Sheffield Hallam University
Semester I of academic year 2021/22

"SoC architectures and FPGA prototyping"

Lab 5 – Coding a Cortex-based SoC using Keil MDK

In this lab you will familiarise with the industry-standard IDE Microcontroller Development Kit (MDK, or Keil MDK or ARM Keil MDK) that can be used for firmware development and debug of a variety of microcontrollers, based on Cortex-M technology, from various vendors. (Michael Keil from Germany founded the Keil company that is now part of ARM.) Our SoC is quite memory-constrained (1 kB or 256 code words only) and for this reason we are unable to subject it to a comprehensive firmware test. Therefore, your development will involve a bit of unconventional C programming that is nevertheless is quite common and useful in the embedded engineering.

You need to download the IDE from either https://www2.keil.com/mdk5 (with registration) or using the direct download link (~1 GB)
https://www.keil.com/fid/jq1hvwwtdy9j1w0oc2w1y1yme9xb1dgwsye6d1/files/eval/mdk536.exe

You may also want to download the device pack that will be required for the lab 6 (~0.3 GB) as the part of your MDK installation.

Follow the installation video from Panopto (link for the MDK 5.33 were used there; use the above link for the MDK 5.36 instead) (***SoC_Labcasts/SoC_Lab_5/Installing_MDK5_STM32H7pack***) .

You will use the Lite version at home; it is sufficient for the purposes of labs 5 and 6.

Both the MDK and STM32H7 device pack are available in the Sheaf 4301 Electronic CAD lab (professional version).

Assignment 1. Simulate a given assembly code using Keil MDK.

Download Lab5 archive from Blackboard and unzip it in your SoC_Labs folder.

Navigate to the folder ***Lab5/ASM_blink*** and double click the ***code.uvprojx*** (this is the project file in the Keil MDK, similar to the .***qpf*** project files in Quartus). The project is fully set up already, but should you want to find how to start a project from scratch (like you did in the lab 0 for the FPGA) please have a look at the ARM handout.

Compile the code by pressing F7 (editing mode of MDK Keil):

