreeRTOS, RTX and Micrium are directly supported
4. Keil Middleware: Network, USB, Flash File and Graphics.
5. Arm Compiler 5 and Arm Compiler 6 (LLVM) are included. GCC is supported.

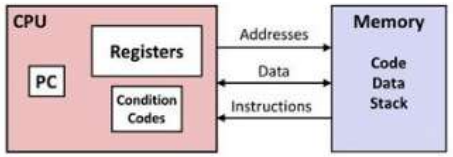
Levels of Abstraction

- C [and other high level languages] are easy for programmers to understand, but computers require lots of software to process them
- Machine code is just the opposite: easy for the computer to process, humans need lots of help to understand it
- Assembly language is a compromise between the two: readable by humans (barely), close correspondence to machine code

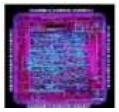
C programmer

```
#include <stdio.h>
int main(){
  int i, n = 10, t1 = 0, t2 = 1, nxt;
  for (i = 1; i <= n; ++i){
    printf("%d, ", t1);
    nxt = t1 + t2;
    t1 = t2;
    t2 = nxt; }
  return 0; }
```

Assembly programmer



Computer designer



Gates, clocks, circuit layout, ...

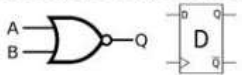


Figure 2

CMSIS

CMSIS stands for Cortex Microcontroller Software Interface Standard, a framework developed by ARM that provides a standardized way to write code for Arm Cortex-M based microcontrollers, simplifying software reuse and reducing development time.

What does it mean to compile code?

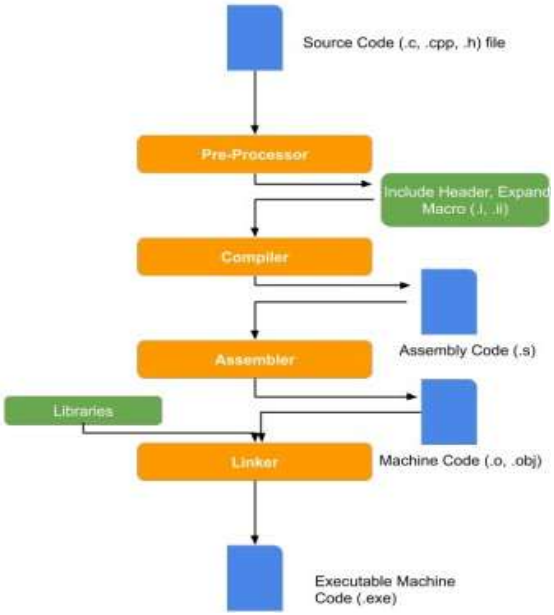


Figure 3

- Computer follows steps to translate your code into something the computer can understand
- This is the process of compiling code [a compiler completes these actions]
- Four steps: (i) preprocessing, (ii) compiling, (iii) assembling, (iv) linking

Pre-Processor

- Peculiar to the C family; other languages don't have this
- Processes #include, #define, #if, macros
  - Combines main source file with headers (textually)
  - Defines and expands macros (token-based shorthand)
  - Conditionally removes parts of the code (e.g. specialize for Linux, Mac, ...)
- Removes all comments
- Output looks like C still

