# ESET 349 – Microcontroller Architecture

## Introduction to ARM Assembly Language

Dr. Muhammad Faeyz Karim

# First Assembly Program (Shifting Data)

```
    AREA Prog1, CODE, READONLY
    ENTRY
    MOV   r0,  #0x11          ; load initial value
    MOV   r1,  r0, LSL #1     ; shift 1 bit left
    MOV   r2,  r1, LSL #1     ; shift 1 bit left
stop Bstop
    END
```

# Assembler Directives vs. Microprocessor Instructions

- Those Blue wordings are Assembler Directives which are instructions to the Assembler ( a piece of software). These instructions are not converted into machine code and will never be executed by the processor. We will cover more of it later.
- The Green wordings are ARM microprocessor instructions. They are converted to machine code during assembly and are executed by the microprocessor when the program is run.

# Assembler Directives

- AREA – for the assembler to create a block of code (in this case) in memory.
- CODE – The block created is a set of instructions (or program)
- READONLY – The block is read-only, which means it cannot be written into.
- AREA Prog1, CODE, READONLY means the assembler will create a block of code called Prog1 (just a name) which is readonly.
- ENTRY – it is the point where the program starts execution.
- END – End of program, the assembler will ignore all other instructions after the END statement.

# ARM Instructions

- MOV r0, #0x11
  - The # means that the constant is an immediate operand
  - This instruction MOVes 11 hexadecimal into register r0.
  - Now register r0 contains 0x11
  - Or binary 0000 0000 0000 0000 0000 0000 0001 0001
- MOV r1, r0, LSL #1
  - LSL stands for Logical Shift Left, the instruction will shift the contents of r0 1 bit to the left and place the result in register r1.
  - Now register r1 contains 0x22
  - Or binary 0000 0000 0000 0000 0000 0000 0010 0010
- What is the result in r2 after the last instruction ?

  MOV r2, r1, LSL #1

# Stopping the program

- stop B stop
- stop is the name of a label, you can call it any other name, it will still work.
- B is short for Branch
- The instruction actually tells the processor to branch to the branch instruction itself, which puts the code into an infinite loop.
- Very crude method but convenient as it allows us to terminate the simulation easily by choosing Start/Stop Debug Session from the Debug menu or clicking the Halt button in our Keil tools

# Factorial Program

- Factorial of n or n! = n(n-1)(n-2)…(1)
- This program will introduce the topics of
  - Conditional execution: The multiplication may or may not be performed, depending on the result of another instruction.
  - Setting flags: The "S" suffix on an instruction directs the processor to update the flags in the CPSR based on the result of the operation
  - Change-of-flow instructions: A branch will load a new address, called a branch target, into the Program Counter, and execution will resume from this new address.

```
        AREA Prog2,  CODE,  READONLY
        ENTRY
        MOV    r6, #10      ; load 10 into r6
        MOV    r4, r6       ;  copy n into a temp register
loop    SUBS   r4, r4, #1   ; decrement next multiplier
        MULNE  r7, r6, r4   ;perform multiply
        MOV        r6, r7
        BNE    loop    ;go again if not complete
stop    B      stop    ;stop program
        END
```

# Explanation Of The Program

- The instruction "**MOV r6, #10**" moves the decimal (default) value of 10 into r6. The program computes the factorial of 10 and store the result in r6.
- The next **MOV** instruction copies the contents of r6 into r4, which also serves as the multiplier which is reduced by one every iteration until it reaches one.
- The next instruction "**SUBS r4, r4, #1**", performs r4 = r4 – 1 operation. The "**S**" at the end of the **SUB** instruction means the condition code flags will be set at the end of the instruction. If the result is 0, the Z (Zero) flag will be set (=1).

# Explanation "Cont'd"

- The **MULNE** instruction multiplies r6 by r4 and put the result in r7, but only if (subtraction result in this case) not equal to zero. Note: when the subtraction result in zero, Z flag is set or Z flag =1, otherwise Z flag is 0 or not set).
- The third **MOV** instruction places the product into r6, which will become the final result eventually.
- The **BNE** works with a label that we placed above it called loop (can be replaced by any name, it is not a reserved word). When executed, the branch instruction changes the program counter to the address of its branch target but only if the result from the subtraction is non-zero (by testing the Z flag).

# Labels

- Labels are names chosen to represent an address somewhere in memory and they will be eventually be translated into a numeric value by the linker.
- A label name can only defined once in your code, multiple labels of the same name is not allowed.
- Label name cannot start with a number .e.g. 1app is not allowed.
- Labels must start on the left.