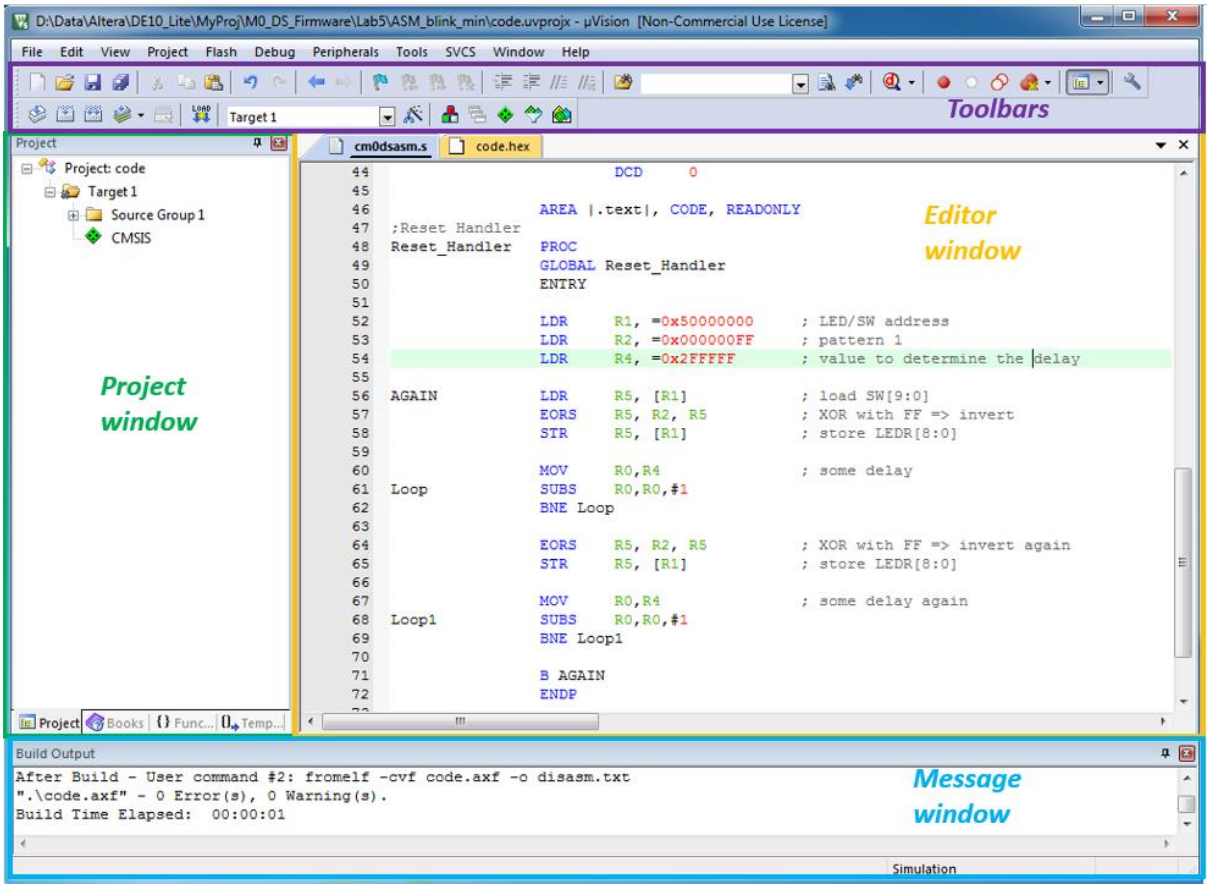

Figure 1 MDK Keil windows after compiling the code

The code shown above first sets some registers, at the line labelled AGAIN reads the values from the switches (address 0x50000000), inverts it (by XORing with 0xFF), writes it back on the LEDs (same address 0x50000000), takes a delay to let humans observe the outcome, inverts again (i.e. restores the original value read from the switches), again writes it back to the LEDs, takes another delay to allow for observing the output, and finally branches back to AGAIN (forever loop or superloop – typical behaviour of an embedded program).

How to delay? We load the R4 register with a reasonably high value (0x2FFFFF) then decrement it. If the result is zero, we continue execution of the code but if not then we branch to the line (Loop or Loop1) decrementing it over and over again.

Select Debug in the top menu then click Start/Stop Debug Session (or press Ctrl+F5, debug mode):