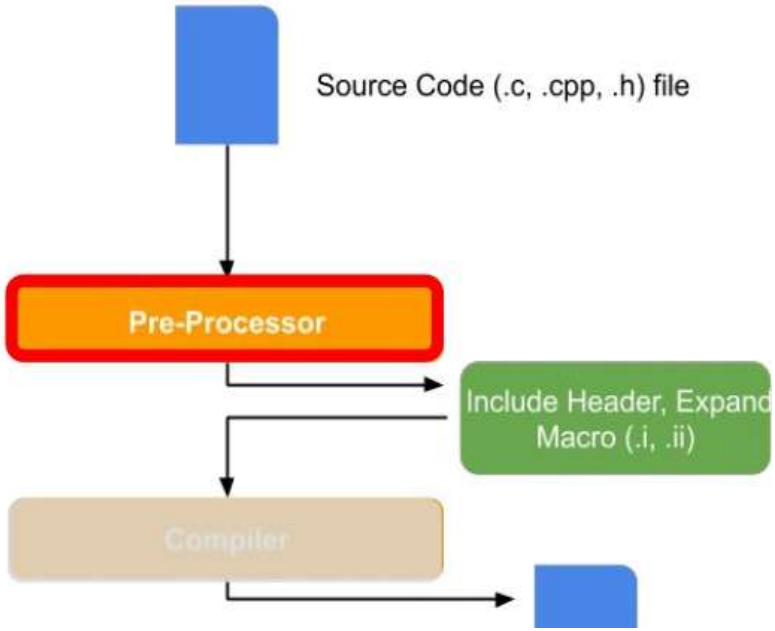


Figure 4

Before and after preprocessing

```c
#include <limits.h>
#include <stdio.h>

int main(void) {
    // Report the range of `char` on this system
    printf("CHAR_MIN = %d\n"
           "CHAR_MAX = %d\n",
           CHAR_MIN, CHAR_MAX);
    return 0;
}
```

- Contents of header files inserted inline
- Comments removed
- Macros expanded
- "Directive" lines (beginning with #) communicate things like original line numbers

```c
# 1 "test.c"
# 1 "/usr/lib/gcc/x86_64-linux-gnu/10/include/limits.h" 1 3 4
...
# 1 "/usr/include/stdio.h" 1 3 4
...
extern int fprintf (FILE *__restrict __stream,
        const char *__restrict __format, ...);
extern int printf (const char *__restrict __format, ...);
...
# 874 "/usr/include/stdio.h" 3 4
# 3 "test.c" 2

int main(void) {
    printf("CHAR_MIN = %d\n"
           "CHAR_MAX = %d\n",
# 6 "test.c" 3 4
           (-0x7f - 1)
# 6 "test.c"
                    , 0x7f);
    return 0;
}
```

Figure 5

C to Machine Code

Compiler

- The compiler translates the preprocessed code into assembly code
  - This changes the format and structure of the code but preserves the semantics (what it does)
  - Can change lots of details for optimization, as long as the overall effect is the same

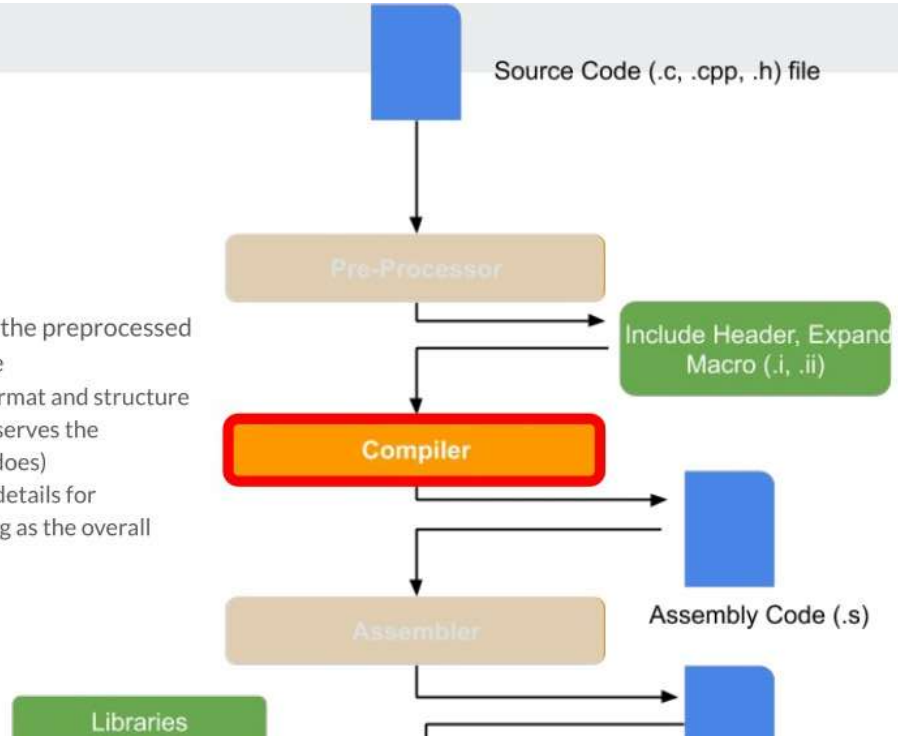


Figure 6

Before and after compilation

```c
extern int printf (const char *__restrict
                   __format, ...);
int main(void) {
    printf("CHAR_MIN = %d\n"
           "CHAR_MAX = %d\n",
           (-0x7f - 1), 0x7f);
    return 0;
}
```

