实验一　ARM 汇编程序设计

姓名：黄梓豪　　学号：2023310103008

本文使用 LaTeX 构建。

1　实验目的

1. 学会使用 Keil 开发工具对系统进行调试。
2. 结合 Keil 开发工具，理解程序的编译、执行过程。
3. 熟悉 Thumb 汇编指令，并且使用汇编指令编程。
4. 了解 C 程序调用汇编和 C 程序内嵌汇编代码的过程。

2　实验原理

2.1　Cortex-M0 调试环境

本实验使用 Keil MDK 集成开发环境进行软件开发与调试。Cortex-M0 处理器内置 CoreSight 调试技术，支持通过仿真器（如 CMSIS-DAP 或 J-Link）进行硬件调试。调试环境的核心功能包括：

- **寄存器观察：**实时监控通用寄存器（R0-R12）、栈指针（SP）、链接寄存器（LR）和程序计数器（PC）的状态变化。
- **内存映射（Memory Map）：**利用 ARM 的统一编址特性，通过 Memory 窗口直接查看代码段、数据段及栈空间的具体数值。
- **执行控制：**支持设置硬件断点、单步运行（Step）和全速运行，便于追踪程序流程。

2.2 指令工作过程

Cortex-M0 处理器基于 ARMv6-M 架构，主要执行 16 位的 Thumb 指令。指令执行遵循 **三级流水线**机制：

1. **取指 (Fetch)**：从存储器中读取指令代码。
2. **译码 (Decode)**：解析指令含义及操作数。
3. **执行 (Execute)**：执行 ALU 运算、数据传输或跳转操作。

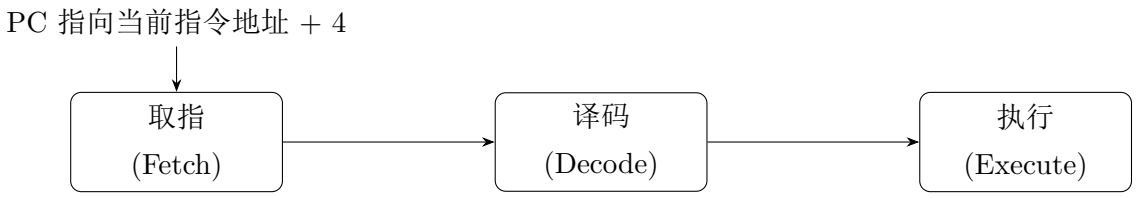


图 1: Cortex-M0 指令三级流水线示意图

由于流水线的存在，PC (R15) 的值通常等于当前执行指令地址加 4（两条指令的偏移）。

2.3 函数调用的基本原理

ARM 汇编程序与 C 语言程序的相互调用遵循 **ATPCS (ARM-Thumb Procedure Call Standard)** 标准：

- **参数传递**：函数的具体参数依次通过寄存器 **R0-R3** 传递；若参数超过 4 个，剩余参数通过栈传递。
- **返回值**：函数的返回值通常保存在 **R0** 寄存器中。
- **栈 (Stack)**：Cortex-M0 使用**满递减 (Full Descending)** 栈。栈指针 SP (R13) 指向当前栈顶数据，入栈 (PUSH) 时地址减小，出栈 (POP) 时地址增加。栈用于保存局部变量及函数调用时的现场（Context）。
- **返回地址**：使用 `BL` 指令调用函数时，下一条指令的地址会自动保存到链接寄存器 **LR (R14)** 中。函数返回时，通常将 LR 的值加载回 PC，或通过 `POP {PC}` 实现返回。
- **寄存器保护**：被调用函数需保证 **R4-R11** 的值在调用前后保持不变（Callee-saved），若需使用这些寄存器，必须先压栈保护。

