Experiment 4

Overview of ARM Assembly Language

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. Computer programs can be written using mnemonics instead of numbers in 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.

Computer cannot interpret assembly language instructions. Assembler translates the source code written with mnemonics into the executable machine language instructions. The input of an assembler is an assembly 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 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:

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

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

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. 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.

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-M4 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 signifies that the rest of the line is a comment and is to be ignored by the assembler.

ARM Assembly Example Code

```
        PRESERVE8
        THUMB

        AREA  myDATA, DATA, READWRITE

Result  SPACE 8          ;reserves eight bytes for variable result

        AREA  |.text|, CODE, READONLY

        ENTRY
        EXPORT   __main

__main

        LDR  R0, =Value1         ; Load the starting address of
        ; first 64-bit number
        LDR  R1, =Value2         ; Load the starting address of
        ; second 64-bit number

        BL   SUM64

        LDR  R0, =Result
        STR  R6, [R0]            ; Store higher 32-bits to Result
        STR  R7, [R0, #4]        ; Store lower 32-bits to Result

        B    STOP

SUM64   PROC

        LDR  R2, [R0]           ; Load the value of higer 32-bits
        LDR  R3, [R0, #4]       ; Load the value of lower 32-bits


        LDR  R4, [R1]
        LDR  R5, [R1, #4]

        ADDS R7, R3, R5         ; Add with flags update
        MOV   R6, #0xD129
        ; Move the 16-bit hex value to the lower halfword of R6

        MOVT  R6, #0xF29A    ; R6 =
        ; Move the 16-bit hex value to the upper halfword of R6

        BX   LR
        ENDP
STOP

Value1  DCD  0x12A2E640, 0xF2100123
Value2  DCD  0x001019BF, 0x40023F51

        END
```

Assembler Directives

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.

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.

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.

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.

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.

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.

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.

SPACE Directive

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

ARM Assembly Instructions

ARM assembly instructions can be divided in three different sets.

1. **Data processing** instructions manipulate the data within the registers. These can be arithmetic (sum, subtraction, multiplication), logical (boolean operations), relational (comparison of two values) or move instructions.
2. **Memory access** instructions move data to and from the main memory.
3. **Branch** instructions change the control flow, by modifying the value of Program Counter (R15). They are needed to implement conditional statements, loops, and function calls.

Data processing Instructions

ARM data processing instructions enable the programmer to perform arithmetic and logical operations on data values in registers.If you use the S suffix on a data processing instruction, then it updates the flags in the CPSR. Move and logical operations update the carry flag C, negative flag N, and zero flag Z. The carry flag is set from the result of the barrel shift as the last bit shifted out. The N flag is set to bit 31 of the result. The Z flag is set if the result is zero.

Move Instruction

Inside the processor, the data resides in the registers. MOV is the basic instruction that moves the constant data in the register or move that data from one register to another. In the Thumb instruction set MOVT instruction moves 16-bit immediate value to top halfword and the bottom halfword remains unaltered.

Shift and Rotate instructions

Shift and Rotate instructions are used to change the position of bit values in a register.
Logical Shift Right (LSR) is similar to the unsigned divide by 2^n where n specifies the number of shifts.
Arithmetic Shift Right (ASR) is similar to the signed divide by 2^n where n specifies the number of shifts.
Logical Shift Left (LSL) works fine for both signed and unsigned numbers. This instruction is synonymous to multiply by 2^n where n specifies the number of shifts.
Rotate Right (ROR) do not discard any bits from the register. Instead, the bit values are removed from one end of the register and inserted into the other end.

Arithmetic/Logic Instructions

These instructions perform arithmetic or logical operations on upto two source operands and write a result to a destination register. Of the two source operands one is always a register and other can be an immediate value or a register value, optionally shifted e.g. ADD, SUB, AND, OR.

Memory Access Instructions

All data processing operations are executed in the registers i.e., data values needed for an operation must be moved into registers before using them. We need instructions that interact with memory to move data from memory to registers and vice versa.
LDR instruction can also be used to load any constant in the registers. Any 32-bit numeric constant can be constructed in a single instruction. It is used to generate constants that are out of range of the MOV and MVN instructions.

These instructions provide the most flexible way to transfer single data items between an ARM register and memory. The data item may be a byte, a 16-bit half-word, a 32-bit word or a 64-bit double word. An optional modifier should be added to access the appropriate data type. Following table shows the data types available and their ranges.

type	Data Type	Range
	32-bit word	0 to $2^{32}-1$ or -2^{31} to $2^{31}-1$
B	unsigned Byte	0 to $2^{8}-1$
SB	Signed Byte	-2^{7} to $2^{7}-1$
H	unsigned Halfword	0 to $2^{16}-1$
SH	Signed Halfword	-2^{15} to $2^{15}-1$
D	Double word	0 to $2^{64}-1$ (two registers used)

ARM Conditional Branch Instructions

ARM supports different branch instructions for conditional executions. Depending on the condition these instructions transfer the control from one part of the program to other. Unlike Branch-and-Link (BL) instruction they do not save contents of Program counter (PC) register to the Link Register (LR).

Branch	Interpretation	Flag	Normal Uses
B		any value	Always take this branch
BAL	Always	any value	Always take this branch
BEQ	Equal	Z=1	Comparison equal or result zero
BNE	Not Equal	Z=0	Comparison not equal or non-zero result
BPL	Plus	N=0	Result positive or zero
BMI	Minus	N=1	Result minus or negative
BCC	Carry Clear	C=0	Arithmetic operation didn't give carry out
BLO	Lower	C=0	Unsigned comparison gave lower
BCS	Carry Set	C=1	Arithmetic operation gave carry out
BHS	Higher or Same	C=1	Unsigned comparison gave higher or same
BVC	Overflow Clear	V=0	Signed integer operation; no overflow occured
BVS	Overflow Set	V=1	Signed integer operation; overflow occured
BGT	Greater Than	Z = 0 and N = V	Signed integer comparison gave greater than
BGE	Greater or Equal	Z = 0 and N = V	Signed integer comparison gave greater than or equal
BLT	Less Than	N != V	Signed integer comparison gave less than
BLE	Less or Equal	Z = 1 and N != V	Signed integer comparison gave less than or equal
BHI	Higher	C = 1 and Z = 0	Unsigned comparison gave higher
BLS	Lower or Same	C = 0 or Z = 1	Unsigned comparison gave lower or same

Further details of the instructions can be found in chapter 3: The Cortex-M4 Instruction Set of the Cortex-M4 Devices Generic User Guide (https://developer.arm.com/documentation/dui0553/b/?lang=en).