- C *source code* converted to *assembly language*
- Textual, but 1:1 correspondence to machine language
- String out-of-line, referred to by label (`.LC0`)
- `printf` just referred to, not declared

```
    .file   "test.c"
    .section    .rodata.str1.1,"aMS",@progbits,1
.LC0:
    .string "CHAR_MIN = %d\nCHAR_MAX = %d\n"
    .text
    .globl  main
main:
    subq    $8, %rsp
    movl    $127, %edx
    movl    $-128, %esi
    leaq    .LC0(%rip), %rdi
    xorl    %eax, %eax
    call    printf@PLT
    xorl    %eax, %eax
    addq    $8, %rsp
    ret
    .size   main, .-main
```

Figure 7

C to Machine Code

Assembler

- Parses assembly code and mainly translates into bits
  - There is some flexibility to generate the most efficient version of machine code

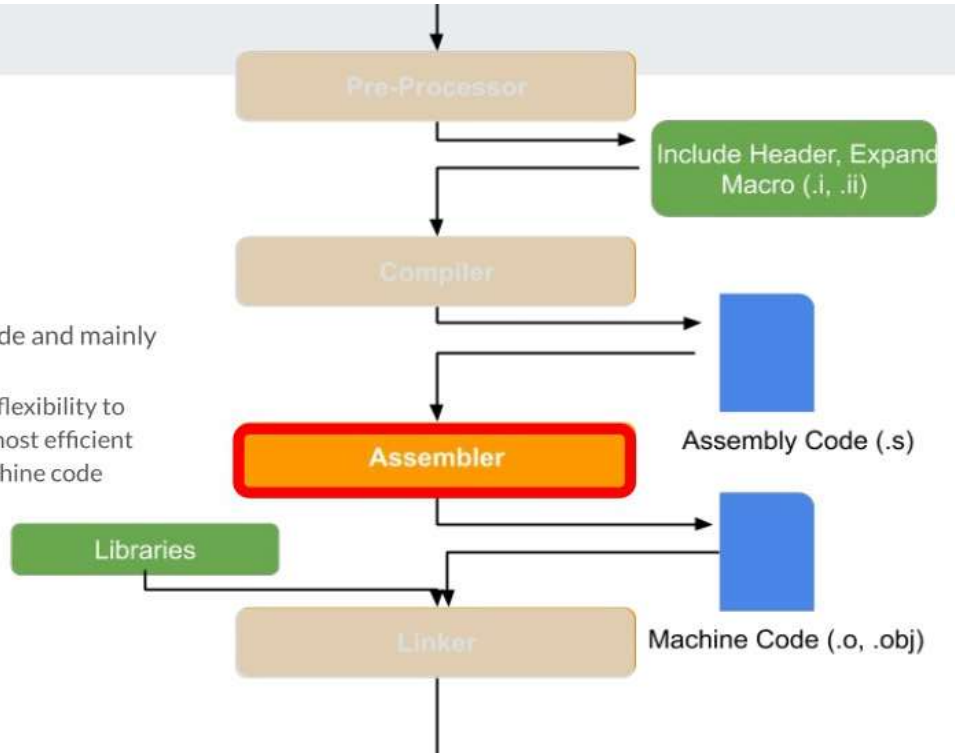


Figure 8

Before and after assembling

```
    .file   "test.c"
    .section    .rodata.str1.1,"aMS",@progbits,1
.LC0:
    .string "CHAR_MIN = %d\nCHAR_MAX = %d\n"
    .text
    .globl  main
main:
    subq    $8, %rsp
    movl    $127, %edx
    movl    $-128, %esi
    leaq    .LC0(%rip), %rdi
    xorl    %eax, %eax
    call    printf@PLT
    xorl    %eax, %eax
    addq    $8, %rsp
    ret
    .size   main, .-main
```

- Everything is now binary
- "Relocations" for addresses not yet known

```
$ objdump -s -r test.o
test.o: file format elf64-x86-64

RELOCATION RECORDS FOR [.text]:
OFFSET           TYPE              VALUE
0000000000000011 R_X86_64_PC32     .LC0-0x0000000000000004
0000000000000018 R_X86_64_PLT32    printf-0x0000000000000004

Contents of section .rodata.str1.1:
 0000 43484152 5f4d494e 203d2025 640a4348  CHAR_MIN = %d.CH
 0010 41525f4d 4158203d 2025640a 00        AR_MAX = %d..

Contents of section .text:
 0000 4883ec08 ba7f0000 00be80ff ffff488d  H.............H.
 0010 3d000000 0031c0e8 00000000 31c04883  =....1......1.H.
 0020 c408c3                               ...
```