3 实验任务

1. **汇编编程**：计算 $a = b \times c + d$。
   - a 的值保存寄存器序号为：选课号对 7 取余的寄存器中。
   - b 的值为：学号中的班级号。
   - c 的值为：学号的后 2 位的值。
   - d 的值为：选课号。
2. **C 程序调用汇编程序**：理解【例 4.62】中 `strcopy` 函数的执行过程，将复制后的字符串变为自己的学号。
3. **C 语言中内嵌汇编程序**：理解【例 4.65】的代码执行过程，将函数 `add(int i, int j)` 修改成 $i \times j + k$ 三个数运算的函数，并验证修改后的结果。

4 实验过程

提示：此处要求对程序的工作原理进行分析，以及解释为什么这么修改示例程序，并且通过 Keil 调试模式的截图证明结果的正确性。

4.1 任务一：汇编编程计算

本任务要求在 Cortex-M0 平台上编写一个简单的汇编程序，实现给定算术表达式的计算，并通过 Keil 调试器观察寄存器中结果的变化，加深对 ARM 指令执行过程和调用约定的理解。

本实验工程包含**提供的启动文件** `startup_CMSDK_CM0.s` 与**自编汇编源文件** `compute.s`。

启动文件主要负责系统复位后的环境初始化与中断向量表定义。具体流程如下：首先在 `STACK` 段中分配 1 KB 的栈空间并导出栈顶符号 `__initial_sp`；其次在 `RESET` 段构建向量表 `__Vectors`，将栈顶地址与复位中断服务程序 `Reset_Handler` 填入表项前两项。

在代码段实现的 `Reset_Handler` 中，程序通过 `IMPORT main` 声明外部符号，并执行 `LDR R0, =main` 与 `BX R0` 指令，将控制权直接转移至用户编写的 `main` 函数。值得注意的是，该启动文件直接跳转至用户入口 `main` 而非标准 C 库入口 `__main`，这意味着程序跳过了 C 运行库的初始化过程（如全局变量初始化）；但为了满足链接器对符号的静态依赖，我在自编的 `compute.s` 中额外定义了桩函数 `__user_initial_stackheap`，从而保证了工程的顺利编译与链接。

`compute.s` 中给出了任务一的具体计算逻辑。程序首先定义代码段并导出符号：

```
AREA    MyTask1, CODE, READONLY
EXPORT  main
EXPORT  __user_initial_stackheap
```

其中 main 为程序入口，__user_initial_stackheap 用于满足运行库对堆初始化符号的依赖（见下文）。在 main 中，按照“先装载操作数，再执行乘加运算”的思路，依次完成如下操作：

1. 使用 LDR 指令将常数装载到通用寄存器：R0, = 2023310103，R1, = 8，R3, = 45；
2. 执行 MULS R0, R1, R0 完成乘法运算，使 R0 = R1 * R0，即 R0 = 2023310103 * 8；
3. 执行 ADDS R0, R0, R3 完成加法运算，使 R0 = R0 + R3，最终结果为

$$R0 = 2023310103 \times 8 + 45;$$

4. 使用 MOV R3, R0 将结果再备份一份到 R3，便于在调试时交叉检查；
5. 最后跳转到标号 Stop，并通过 B Stop 形成死循环，使程序停在固定位置，便于在调试器中暂停和观察寄存器状态。

为解决链接阶段运行库对 __user_initial_stackheap 符号的依赖，我在文件末尾添加了一个最小实现：

```
__user_initial_stackheap
    BX      LR
```

