Classwork 1 – Keil ARM quick start guide

Basics

1. launch Keil ARM Simulator, version uVision V5.22.0.0
2. click on the `Project` menu, select `New Microvision Project`. Take note that a bright blue font indicates input to and from the computer
3. enter a file name in the `File name` field . Let's say `MyFirstExample`
4. click on the `Save` button
5. this will cause the `Select Device for Target 'Target 1'` window to appear. Now you have to Choose the family and version of the processor that you are going to use
6. in the list of devices, select `ARM`, and then select `ARM7 (Big Endian)`
7. click on the `OK` button. The window will disappear, you will return to the uVision main window
8. we need to enter the program. Click `File`, choose `New`. A `Text1` editing window will appear. Now we can enter our simple program. For instance:

```
    AREA MyFirstExample, CODE, READONLY
    ENTRY
    MOV r0, #0x4        ; load 4 into r0
    MOV r1, #0x5        ; load 5 into r1
    ADD r2, r0, 0xe1    ; add r0 to r1 and put the result in r2
S   B   S               ; force infinite loop by branching to this line
    END                 ; end of program
```

9. after typing the program, select `File`, then `Save` on the menu, enter `MyFirstExample.s` as the file name. The suffix `.s` indicates that it is a source
10. the system will return you to the window, which will now be called `MyFirstExample`
11. now you must set up the environment. Click `Project` in the main menu, from the drop-down list, select `Manage`. In the list, click `Select Components, Environment, Books ..`
12. now you will get a form with three windows. At the bottom of the right window, select `Add Files`
13. click on the selection arrow file type (file extension) and select the file type `Asm Source File (* .s *; *.src; * .a *)`. You should now see your own file `MyFirstExample.s` in the window. Select it and go to the `Add` tab. It adds the source file to the project. Click `Close`. Now you will see the file `MyFirstExample.s` in the far right window. Click `OK` to exit
14. you are ready to assemble your file
15. click `Project` in the main menu and click on `Built target`
16. in the bottom `Build Output` window you will see the result of the assembly process. You should see something like:

```
*** Using Compiler 'V5.06 update 4 (build 422)', folder: 'G:\Keil_v5\ARM\ARMCC\Bin'
Build target 'Target 1'
assembling MyFirstExample.s...
linking...
Program Size: Code=16 RO-data=0 RW-data=0 ZI-data=0
".\Objects\MyFirstExample.axf" - 0 Error(s), 0 Warning(s).
Build Time Elapsed: 00:00:01
```

17. The magic phrase: `0 Error(s)`. If you get an error, you should find the error in the source file and repeat the steps to assemble again: click on `Project` and `Build target`

Example 1. Addition

Problem: Calculate P = Q + R + S
Assume Q = 2, R = 4, S = 5. Suppose that r1 = Q, r2 = R, r3 = S
Save result P in register r0

- Pseudo code:

```
ADD r0, r1, r2     ; add Q and R, save the result in P
ADD r0, r0, r3     ; add S and P, save the result in P
```

- Assembly code:

```
     AREA Example1, CODE, READONLY
     ADD r0, r1, r2
     ADD r0, r3
Stop B   Stop
     End
```

- Notes:

1. a semicolon indicates a user-entered comment, everything after it is ignored by the assembler
2. the first line of `AREA Example1, CODE, READONLY` is an assembler directive and is required to run the program. This is a development system feature, not ARM assembler. Assemblers various companies may have different directives for determining the start of a program. In our case, `AREA` refers to a code segment, `Example1` is the name that we gave it, `CODE` indicates the beginning of the executable code rather than the data area, and `READONLY` indicates that this is not a modifiable piece of code
3. everything that is in the first column of the code (in our case `Stop`) represents a label that can be used to refer to this line
4. the `Stop B Stop` command means "branching to the line labeled `Stop`" and is used to create an endless loop. This is a convenient way to terminate programs in simple examples, like this one
5. the last line of `END` is an assembler directive that indicates that there is no more code to be executed. This directive ends the program

Because there are no means of inputting input data into registers, you must do it manually. Just double click on the desired register and change its value.
Note that the content of `r0` is 2 + 4 + 5 = 11 = 0x0B. This is the result that we expected.

Example 2. Addition

Calculate P = Q + R + S
Q = 2, R = 4, S = 5 and let r1 = Q, r2 = R, r3 = S
In this case, before running the program, we will place the data in memory with constants

- Pseudo code:

```
MOV r1, #Q          ; load Q into r1
MOV r2, #R          ; load R into r2
MOV r3, #S          ; load S into r3
ADD r0, r1, r2      ; Add Q to R
ADD r0, r0, r3      ; Add S to (Q + R)
```

Here we use the `MOV` instruction, which copies a value to a register. Values may be the contents of another register or literal. A literal is indicated by #. We can write, for example, `MOV r7, r0`, `MOV r1, #25` or `MOV r5, #Second`. We used the symbolic names `Q`, `R`, and `S`. We must associate these names with the actual values. This can be done using the `EQU` assembler directive:

```
Q  EQU  2
```

This directive maps the symbolic name `Q` to a value of `2`. If a programmer uses `Q`, it is exactly the same as using `2`. The purpose of using `Q`, not `2`, is to make the program more readable.