Figure 9

Before and after assembling

```
    .file   "test.c"
    .section    .rodata.str1.1,"aMS",@progbits,1
.LC0:
    .string "CHAR_MIN = %d\nCHAR_MAX = %d\n"
    .text
    .globl  main
main:
    subq    $8, %rsp
    movl    $127, %edx
    movl    $-128, %esi
    leaq    .LC0(%rip), %rdi
    xorl    %eax, %eax
    call    printf@PLT
    xorl    %eax, %eax
    addq    $8, %rsp
    ret
    .size   main, .-main
```

- Just to emphasize that 1:1 correspondence between assembly and machine instructions

```
$ objdump -d -r test.o
test.o: file format elf64-x86-64
Disassembly of section .text.startup:

0000000000000000 <main>:
   0:   48 83 ec 08             sub    $0x8,%rsp
   4:   ba 7f 00 00 00          mov    $0x7f,%edx
   9:   be 80 ff ff ff          mov    $0xffffff80,%esi
   e:   48 8d 3d 00 00 00 00    lea    0x0(%rip),%rdi
                11: R_X86_64_PC32 .LC0-0x4
  15:   31 c0                   xor    %eax,%eax
  17:   e8 00 00 00 00          call   1c <main+0x1c>
                18: R_X86_64_PLT32    printf-0x4
  1c:   31 c0                   xor    %eax,%eax
  1e:   48 83 c4 08             add    $0x8,%rsp
  22:   c3                      ret
```

Figure 10

C to Machine Code

Linker

- Aggregates multiple independently compiled files containing machine code
- Fills in those unknown addresses
- The goal is to create 1 file with all of the needed code to run the program
  - This is the file you run to check your code!

