Experiment 3

Assembly Language Programming

Every computer, no matter how simple or complex, has a microprocessor that manages the computer's arithmetical, logical and control activities. A computer program is a collection of numbers stored in memory in the form of ones and zeros. CPU reads these numbers one at a time, decodes them and perform the required action. We term these numbers as machine language.

Although machine language instructions make perfect sense to computer but humans cannot comprehend them. A long time ago, someone came up with the idea that computer programs could be written using words instead of numbers and a new language of mnemonics was developed and named as assembly language. An assembly language is a low-level programming language and there is a very strong (generally one-to-one) correspondence between the language and the architecture's machine code instructions.

We know that computer cannot interpret assembly language instructions. A program should be developed that performs the task of translating the assembly language instructions to machine language. A computer program that translates the source code written with mnemonics into the executable machine language instructions is called an assembler. The input of an assembler is a source code and its output is an executable object code. Assembly language programs are not portable because assembly language instructions are specific to a particular computer architecture. Similarly, a different assembler is required to make an object code for each platform.

The ARM Architecture

The ARM is a *Reduced Instruction Set Computer*(RISC) with a relatively simple implementation of a load/store architecture, i.e. an architecture where main memory (RAM) access is performed through dedicated load and store instructions, while all computation (sums, products, logical operations, etc.) is performed on values held in registers. ARM supports a small number of addressing modes with all load/store addresses being determined from registers and instruction fields only.

The ARM architecture provides 16 registers, numbered from R0 to R15, of which the last three have special hardware significance.

R13, also known as SP, is the stack pointer, that is it holds the address of the top element of the program stack. It is used automatically in the PUSH and POP instructions to manage storage

```
label
        opcode    operand1,  operand2, …   ; Comment
```

and recovery of registers in the stack.

R14, also known as LR is the link register, and holds the return address, that is the address of the first instruction to be executed after returning from the current function.

R15, also known as PC is the program counter, and holds the address of the next instruction.

Assembly Language Syntax

ARM assembler commonly uses following instruction format:

Normally, the first operand is the destination of the operation. The number of operands in an instruction depends on the type of instruction, and the syntax format of the operand can also be different.

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. Labels can be subsequently used in our program as an operand of some other instruction. When creating the executable file the assembler will replace the label with the assigned value. Labels must be unique in the executable file because an identical label encountered by the Assembler will generate an error.

ARM assembler has reserved first character of a line for the label field and it should be left blank for the instructions with no labels. In some compilers, labels can be optionally ended with colon(:) but it is not accepted by the ARM assembler as an end of the label field.

Defining appropriate labels makes your program look more legible. Program location can be easily found and remembered using labels. It is easier to use certain functionality of your program in an entirely different code .i.e., your code can be used as a library program. You do not have to figure out memory addresses because it is a tedious task especially when the instruction size in your microcontroller is not fixed.

Opcode (Mnemonics)

Opcode is the second field in assembly language instruction. Assembly language consists of mnemonics, each corresponding to a machine instruction. Using a mnemonic you can decide

what operation you want to perform on the operands. Assembler must translate each mnemonic opcode into their binary equivalent.

Operands

Next to the opcode is the operand field which might contain different number of operands. Some of the instructions in Cortex-M3 will have no operand while other might have as many as four operands. The number of operands in an instruction depends on the type of instruction, and the syntax format of the operand can also be different. Normally, the first operand is the destination of the operation.

Comments

Comments are messages intended only for human consumption. They have no effect on the translation process and indeed are not acted on by the ARM Assembler. The comment field of an assembly language instruction is also optional. A semicolon signies that the rest of the line is a comment and is to be ignored by the assembler. If the semicolon is the rst non-blank character on the line, the entire line is ignored. If the semicolon follows the operands of an instruction, then only the comment is ignored by the assembler. The purpose of comments is to make the program more comprehensible to the human reader. If you want to modify the program in response to a product update and you haven't used that code before then comments go a long way to helping comprehension.