# Constant Values

- Constants can be Expressed as Numeric Values or Character Strings
  - Decimal: 1324
  - Hexadecimal: 0x3DE2 (32-bit value, zero-padded)
  - General: n_xxxx (n is base in [2,9], xxx is digit string)
  - Character: 'V' (enclosed in single quote)
  - String: "Hello world!\n"
- Control Characters Specified as in C Language
- Single Quote:' \' '
- Dollar Sign or Double Quote: Use Two in a row "$$" is a SINGLE Dollar Sign; " "" " is a single Double Quote

# Predefined Register Names

- r0-r15 or R0-R15
- a1-a4 (argument, result, or scratch registers, synonyms for r0 to r3)
- sp or SP (stack pointer, r13)
- lr or LR (link register, r14)
- pc or PC (program counter, r15)
- cpsr or CPSR (current program status register)
- spsr or SPSR (saved program status register)

# Some Common Directives

| Directive | Comment |
|---|---|
| AREA | Defines Block of data or code |
| RN | Equates a Register with a name |
| EQU | Equates a Symbol to a Numeric Constant |
| ENTRY | Declares an Entry Point to a Program |
| DCB | Allocates one or more Bytes of memory. It also specifies initial runtime contents of memory. |

# Some Common Directives

| Directive | Comment |
|---|---|
| DCW | Allocates one or more Halfwords (16 bits) of memory. It also specifies initial runtime contents of memory. |
| DCD | Allocates one or more Words (32bits) of memory. It also specifies initial runtime contents of memory. |
| ALIGN | Aligns data or code to a specific boundary |
| SPACE | Reserves a zeroed block of memory of a certain size. |
| LTORG | Assigns starting point of a literal pool. |
| END | Designates end of source file |

# AREA-Define a Block of Data or Code

- The AREA directive instructs the assembler to begin a new code or data section.
- There must be at least one AREA directive in an assembly module.
- Normal to use separate sections for code and data.
- Large programs can be divided into several code sections.
- Large independent data sets, such as tables, should also be placed in separate sections.
- Syntax is

AREA *sectionname* {, attr} {, attr}

- Where *sectionname* is name of the section and any name can be chosen as long as it does not start with a numeric digit.

# Attributes for AREA

- CODE : The section contains machine instructions. READONLY is the default
- DATA : The section contains data.  READWRITE is the default.
- READONLY: This indicates that this section should not be written to and there it can be placed in read-only memory.
- READWRITE: This section can be read from and written to and it must therefore be placed in read-write memory.

# REGISTER NAME DEFINITION

- **RN** directive defines a register name for a specified register.
- Syntax is

  *name* RN *expr*

  where *name* is the name to be assigned to the register. The parameter *expr* is a number from 0 to 15.
- Examples

```
coeff1   RN 8   ; coefficient 1
dest  RN 0   ; register 0 holds the pointer to
                ;  destination matrix
```

# EQUATE A SYMBOL TO A NUMERIC CONSTANT

- The EQU directive gives a symbolic name to a numeric constant, a register-relative value, or a program-relative value.
- Syntax : *name* EQU *expr* {,*type*}

  where parameter *type* is optional and can be any one of the following:

  CODE16, CODE32, DATA

# ENTRY

- The ENTRY directive declares an entry point to a program.
- You must specify at least one ENTRY point for a program.
- If no ENTRY exists, a warning is generated at link time.
- You must not use more than one entry directive in a single source file and not every source file has to have an ENTRY directive.

# Allocate Memory and Specify Contents

- The DCB directive allocates one or more bytes of memory and defines the initial contents of the memory.
- The syntax is

  {*label*} DCB *expr* {,*expr*} …

  where expr is either a numeric expression that evaluates to an integer in the range of -128 to 255 or a quoted string
- Examples

```
max_mark DCB 100                ;define max_mark to be 100
marks DCB 10, 20, 25, 33    ;define 4 bytes with values of 10,
                                ;20, 25 and 33
C_string DCB “C_string”, 0
;ARM assembly strings are not null-;terminated, hence a 0 is added behind
to form a null-terminated ;string like in C language.
```

# DCW

- DCW directive allocates one or more halfwords of memory aligned on two byte boundaries and defines the initial contents.
- The syntax is