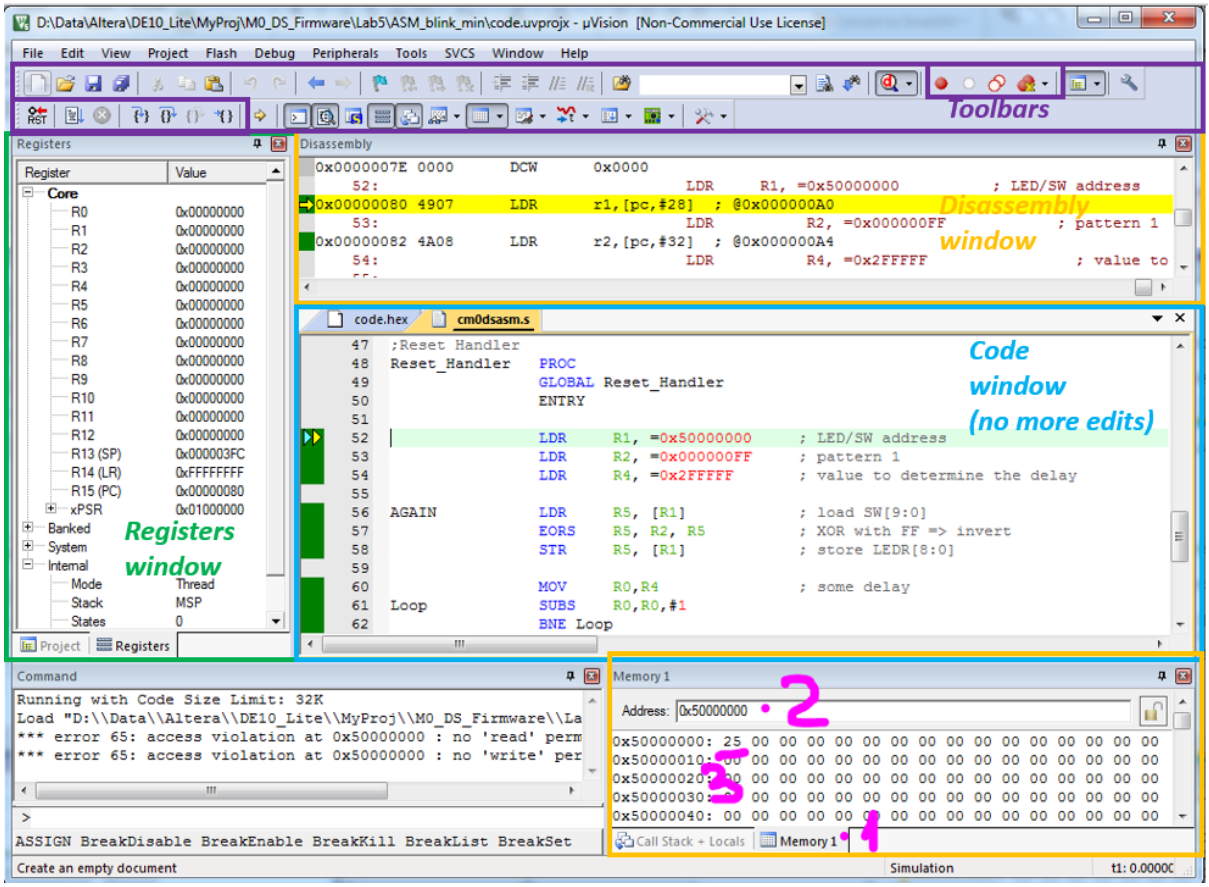

Figure 2 Keil MDK window in the debug mode

If, after entering debug mode, the two triangles (blue and yellow in the editor window) do not point to the line 52 (LDR R1,…), please change the Reset Handler address in the line 11 from 0x21 to 0x81).

Familiarise with the icons available on the toolbars. Familiarise yourself with the buttons used to control the breakpoint (places in the code where you want the code execution to pause in order to observe/change variables etc) and code execution (single step, run to cursor, run (will stop at the next breakpoint), reset etc). These buttons are supplied with tips that became visible when you hover over them by your mouse.

Registers window will highlight registers that changed compared to the previous view.

Disassembly windows takes the output code and displays it along with the memory addresses and associate assembly instructions along with the original code text. It does not show statement labels though. It is more useful for simulating C programs as stepped execution worked for compiled code rather than original C statements.

(Please note: you can see that the majority of the used instructions require 2B (16b) rather than the complete 4B for the 32 bits Cortex M0. That is because ARM introduced the code compression feature called first Thumb and now Thumb 2. Most common instructions can be coded using 16 bits only, and they are expanded to the full 32 bits inside the CPU. What is the gain then? Reduction in the size of the flash memory that is required to store these instructions, which is worthwhile to reduce the cost of a microcontroller.)

Code window presents the program that no longer can be edited but where you can see where does the simulation pauses at the moment (triangles). Green bars mark the lines that were executed at least once; grey bars – the others.
Before proceeding with the simulations, we also need to edit the memory map - telling the MDK how different memory areas are supposed to be used by the code in order to detect any possible memory use violations (e.g. trying to fetch instruction from the data memory).
Modify the memory map (Debug->Memory Map) as shown below to enable simulating values for switches (when the address 0x50000000 is read by the Cortex-M0), and observe the value written into the LEDs (when the address 0x50000000 is written into by the Cortex-M0):