A First Assembly Program

In this section we learn to write a simple assembly language program with and without the startup file. This program can be used as a template for writing any assembly language program. An assembler is requires to compile an assembly language program. Keil has an ARM assembler which can compile and build ARM assembly language programs.

Assembly language is only a set of instructions for specific hardware resources. To drive the assembly and linking process, we need to us *directives*, which are interpreted by the assembler. Some commonly used directives are explained in section —-

Template for ARM Assembly Program

```
; First ARM Assembly language program
  PRESERVE8
  THUMB   ; marks the THUMB mode of operation
  AREA    |.text|, CODE, READONLY, ALIGN=2
  ENTRY   ; starting point of the code execution
```

```
EXPORT __main   ; declaration of identifier main
__main   ; address of the main function
; User Code Starts from the next line
END   ; End of the program, matched with ENTRY keyword
```

Template for ARM Assembly Program

Example 3.1: Assembly Program with Startup File

```
; This is the first ARM Assembly language program. Describe the
; functionality of the program at the top. You can also include
; the info regarding the author and the date

;;; Directives
    PRESERVE8
    THUMB    ; Marks the THUMB mode of operation

;;;;;;;;;; The user code (program) is placed in CODE AREA

    AREA  |.text|, CODE, READONLY, ALIGN=2
    ENTRY    ; Marks the starting point of code execution
    EXPORT __main

__main
; User Code Starts from the next line

    MOV   R5, #0x1234    ; Store some arbitrary numbers
    MOV   R3, #0x1234
    ADD   R6, R5, R3     ; Add the values in R5 and R3 and store the
                         ; result in R6
STOP
    B     STOP           ; Endless loop
    END                  ; End of the program, matched with
                         ; ENTRY keyword
```

Example 3.2: Assembly Language Program without Startup File

```
;;; Directives
    PRESERVE8
    THUMB    ; Marks the THUMB mode of operation

Stack   EQU     0x00000100  ; Define stack size to be 256 byes
  ; Allocate space for the stack.
    AREA  STACK, NOINIT, READWRITE, ALIGN=3
```

```
StackMem  SPACE    Stack

  ; Initialize the two entries of vector table.
    AREA  RESET, DATA, READONLY
    EXPORT  __Vectors

__Vectors
    DCD  StackMem + Stack    ; stack pointer value when stack is empty
    DCD  Reset_Handler       ; reset vector

    ALIGN

;;;;;;;;;; The user code (program) is placed in CODE AREA

    AREA     |.text|, CODE, READONLY, ALIGN=2
    ENTRY     ; Marks the starting point of the code execution
    EXPORT Reset_Handler

Reset_Handler
; User Code Starts from the next line

    MOV    R5, #0x1234     ; Move some arbitrary number in R5
    MOV    R3, #0x1234
    ADD    R6, R5, R3      ; Add the values in R5 and R3 and store the
                           ; result in R6
STOP
    B      STOP            ; Endless loop
    END                    ; End of the program, matched with
                           ; ENTRY keyword
```

Assembler Directives

Every program to be executed by a computer is a sequence of statements that can be read by the humans. An assembler must differentiate between the statements that will be converted into executable code and that instruct the assembler to perform some specific function.

Assembler directives are commands to the assembler that direct the assembly process. Assembler directives are also called pseudo opcodes or pseudo-operations. 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. Some tasks performed by these directives are:

1. Assign the program to certain areas in memory.
2. Define symbols.
3. Designate areas of memory for data storage.
4. Place tables or other fixed data in memory.
5. Allow references to other programs.

Attribute	Explanation
CODE	Contains machine instructions. READONLY is the default.
DATA	Contains data, not instructions. READWRITE is the default.
READONLY	Indicates that this area should not be written to.
READWRITE	Indicates that this area may be read from or written to.
NOINIT	Indicates that the data area is initialized to zero. It contains only reservation directives with no initialized values.

ARM assembler supports a large number of assembler directives but in this lab manual we will discuss only some important directives which are required for this lab.