该函数本身不做任何实际堆初始化操作，但可以满足运行库对符号的需求，不会影响程序行为，因为本实验并不使用 malloc 等需要堆空间的函数。

在 Keil 调试器中，我将程序运行至 Stop 死循环处，并在寄存器窗口中观察结果。此时寄存器 R0 的值为 0xC4CA18E5，换算为十进制为

$$0xC4CA18E5 = 3\,301\,578\,981,$$

而算术表达式的数学结果为

$$2023310103 \times 8 + 45 = 16\,186\,480\,869.$$

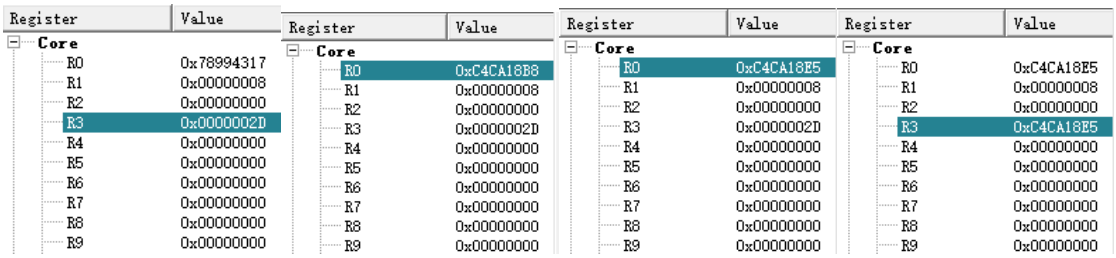
通用寄存器宽度为 32 位，按无符号数解释时的范围是 $0 \sim 2^{32} - 1$，寄存器中实际保留的是上述结果对 2^{32} 取模的值：

$$16\,186\,480\,869 \bmod 2^{32} = 3\,301\,578\,981,$$

与寄存器中的数值完全一致；同时 R3 中也保存了相同的结果。通过单步执行 MULS 和 ADDS 指令，可以直观地观察到 R0 中数值的逐步变化。以上调试结果表明，任务一的汇编程序正确实现了题目要求的算术运算，程序设计和运行过程均符合预期。

Listing 1: 任务一汇编程序代码

```
1     ;compute.s
2     AREA     MyTask1, CODE, READONLY
3
4     EXPORT   main
5     EXPORT   __user_initial_stackheap
6
7  main
8     LDR      R0, =2023310103
9     LDR      R1, =8
10    LDR      R3, =45
11
12    MULS     R0, R1, R0
13
14    ADDS     R0, R0, R3
15
16    MOV      R3, R0
17
18 Stop
19    B        Stop
20
21    ALIGN
22
23 __user_initial_stackheap
24    BX       LR
25
26    END
```



(a) (b) (c) (d)

图 2: 实验结果图示

如图 2a 所示，b, c, d 写入寄存器 R_0, R_1, R_3；图 2b 所示，执行乘法指令后，R_0 中存储 $a = b \times c$ 的结果；图 2c 所示，执行加法指令后，R_0 中存储 $a = b \times c + d$ 的结果；图 2d 所示，寄存器 R_0, R_3 中均存储 $a = b \times c + d$。

考虑到这里由于班级号太大导致溢出，可以考虑适当缩小之，比如写成 3，这样计算结果就不会溢出了。这里不做展示。

4.2 任务二：C 程序调用汇编 (strcopy)

4.2.1 任务目标与原理分析

本任务要求实现 C 语言调用汇编程序完成字符串复制（目标是将字符串修改为个人学号），并验证结果的正确性。

- **程序工作原理：**程序执行始于 *main*() 函数。在 *main*() 函数中，通过外部声明 `extern void strcopy(char *d, const char *s);` 调用汇编函数 `strcopy`。
- **传参机制 (ATPCS - ARM-Thumb Procedure Call Standard)：**根据 *ATPCS* 规则，C 语言函数调用时，前四个 32 位参数依次通过寄存器 *R*0 至 *R*3 传递。因此：
  - 目的字符串地址（`dststr`）作为第一个参数，存放在寄存器 *R*0 中。
  - 源字符串地址（`srcstr`）作为第二个参数，存放在寄存器 *R*1 中。
- **汇编实现：**汇编函数 `strcopy` 利用寄存器间接寻址，循环地将 *R*1（源地址）指向的字符加载到 *R*2，然后将 *R*2 存储到 *R*0（目的地址）。通过判断字符是否为 0（空字符 \0）来确定字符串的结束。
- **返回机制：**子程序执行完复制操作后，使用 *BX lr* (Branch and Exchange to Link Register) 指令返回到调用者（*main* 函数）。