- Assembly code:

```
     AREA Example2, CODE, READONLY
     MOV r1, #Q     ; load r1 with the constant Q
     MOV r2, #R
     MOV r3, #S
     ADD r0, r1, r2
     ADD r0, r0, r3
Stop B   Stop

Q    EQU 2          ; Equate the symbolic name Q to the value 2
R    EQU 4          ;
S    EQU 5          ;
     END
```

Example 3. Addition

Calculate P = Q + R + S
Q = 2, R = 4, S = 5 and let r1 = Q, r2 = R, r3 = S
In this case, before starting the program, we will place the data in memory in the form of constants. First we use the register load command `LDR r1, Q`, to load register r1 the contents of the memory cell referenced by `Q`. Such a command does not actually exist, and it is not in the ARM command set. But ARM assembler automatically changes it to a real instruction. The `LDR r1, Q` command is called a pseudo instruction and it makes life easier for the programmer.

- Pseudo code:

```
LDR r1, Q              ; load r1 with Q
LDR r2, R              ; load r2 with R
LDR r3, S              ; load r3 with S
ADD r0, r1, r2         ; add Q to R
ADD r0, r0, r3         ; add in S
STR r0, Q              ; store result in Q
```

- Assembly code:

```
     AREA Example3, CODE, READONLY
     LDR r1, Q          ; load r1 with Q
     LDR r2, R          ; load r2 with R
     LDR r3, S          ; load r3 with S
     ADD r0, r1, r2     ; add Q to R
     ADD r0, r3         ; add in S
     STR r0, Q          ; store result in Q
Stop B   Stop

     AREA Example3, CODE, READWRITE
P    SPACE 4            ; save one word of storage
Q    DCD 2              ; create variable Q with initial value 2
R    DCD 4              ; create variable R with initial value 4
S    DCD 5              ; create variable S with initial value 5
     END
```

Notice how we must create the data area at the end of the program. We reserved space for `P`, `Q`, `R`, and `S`. Used the `SPACE` directive to reserve 4 bytes of memory for variable `S`. After that we reserve the memory area for `Q`, `R` and `S`. In each case, we use the `DCD` assembler directive to reserve a word (4 bytes) and initialize it. For instance

```
Q  DCD  2              ; create variable Q with initial value 2
```

means "call the current line Q and save the word 0x00000002 in this place." If you look at the disassembled code, you can see that the pseudo-instruction `LDR r1, Q` translates into the actual ARM instruction

`LDR r1, [PC, #0x0018]`. It is still the load command, but the addressing mode in this case is register indirect. The address is calculated by adding the contents of the program counter (PC) and hexadecimal offset 0x18. Note also that the instruction counter is always 8 bytes more than the address of the current instruction. This is a feature of the ARM processor pipeline. Therefore, the operand address is: [PC] + 0x18 + 8 = 0x0 + 0x18 + 0x08 = 0x20. If you look at the memory display area, you will see that the contents of cell 0x20 is `0x00000002`.

Example 4. Addition

Calculate P = Q + R + S
Q = 2, R = 4, S = 5
In this case, we use register indirect addressing for accessing variables. In other words, we need to set a pointer to the variables and have access to them through this pointer.

- Pseudo code:

```
ADR r4, TheData                ; r4 points to the data area
LDR r1, [r4, #Q]               ; load Q into r1
LDR r2, [r4, #R]               ; load R into r2
LDR r3, [r4, #S]               ; load S into r3
ADD r0, r1, r2                 ; add Q and R
ADD r0, r0, r3                 ; add S to the total
STR r0, [r4, #P]               ; save the result in memory
```

- Assembly code:

```
        AREA Example4, CODE, READWRITE
        ENTRY
        ADR     r4, TheData     ; r4 points to the data area
        LDR     r1, [r4, #Q]    ; load Q into r1
        LDR     r2, [r4, #R]    ; load R into r2
        LDR     r3, [r4, #S]    ; load S into r3
        ADD     r0, r1, r2      ; add Q and R
        ADD     r0, r0, r3      ; add S to the total
        STR     r0, [r4, #P]    ; save the result in memory
Stop    B       Stop
P       EQU     0               ; offset for P
Q       EQU     4               ; offset for Q
R       EQU     8               ; offset for R
S       EQU     12              ; offset for S

AREA Example4, CODE, READWRITE
TheData SPACE   4               ; save one word of storage for P
        DCD     2               ; create variable Q with initial value 2
        DCD     4               ; create variable R with initial value 4
        DCD     5               ; create variable S with initial value 5
        END
```