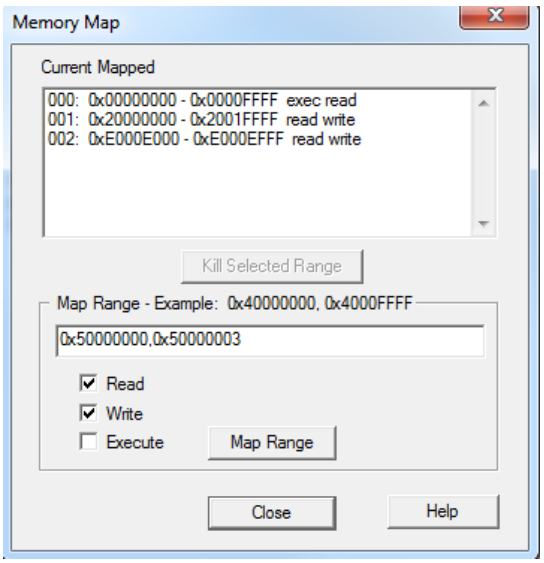

Figure 3 Modifying the memory map

Then click on the memory tab in the bottom right corner (as shown by purple 1 in Fig.2), enter address 0x50000000 in the Address field (purple 2) then enter the last two decimal digits of your student ID number at the addess 0x50000000 as hexadecimal number (purple 3; 25 enetred here as an example for 37 = 0x25). The content of your MDK window will match that of the Fig.2.

You are now ready to proceed with the simulation, first using single step execution (step - execute the following statement or step over – if the following statement is a subroutine call, fully execute this subroutine and stop at the following statement).
Notice changes in the registers and memory after each instruction.
Learn how to get through the delay loops (inserted in order to lower the speed of the MCU operation to be clearly observed by humans) by using the breakpoints.
Inspect the memory content (address 0x0) and observe that each instruction is stored in the little-endian manner (i.e. LSB at the lower memory address).

For the report you will need to collect two screenshots of the full Keil MDK window straight after the execution of the store LEDR[8:0] statements (lines 58 and 65).

At this point you have experienced firmware development and debugging for a well-established SoC architecture. As you have seen, it can start even before the SoC is fully prototyped and/or manufactured even when the SoC includes some custom peripherals (LED module in this case that allows setting the LEDs and reading switches on the DE10-Lite FPGA board).

Assignment 2. Compile the given Keil projects (first *Lab5/ASM_blink* then *Lab5/C_blink*), prepare the appropriate files to be used with Quartus, include them to the Lab4 FPGA project, update the FPGA bitfile and run the Cortex-M0 code on the FPGA board.

There are quite a few steps to get to the end of this assignment, but they are straightforward: (1) compile you code (like in the assignment 1) to get a HEX file; (2) convert the HEX file into a MIF file using Matlab; there is a video of how to do this on Panopto); (3) place obtained MIF file into the Quartus project folder; (4) update the code file in the Quartus project and run it on the FPGA.

All the MDK projects were configured to produce, in addition to the AXF file used to program MCUs (you will implicitly use this file in the lab 6), a HEX file with the CPU binary codes. You can review the content of this file; each line defines 4 B. Unfortunately, this file is not suitable to be used in Quartus as it is. Moreover, it does not comply with the "proper" Intel HEX format (https://en.wikipedia.org/wiki/Intel_HEX).

Converting the compiled output into MIF format (easier to use than the "proper" HEX) can be done with the Matlab script I wrote **`hex2mif.m`** . Some of you have installed Matlab on your computers, and there is the online option I personally used. You need to register for a free Matlab account with your university credentials; after this following the video on Panopto should be quite straightforward. If you have Matlab installed locally, make sure that both the **`hex2mif.m`** and **`code.hex`** to convert are placed in the same folder.

The new (converted) file should be called **`code.mif`** and needs to be placed in the same folder where you have the QPF file for the lab 4. Update the code file for the lab 4 project and run it on the FPGA. First, you will need to execute the ASM_blink code, and take two photographs of different values shown by the LEDs. Second, you need to execute the C_blink code, take a photograph and figure out why did the code stop where it stopped.

Useful advice: you do not need to recompile the whole Quartus project when you only need to change the MIF file. Instead overwrite the old file with the file with the same name (**`code.mif`**) then click (1) ***Processing/Update memory initialisation file*** then re-assemble the bitstream ((2) ***Processing/Start/Start Assembler***) as shown below

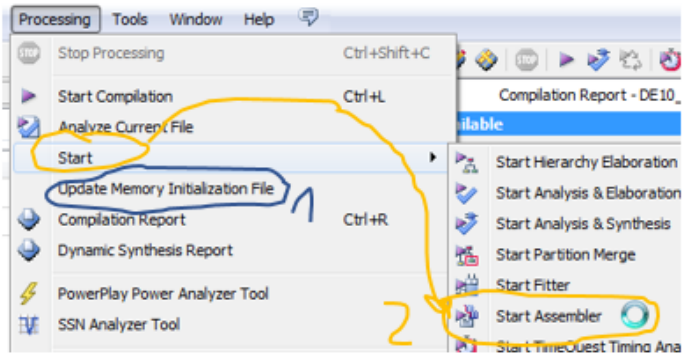

Figure 4 How not to recompile the whole Quartus project

Assignment 3. You are going to program in C and implement using your designed SoC the following combinational digital circuit for w=3 starting from the project, located in the folder lab5_3:

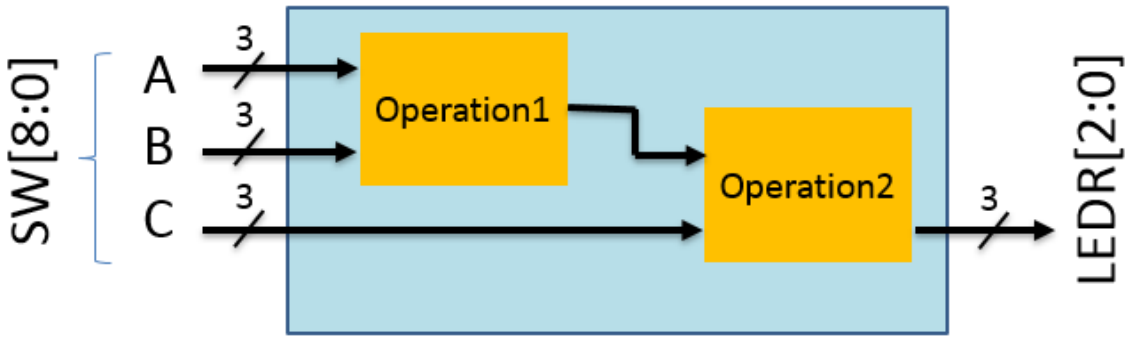

Figure 5 Schematic of the logic circuit to implement

The input data to the circuit will be set by the on-board switches; the output data will be displayed by the on-board LEDs.

Your design will depend on your seven-digit student ID number, here ABCDEFG, as follows:

- ***bitwise*** operations to implement
-

G	0	1	2	3	4	5	6	7	8	9
Op1	AND	AND	NAND	NAND	OR	OR	NOR	NOR	XOR	XNOR

F	0	1	2	3	4	5	6	7	8	9
Op2	NOR	XNOR	OR	XOR	NAND	XNOR	AND	XOR	NAND	OR

- the nine LSbits of your student ID number will define the input combination to be used for testing.

The students with ID number 123456**7** will use

- operations NOR and AND respectively;
- use input data of 0100 00111 as shown below (1234567_{10}=100101101011010000111_2 using any online decimal to binary converter; leaving only 3*3 = 9 LSBs we get 010_000_111 to set using SW[8:0].

Write your C code, compile it, convert to MIF, assemble the Quartus project and run it on board.

Guidance note for C programming

Our SoC will return the state of all the SW[9:0] switches when the memory location 0x50000000 was read. However you will only need to use the state of a few switches, for example, SW[5:3] to find one of the datum coming to the middle line in the schematic diagram.

Highly recommended reading – note on C programming on BB.

Here is a code snippet to put the read values of the required switches only in the variable B:

```
uint32_t *ptr=0x50000000; // pointer to the LED peripheral location

uint32_t tmp, B; // temporary and required variable


tmp = *ptr; // read the value of all the switches

// 1. mask off all the unnecessary bits -
// all the bits except for the SW[5:3] ones will become zeroes
// when ANDed with 0
// but these bits will remain in the wrong place
B = tmp & 0x38; // 0_000_111_000 = 0011_1000 = 0x38

// 2. shift the selected bits to the right by 3 shifts
B = B >> 3;

// process SW[8:6] here using the mask 0x1C0 and 6 shifts

// process SW[2:0] here using the mask 0x7 and no shifts
```

The variable B is fully ready to be used for calculation of the result. You will need to have three similar snippets of C code for your assignment to get inputs A, B and C.

After calculating the result (for the example below, by logical NOR and AND, and masking off all the unused bits to display LEDR[2:0] only), display it by sending to the LED peripheral as follows

```
*ptr = (~(A | B) & C) & 0x7;
```