4.2.2 示例程序代码

本任务主要使用了 `main.c` 和 `strcopy.s` 两个文件。

Listing 2: 汇编字符串复制代码

```
1             ;strcopy.s
2             AREA SCopy, CODE, READONLY
3             EXPORT  strcopy
4 strcopy
5             LDRB      r2,      [r1]     ; R1 (源地址) 读取字符到 R2
6             ADDS      r1,      #1       ; 源地址指针递增
7             STRB      r2,      [r0]     ; R2 字符存储到 R0 (目的地址)
8             ADDS      r0,      #1       ; 目的地址指针递增
9             CMP       r2,      #0       ; 判断是否为 NULL 结束符
10            BNE       strcopy           ; 非 NULL 则继续循环
11            BX        lr                ; 汇编子程序返回
12            END
```

Listing 3: C 语言调用汇编函数代码

```c
//main.c
#include <string.h>
#include <stdint.h>
#include <stdio.h>

extern void strcopy(char *d, const char *s);

int main()
{
    char srcstr[] = "2023310103008";
    char dststr[] = "Second string - destination";
    strcopy(dststr,srcstr);
    return 0;
}
```

4.2.3 实验调试过程与结果验证

在 Keil 调试器中，我们通过观察寄存器和内存窗口来验证程序的执行流程和结果。

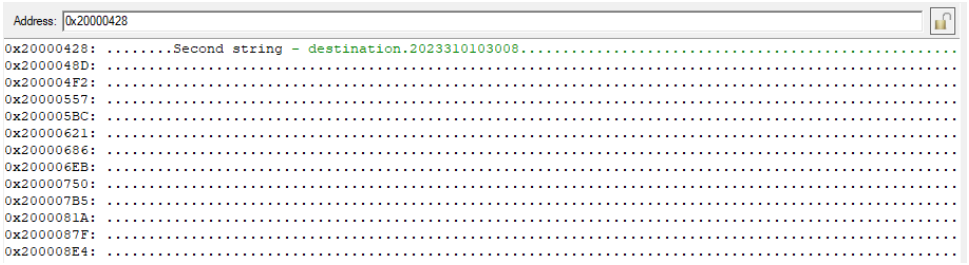


图 3: 初始状态

如图 3 所示，内存显示 *Ascii* 码“Second string - destination”存储在目的字符串地址处。此时 $RO = 0x20000430, R1 = 0x2000044C$

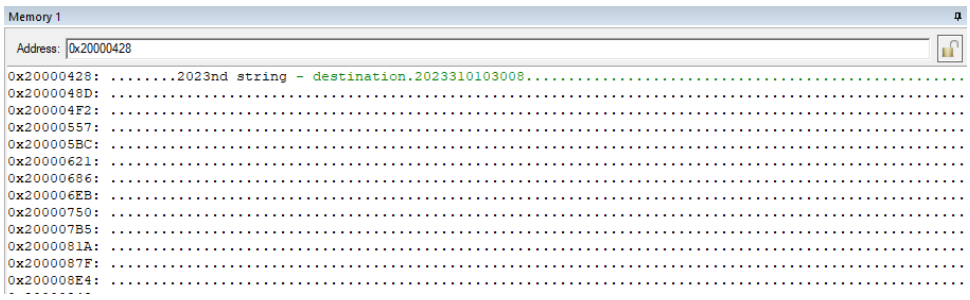


图 4: 开始复制

如图 4 所示，开始复制后，寄存器 $R0 = 0x20000433, R1 = 0x20000450$，$R0$ 指向正在粘贴的位置，$R1$ 指向将要复制的位置