  {*label*} DCW{U} *expr* {,*expr*} …

  where expr is either a numeric expression that evaluates to an integer in the range of -32768 to 65535
- DCWU is same as DCW except that there is no memory alignment
- Examples

  coeff DCW 0xFE37, 0x8ECC ;defines 2 halfwords

# DCD

- DCD directive allocates one or more words of memory aligned on 4-byte boundaries and defines the initial contents.
- The syntax is

  {*label*} DCD{U} *expr* {,*expr*} …
- DCDU is same as DCD except that there is no memory alignment
- Examples

```
data1 DCD 1, 5, 20    ;Defines 3 words containing
                      ;decimal values 1, 5, and 20
data2 DCD mem06 + 4   ;Defines 1 word containing 4+
                      ;the address of label mem06
```

# ALIGN

- The ALIGN directive aligns the current location to a specified boundary by padding with zeros.
- The syntax is

ALIGN {*expr* {,*offset*}}

where *expr* is a numeric expression evaluating to any power of two from $2^0$ to $2^{31}$.

- ALIGN itself without any *expr* sets the current location to the next word (4-byte) boundary. This is the most common usage of this directive.
- Proper alignment can speed up the efficiency of the program in some cases

# SPACE

- The SPACE directive reserves a zeroed block of memory.
- The syntax is

  {*label*} SPACE *expr*

where *expr* evaluates to the number of zeroed bytes to reserve.

- Example

```
data1 SPACE 255   ;defines 255 bytes of zeroed storage
```

# END

- The END directive informs the assembler that it reached the end of a source file.
- Every assembly language source file must terminate with END on a line by itself

# ARM Instruction Set Summary (1/4)

| Mnemonic | Instruction | Action |
|---|---|---|
| ADC | Add with carry | Rd:=Rn+Op2+Carry |
| ADD | Add | Rd:=Rn+Op2 |
| AND | AND | Rd:=Rn AND Op2 |
| B | Branch | R15:=address |
| BIC | Bit Clear | Rd:=Rn AND NOT Op2 |
| BL | Branch with Link | R14:=R15<br>R15:=address |
| BX | Branch and Exchange | R15:=Rn<br>T bit:=Rn[0] |
| CDP | Coprocessor Data Processing | (Coprocessor-specific) |
| CMN | Compare Negative | CPSR flags:=Rn+Op2 |
| CMP | Compare | CPSR flags:=Rn-Op2 |

# ARM Instruction Set Summary (2/4)

| Mnemonic | Instruction | Action |
|---|---|---|
| EOR | Exclusive OR | Rd:=Rn^Op2 |
| LDC | Load Coprocessor from memory | (Coprocessor load) |
| LDM | Load multiple registers | Stack Manipulation (Pop) |
| LDR | Load register from memory | Rd:=(address) |
| MCR | Move CPU register to coprocessor register | CRn:=rRn{<op>cRm} |
| MLA | Multiply Accumulate | Rd:=(Rm*Rs)+Rn |
| MOV | Move register or constant | Rd:=Op2 |
| MRC | Move from coprocessor register to CPU register | rRn:=cRn{<op>cRm} |
| MRS | Move PSR status/flags to register | Rn:=PSR |
| MSR | Move register to PSR status/flags | PSR:=Rm |

# ARM Instruction Set Summary (3/4)

| Mnemonic | Instruction | Action |
|---|---|---|
| MUL | Multiply | Rd:=Rm*Rs |
| MVN | Move negative register | Rd:=~Op2 |
| ORR | OR | Rd:=Rn OR Op2 |
| RSB | Reverse Subtract | Rd:=Op2-Rn |
| RSC | Reverse Subtract with Carry | Rd:=Op2-Rn-1+Carry |
| SBC | Subtract with Carry | Rd:=Rn-Op2-1+Carry |
| STC | Store coprocessor register to memory | address:=cRn |
| STM | Store Multiple | Stack manipulation (Push) |

# ARM Instruction Set Summary (4/4)

| Mnemonic | Instruction | Action |
|---|---|---|
| STR | Store register to memory | <address>:=Rd |
| SUB | Subtract | Rd:=Rn-Op2 |
| SWI | Software Interrupt | OS call |
| SWP | Swap register with memory | Rd:=[Rn]<br>[Rn]:=Rm |
| TEQ | Test bitwise equality | CPSR flags:=Rn EOR Op2 |
| TST | Test bits | CPSR flags:=Rn AND Op2 |

# Summary

- You should be able to edit, debug, step, run a simple assembly program in Keil µVision.
- Understand the pros and cons of writing in assembly language.
- Know the difference between Assembly directives and ARM instructions.
- Have some idea of the ARM instruction format.