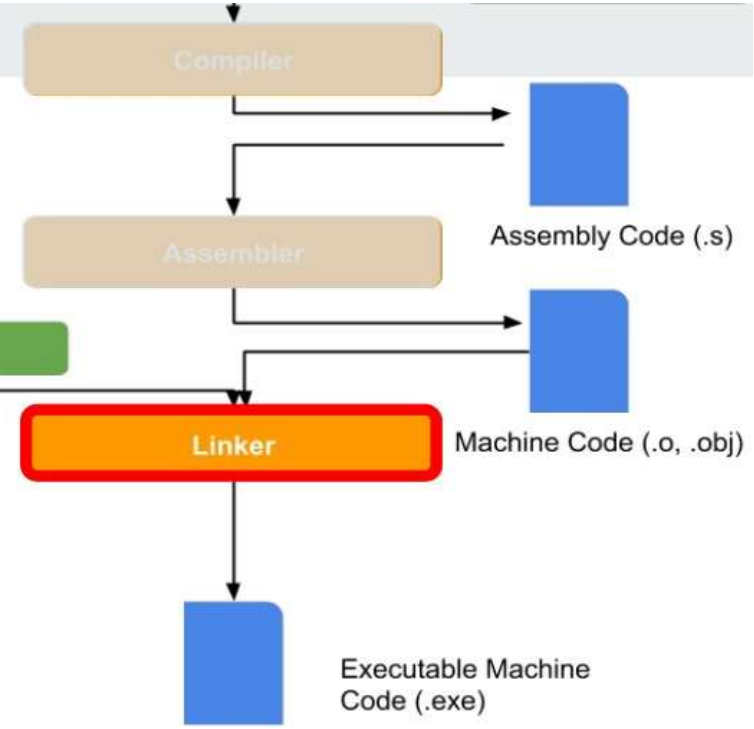


Figure 11

Cortex-M4 Memory Layout

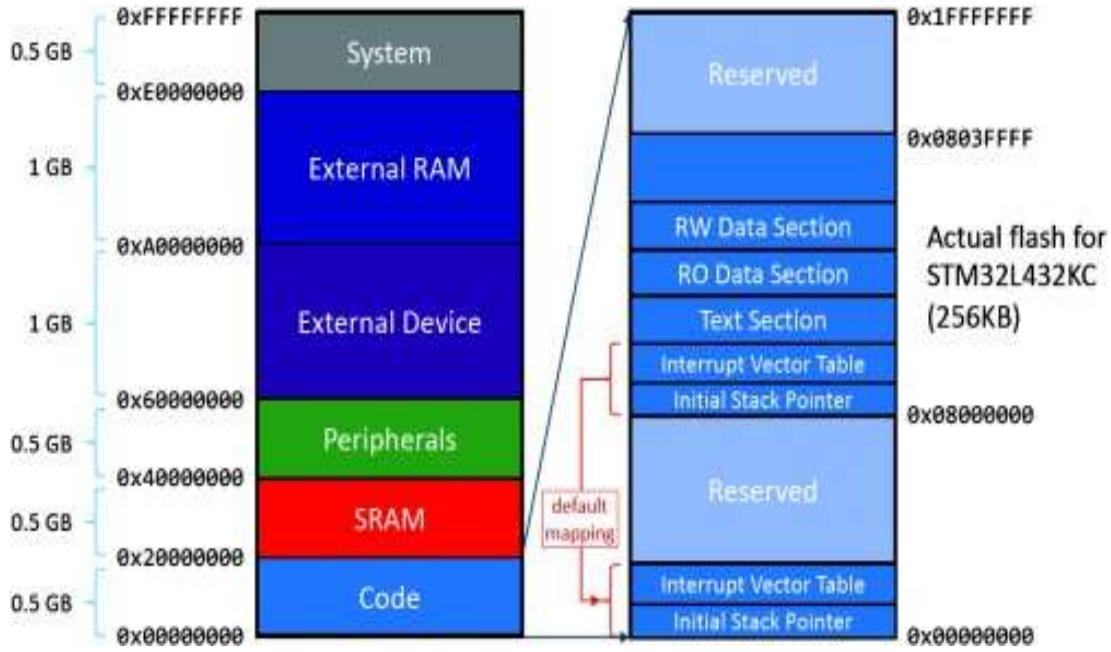


Figure 12

Useful website

For step by step

- installation: Setup Keil

  MDK:

- Creating first project with keil uvision 5 ARM:

Assembly Language Syntax

label
opcode operand1, operand2, ... ;
Comment

1. label:
   - label is an optional first field of an assembly statement.
   - labels are alphanumeric names used to define the starting location of a block of statements.
   - When creating the executable file the assembler will replace the label with the assigned value.
2. Opcode (Mnemonics) :
   - Opcode is the second field in assembly language instruction.
   - Assembly language consists of mnemonics, each corresponding to a machine instruction.
   - Assembler must translate each mnemonic opcode into their binary equivalent.
3. Operands:
   - Next to the opcode is the operand field which might contain different number of operands.
   - Normally, the first operand is the destination of the operation.
4. Comments:
   - Comments are messages intended only for human consumption.

A Sample ARM Assembly Program

```
        AREA test, CODE, READONLY
        ENTRY ; starting point of the code execution
        EXPORT main ; the declaration of identifier main
main ; address of the main function
     ; User code starts from the next line
        MOV r0, #4 ; store some arbitrary numbers
        MOV r1, #5
        ADD r2, r0, r1 ;  add the values in r0 and r1 and store the  result in r2
STOP  B Stop ; Endless loop
        END ; End of the program, matched with ENTRY keyword
```

Figure 13

A Sample ARM Assembly Program

- ; indicates user- supplied comment.
- AREA test, CODE, READONLY is an assembler directive and is required to setup the program.
- AREA refers to the segment code, test is the name I have
- defined, CODE means executable code rather that data, and
- READONLY indicates that it cannot be modified at runtime.
- Anything used in column1 is a label that is used to label that line.
- Stop B Stop means “ Branch to line labeled Stop” , used to create an infinite loop. This is a way to end the program.
- Last line END tells the assembler that there is no more code to execute.

Directives

- Assembler Directives:
  - Keil has an ARM assembler which can compile and build ARM assembly language programs.
  - To drive the assembly and linking process, we need to use directives, which are interpreted by the assembler.
  - Assembler directives are commands to the assembler that direct the assembly process.
  - They are executed by the assembler at assembly time not by the processor at run time.
  - Machine code is not generated for assembler directives as they are not directly translated to machine language.

Directives

- Area Directive:
  - AREA directive allows the programmer to specify the memory location to store code and data.
  - A name must be specified for an area directive.
- ENTRY and END Directives
  - The first instruction to be executed within an application is marked by the ENTR Y directive.
  - Entry point must be specified for every assembly language
  - program. This directive causes the assembler to stop processing the current source file.
  - Every assembly language source module must therefore finish with this directive.