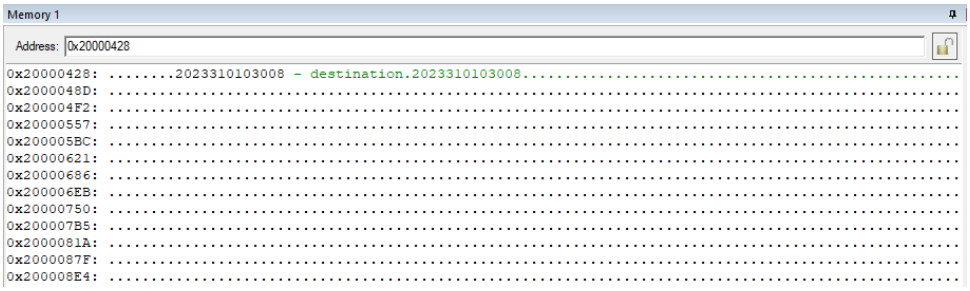


图 5: 复制完成

如图 5 所示，复制完成后，内存显示目的字符串已被修改为“2023310103008”，验证了字符串复制的正确性。此时，由于检测到字符串结束符，标志位 Z 被置为 1，导致条件跳转指令（BNE）失效，从而跳出循环，标志着复制操作的结束。

4.3 任务三：C 语言内嵌汇编

本任务要求在 C 语言中插入少量汇编代码，实现简单算术运算，并通过调试器观察寄存器与变量之间的对应关系，从而理解参数传递约定和内联汇编的用法。实验中使用的 C 程序如下所示：

```c
//main.c
#include <stdio.h>

int math_calc(int i, int j, int k)
{
    int res;

    __asm
    {
        MOVS res, i
        MULS res, j
        ADDS res, res, k
    }

    return res;
}

int main()
{
    int result;
    result = math_calc(2, 3, 4);

    return 0;
}
```

工作原理分析　在 `math_calc` 函数中，首先按照 C 语言的形式定义了三个整型参数 `i`、`j`、`k` 和一个局部变量 `res`。函数体内部使用 `__asm { ... }` 块插入 ARM 汇编指令。编译器会在生成目标代码时，将 `i`、`j`、`k`、`res` 映射到对应的寄存器或栈上的临时单元，在内联汇编块中，标识符 `i`、`j`、`k`、`res` 就可以像普通寄存器一样被汇编指令访问。

内联汇编块完成的运算过程为：

1. `MOVS res, i`：将形参 `i` 的值复制到局部变量 `res` 中，作为后续计算的初始值；
2. `MULS res, j`：执行乘法运算 `res = res * j`，此时 `res = i * j`；
3. `ADDS res, res, k`：执行加法运算 `res = res + k`，最终得到表达式

$$res = i \times j + k.$$

内联汇编结束后，`res` 的值通过 `return res;` 作为函数返回值返回。根据 ARM 的 ATPCS 约定，函数返回值最终会存放在寄存器 `R0` 中，因此在调试器中观察到 `R0` 的值即为计算结果。

在 `main` 函数中调用 `math_calc(2, 3, 4)`，期望结果为

$$2 \times 3 + 4 = 10.$$

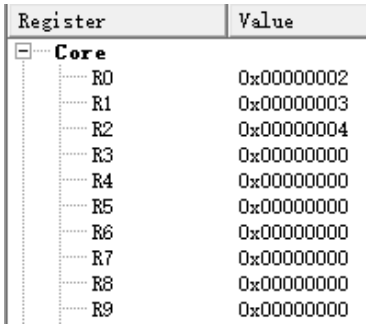


图 6: 初始赋值

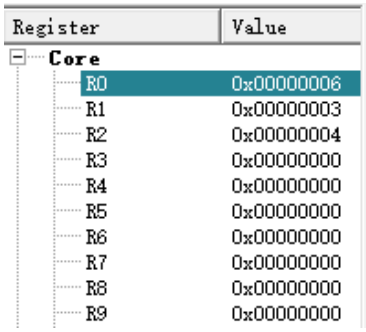


图 7: $i \times j$

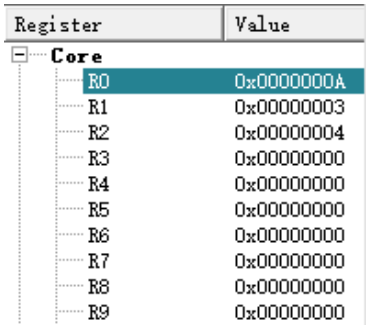


图 8: 得出结果

在 Keil 调试模式下，单步执行到 `return 0;` 前暂停，此时在寄存器窗口可以看到 `R0 = 0x0000000A`，与十进制 10 一致；在 Watch 窗口中，`result` 的值也为 10。调试结果验证了内联汇编块正确实现了给定的算术表达式，说明 C 语言与内嵌汇编之间的数据传递和寄存器使用均符合预期。