AREA Directive

AREA directive allows the programmer to specify the memory location to store code and data. Depending on the memory configuration of your device, code and data can reside in different areas of memory. A name must be specified for an area directive. There are several optional comma delimited attributes that can be used with AREA directive. Some of them are discussed below.

Example 3.3

```
AREA    Example, CODE, READONLY    ; An example code section.
        ; user code
```

ENTRY and END Directives

The first instruction to be executed within an application is marked by the ENTRY directive. Entry point must be specified for every assembly language program. An application can contain only a single entry point and so in a multi-source module application, only a single module will contain an ENTRY directive.

This directive causes the assembler to stop processing the current source file. Every assembly language source module must therefore finish with this directive.

Example 3.4

```
        AREA    MyCode, CODE, READONLY
        ENTRY       ; Entry point for the application

Start
```

```
        ; user code

        END         ; Informs the assembler about the end of a source file
```

EXPORT and IMPORT Directives

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. GLOBAL is a synonym for EXPORT.

The IMPORT directive provides the assembler with a name that is not defined in the current assembly.

Example 3.5

```
AREA     Example, CODE, READONLY
         IMPORT   User_Code        ; Import the function name from
                                   ; other source file.
         EXPORT   DoAdd            ; Export the function name
                                   ; to be used by external
                                   ; modules.
DoAdd    ADD      R0,R0,R1
```

ARM, THUMB Directives

The ARM directive instructs the assembler to interpret subsequent instructions as 32-bit ARM instructions. If necessary, it also inserts up to three bytes of padding to align to the next word boundary. The ARM directive and the CODE32 directive are synonyms.

The THUMB directive instructs the assembler to interpret subsequent instructions as 16-bit Thumb instructions. If necessary, it also inserts a byte of padding to align to the next halfword boundary.

In files that contain a mixture of ARM and Thumb code Use THUMB when changing from ARM state to Thumb state. THUMB must precede any Thumb code. Use ARM when changing from Thumb state to ARM state. ARM must precede any ARM code.

Example 3.6

```
AREA    ChangeState, CODE, READONLY
        ARM
                              ; This section starts in ARM state
        LDR     r0,=start+1   ; Load the address and set the
                              ; least significant bit
        BX      r0            ; Branch and exchange instruction sets

                              ; Not necessarily in same section
        THUMB                 ; Following instructions are Thumb
start   MOV     r1,#10        ; Thumb instructions
```

ALIGN Directive

Use of ALIGN ensures that your code is correctly aligned. By default, the ALIGN directive aligns the current location within the code to a word (4-byte) boundary. ALIGN 2 can also be used to align on a halfword (2-byte) boundary in Thumb code. As a general rule it is safer to use ALIGN frequently through your code.

PRESERVE8 Directive

The PRESERVE8 directive specifies that the current file preserves 8-byte alignment of the stack. LDRD and STRD instructions (double-word transfers) only work correctly if the address they access is 8-byte aligned. If your code preserves 8-byte alignment of the stack, use PRESERVE8 to inform the linker. The linker ensures that any code that requires 8-byte alignment of the stack is only called, directly or indirectly, by code that preserves 8-byte alignment of the stack.

Data Reservation Directives (DCB, DCD, DCW)

ARM assembler supports different data definition directives to insert constants in assembly code. This directive allows the programmer to enter fixed data into the program memory and treats that data as a permanent part of the program. Different variants of these directives are:

1. DCB (Define Constant Byte) to define constants of byte size.
2. DCW (Define Constant Word) allocates one or more halfwords of memory, aligned on two-byte boundaries.
3. DCD (Define Constant Data) allocates one or more words of memory, aligned on four-byte boundaries.

Example 3.7

```
data    DCD     0, 0, 0         ; Defines 3 words initialized to zeros
```

SPACE Directive

The SPACE directive reserves a zeroed block of memory. ALIGN directive must be used to align any code following a SPACE directive.

Example 3.8

```
AREA    MyData, DATA, READWRITE
data1   SPACE   255         ; defines 255 bytes of zeroed store
```