Directives

- EXPORT Directive
  - A project may contain multiple source files. You may need to use a symbol in a source file that is defined in another source file.
  - In order for a symbol to be found by a different program file, we need to declare that symbol name as a global variable.
  - The EXPORT directive declares a symbol that can be used in different program files.
- The EQUATE Directive
  - The EQUATE directive allows the programmer to equate names with addresses or data.
  - This pseudo-operation is almost always given the mnemonic
  - EQU. The names may refer to device addresses, numeric data, starting addresses, fixed addresses, etc.
- READONLY as the name indicates protects this area from being overwritten by the program code.

Some Basic Instruction

Data Processing Instructions

- Arithmetic operations: – ADD, SUB, MUL
- Bit-wise logical operations: – AND, EOR, ORR,
- BIC Register movement operations: – MOV
- Comparison operations: – TST, TEQ, CMP,
- CMN LDR : Load Word from memory to register
- STR: Store Word from register to memory

Debug Scenario of the Sample Program

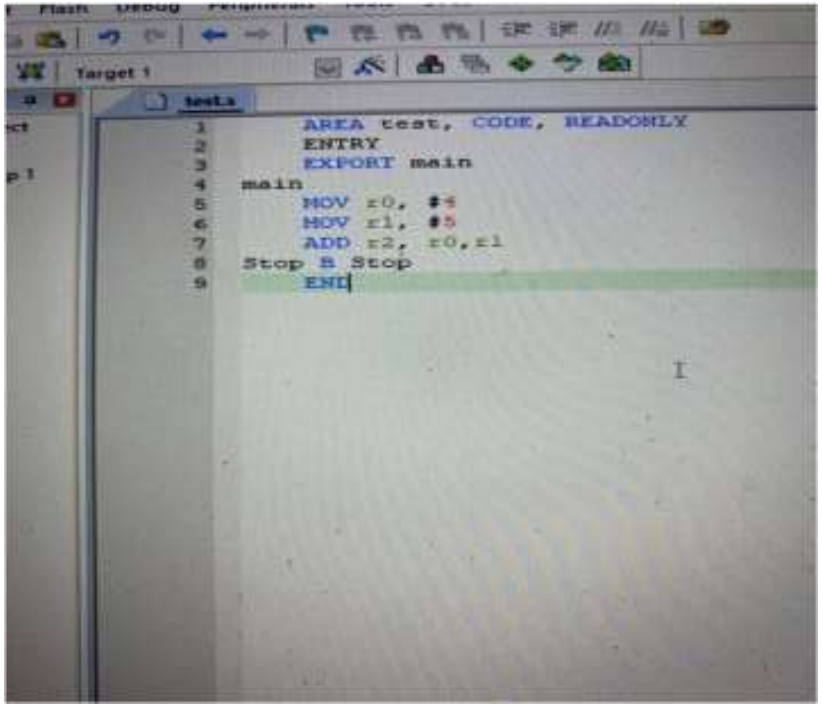


Figure 14

Debug Scenario of the Sample Program

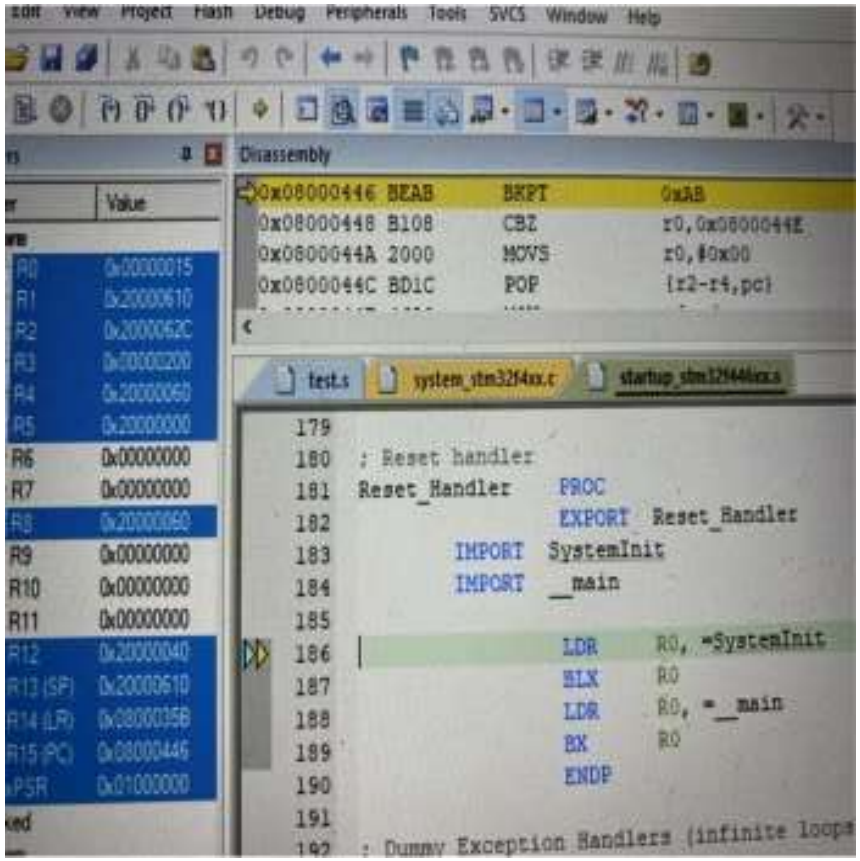


Figure 15

Debug Scenario of the Sample Program

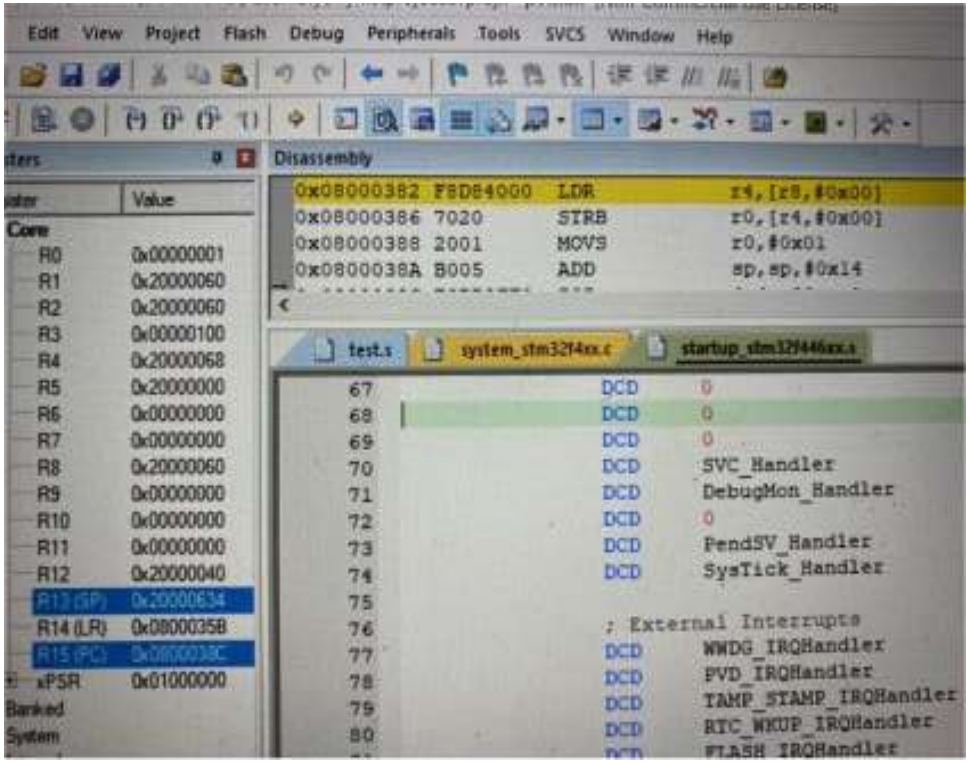


Figure 16

Debug Scenario of the Sample Program

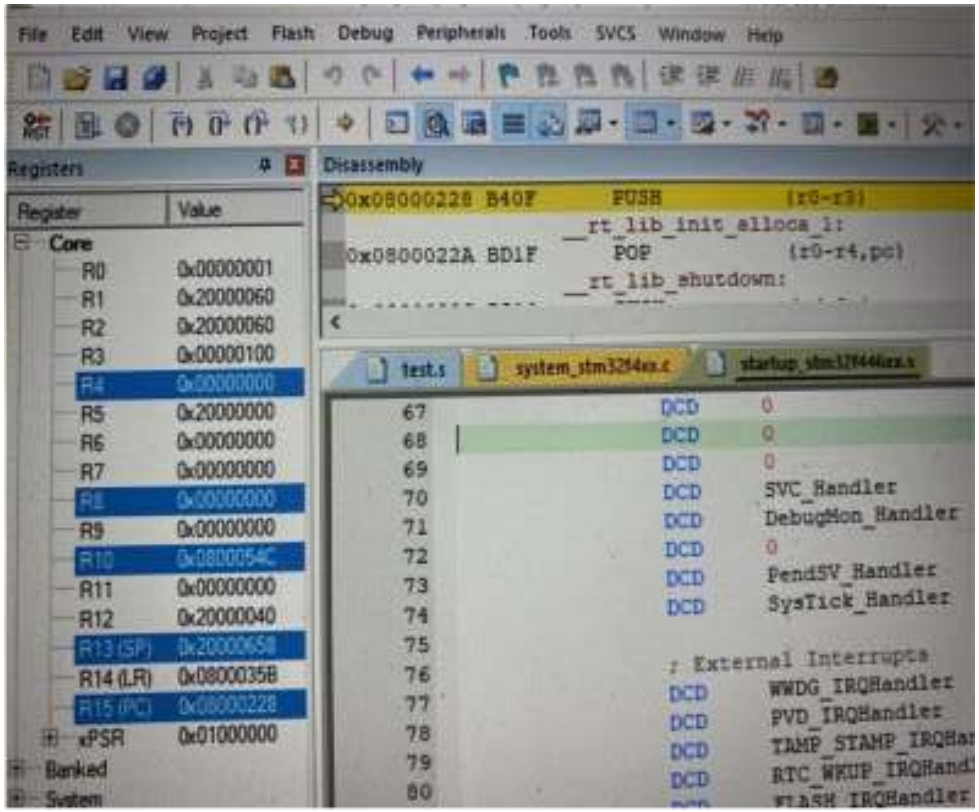


Figure 17

Debug Scenario of the Sample Program

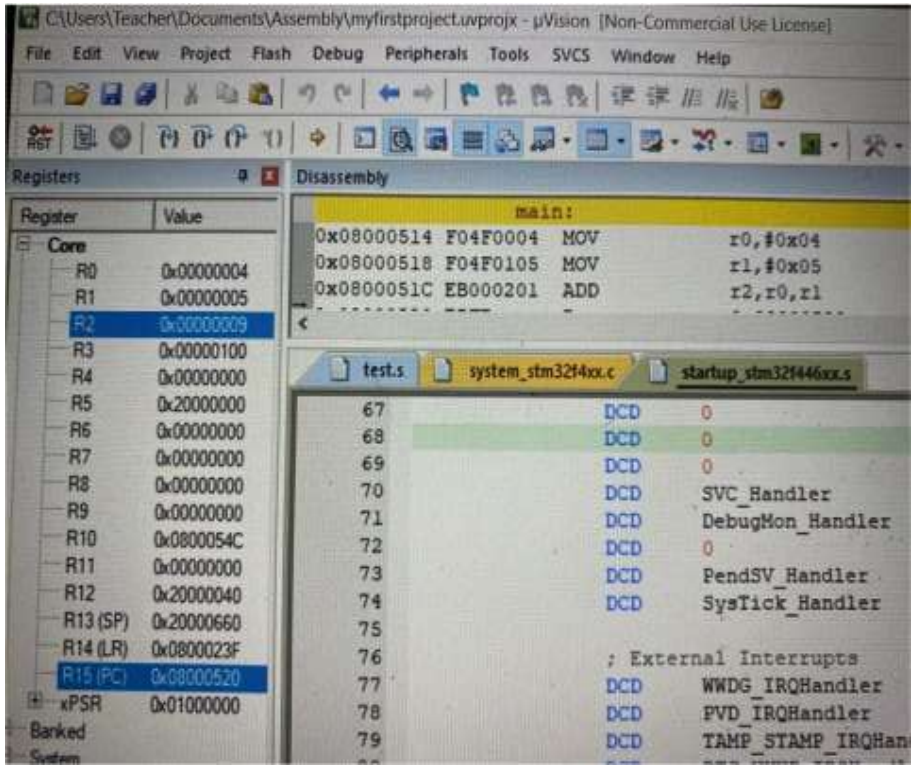


Figure 18