This code writing is not very effective, the code is too long, but nonetheless, this program illustrates several concepts. First, the `ADR r4, TheData` instruction loads the start address of the data region (`TheData` label) into r4 register. That is, r4 indicates the beginning of a data region. If you look at the code, you can see that four bytes are reserved for P, and then the values for Q, R and S were loaded into successive memory cells by word. Note that none of these cells have a label. The `ADR` instruction (load an address into a register) is a pseudo-instruction. If you look at the disassembled code in, you can see that this instruction is translated into the command `ADD r4, PC, # 0x18`. Instead of loading the actual address of the `TheData` label into r4, it loads the contents of the PC command counter plus bias. Programmers need not worry about how ARM is going to translate the command `ADR` into real code. This is the beauty of pseudo-instructions, the assembler itself will do the right thing. When it is necessary to load `Q` into r1, the `LDR r1, [r4, #Q]` command is used. This is an ARM command for register loading with literal offset, i.e. `Q` value. If you look at `EQU` area, then `Q` is equal to 4 and, therefore,

register r1 is loaded with the value data that is 4 bytes from the cell where r4 points to. This memory location is where data corresponding to `Q` is stored.

Example 5. Addition

Calculate P = Q + R + S
Q = 2, R = 4, S = 5
This time we will write the program in a more compact form. To simplify the code, we used simple numerical offsets (since there was relatively little data and comments tell us what actions happen). Notice we used the Q, R, and S labels for data. These tags are redundant and not necessary, as they are not mentioned anywhere in the program. But there is no mistake. These tags simply serve as a reminder for the programmer.

- Assembly code:

```
      AREA Example5, CODE, READWRITE
      ENTRY
      ADR    r0, P           ; r4 points to the data area
      LDR    r1, [r0, #4]    ; load Q into r1
      LDR    r2, [r0, #8]    ; load R into r2
      ADD    r2, r1, r2      ; add Q and R
      LDR    r1, [r0, #12]   ; load S into r3
      ADD    r2, r2, r1      ; add S to the total
      STR    r1, [r2]        ; save the result in memory
Stop  B      Stop

      AREA Example5, CODE, READWRITE
P     SPACE  4               ; save one word of storage for P
Q     DCD    2               ; create variable Q with initial value 2
R     DCD    4               ; create variable R with initial value 4
S     DCD    5               ; create variable S with initial value 5
      END
```

Note also that we reused registers to avoid using more registers. In this example, only r0, r1, and r2 are used. If after use of the register its value is no longer involved in the program, then it may be reused. However, this can make debugging more difficult. In this example in one part of the program the register r1 contains Q and at another point S. And, finally, it contains the result S.

Examples 1-5. Conclusion

Programs using the Keil ARM IDE begin with

```
AREA <nameOfProg>, CODE, READONLY
```

and end with

```
END
```

- data can be stored in memory until the program starts using the DCD directive (define a constant)
- you can write `ADD r0, r1, #4` or `ADD r0, r1, K1`. However, if you use a symbolic name (for example, K1), then you need to use the `EQU` operator to equate it (tie it) to the actual value
- some instructions are pseudo instructions. They are not real ARM instructions, but they are a short form of command notation, which automatically translates to one or more ARM instructions
- the instruction `MOV r1, r2` or `MOV r1, #<literal>` has two operands and moves the contents to a register or literal to register

Example 6. Arithmetic expression

Calculate the value of the arithmetic expression `(A + 8B + 7C - 27) / 4`
A = 25, B = 19 and C = 99. This example uses literals.

- Pseudo code:

```
MOV r0, #25             ; Load register r0 with A which is 25
MOV r1, #19             ; Load register r1 with B which is 19
ADD r0, r0, r1, LSL #3  ; Add 8 x B to A in r0
MOV r1, #99             ; Load register r1 with C which is 99 (reuse of r1)
MOV r2, #7              ; Load register r2 with 7
MLA r0, r1, r2, r0      ; Add 7 x C to total in r0
SUB r0, r0, #27         ; Subtract 27 from the total
MOV r0, r0, ASR #2      ; Divide the total by 4
```

There are a few things to note. Firstly, multiplication or division by a power of 2 can be done by shifting to the left or right, respectively. For this, in the ARM command system there are corresponding commands. For example, the instruction `ADD r0, r0, r1, LSL #3` means: shift the contents of the register r1 left three times (multiply by 8) and then add it to the contents of register r0 and the result put in r0. Secondly, we can use addition and multiplication (`MLA`) instructions to execute the operation `P = P + Q * R`. In our case, we can multiply 7 x C and add the result to the current value in r0. Pay attention to the format of this manual. Finally, we can divide by 4 by applying two shifts to the right to the dividend.