通过本实验，可以直观地理解：

- C 形参和局部变量如何被映射到寄存器 / 栈空间，并在内联汇编中被访问；
- ARM 汇编中 `MOVS`、`MULS`、`ADDS` 等指令的具体作用；
- 函数返回值通过寄存器 `R0` 传递这一 ATPCS 规则在实际代码中的体现。

这为后续在更复杂的 C 工程中插入优化用汇编代码打下了基础。

5 实验总结

本次实验通过“汇编计算、C 调用汇编、C 内嵌汇编”三个任务，理解了 Cortex-M0 启动流程、寄存器与栈的用法以及 ATPCS 调用约定，学会在 Keil 中单步调试和查看寄存器、内存变化，并在多次解决链接错误和库配置问题的过程中体会到环境配置与问题定位的重要性。例如在实验中先后遇到 `__initial_sp` 和 `__user_initial_stackheap` 未定义、启动文件重复、编译器 V5/V6 选择不当、是否启用 MicroLIB 等问题，最终通过查阅文档和分析报错信息逐一解决。总体来说，本次实验既加深了我对 Cortex-M0 架构和调用约定的理解，也让我实际体验了一遍“环境搭建—出错—定位—修复”的完整调试流程，为后续进一步学习嵌入式系统开发打下了基础。

思考题

1. **Cortex-M0 执行完一条非跳转指令之后，PC 寄存器的值怎样变化？**

   Cortex-M0 使用顺序执行和三级流水线结构。对一条普通的非跳转 Thumb 指令来说，硬件会在执行结束后自动把程序计数器 PC 增加一个指令长度，使其指向“下一条指令”的地址：16 位指令时 PC 增加 2 字节，32 位指令时增加 4 字节。因此，从程序员角度看，如果按地址顺序写代码，执行完一条非跳转指令后，PC 会顺序指向下一条指令的首地址。

2. **为什么 Cortex-M0 使用的寄存器都是低寄存器（即 R0–R7）？**

   Cortex-M0 只实现了 Thumb-1 子集，大多数数据处理指令采用 16 位定长编码，指令中用于指出寄存器的字段只有 3 位，因此一次最多只能直接编码 8 个寄存器号。为了提高代码密度、简化硬件解码逻辑，Thumb-1 规定普通算术、逻辑指令优先使用低寄存器 $R0 \sim R7$，而高寄存器（$R8 \sim R12$、SP、LR、PC）通常通过专用指令或特定用途访问。这样既保持了较小的内核面积和功耗，又能兼顾较好的代码密度，非常适合 Cortex-M0 这类低功耗嵌入式应用。

3. **结合 Keil 的调试过程，简要说明 Cortex-M0 的启动过程？**

   在上电或复位后，Cortex-M0 首先从地址 0x00000000 处读取初始栈顶值装入 SP，再从地址 0x00000004 处读取复位向量，即 `Reset_Handler` 的入口地址装入 PC，开始执行启动代码。在 Keil 调试时可以看到，程序一开始就跳入启动文件中的 `Reset_Handler`：这里通常完成堆栈初始化、拷贝初始化数据段、清零 BSS 段，以及（必要时）调用系统初始化函数 `SystemInit` 等。随后启动代码再跳转到 C 运行库入口或用户的 `main` 函数（本实验中是直接 `LDR r0, =main; BX r0`），此后程序进入用户编写的 C/汇编主程序逻辑。通过单步调试可以清楚地看到 PC 从向量表指向 `Reset_Handler`，再从 `Reset_Handler` 跳转到 `main` 的完整启动过程。