EE8205: Embedded Computer Systems

Electrical, Computer & Biomedical Engineering, Ryerson University

Study of ARM Cortex-M3 Features

1. Objectives

The purpose of this lab is to introduce students some of the features of Cortex-M3/M4 embedded processor such as of bit-banding, conditional branching, and barrel shifting. This will help you to understand the Cortex M-series processors being commonly used in the embedded systems. In particular, students will have hands-on experience with bit-banding, conditional branching, and barrel shifting to examine their efficiency both in Debug and Target mode using performance assessment techniques. You will apply this knowledge at the end of the lab and implement an LED bit-band application.

2. Creating the New Project

1. When uVision has launched, select Project >> New uVision Project in the main menu bar. If a project already exists, first close the project by selecting Project >> Close Project.
   Select New uVision Project as shown in Figure 1.

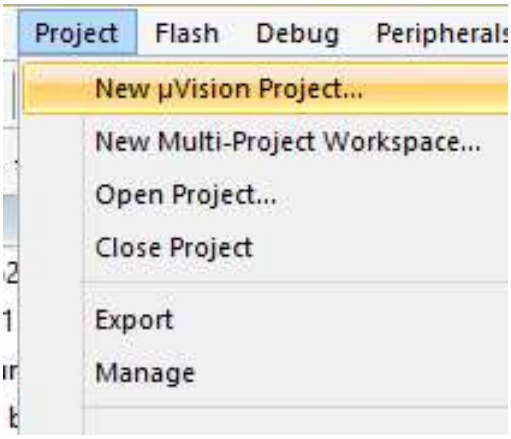

Figure 1

2. You should see a window like the one shown in Figure 2. Select the icon for "New Folder" and name your working folder “Lab2". Name the project as “Bitband” and Press Save.

3. Type “LPC1768” as shown in the Figure 3 below and select the device and press okay.

4. Select the following Packages from Run Time Environment and add those to your project Figure 4.
   a. Board Support>LED
   b. CMSIS>CORE
   c. Compiler>Event Recorder
   d. Device > Startup, GPIO, PIN
   e. Press okay once selected

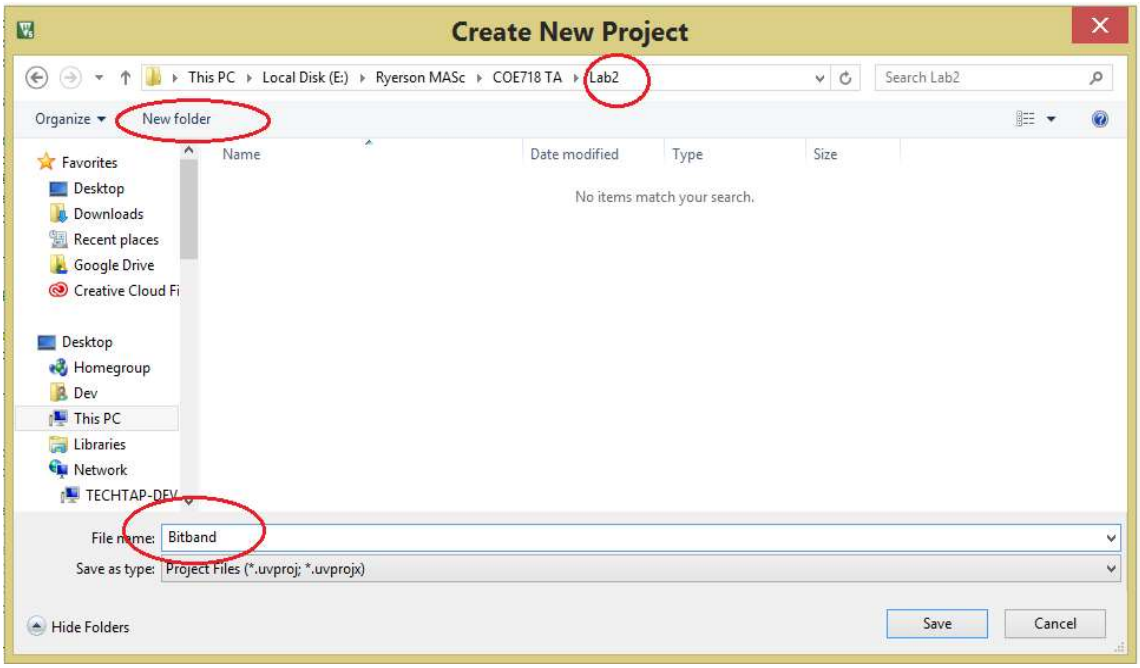

Figure 2

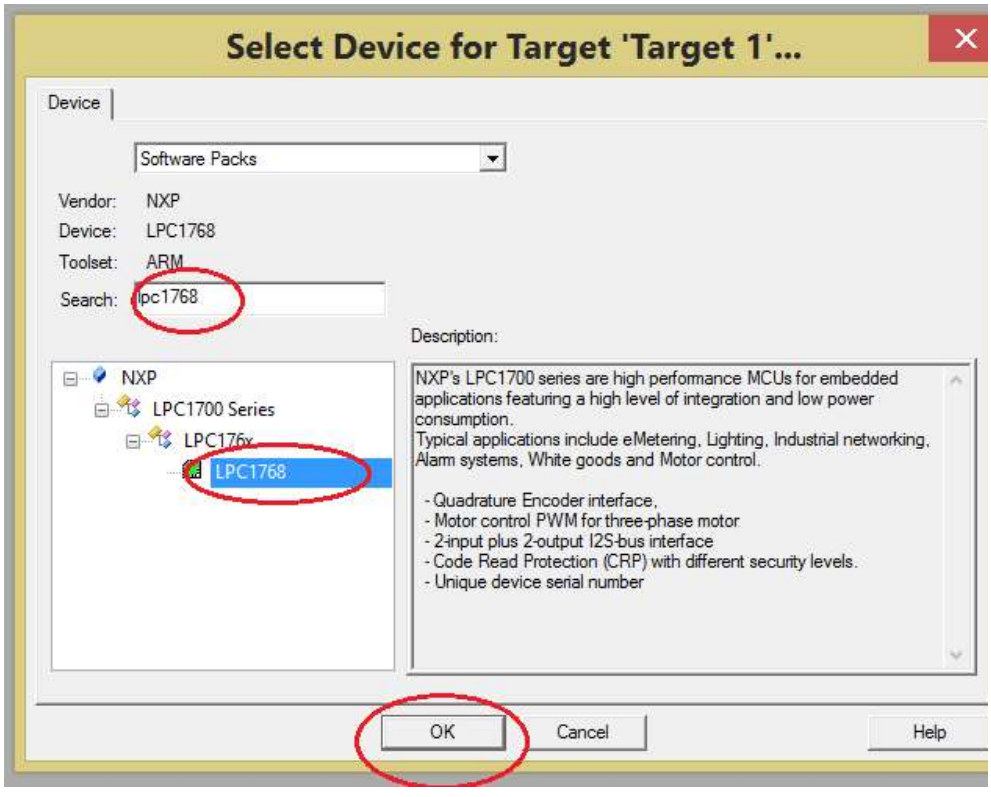

Figure 3

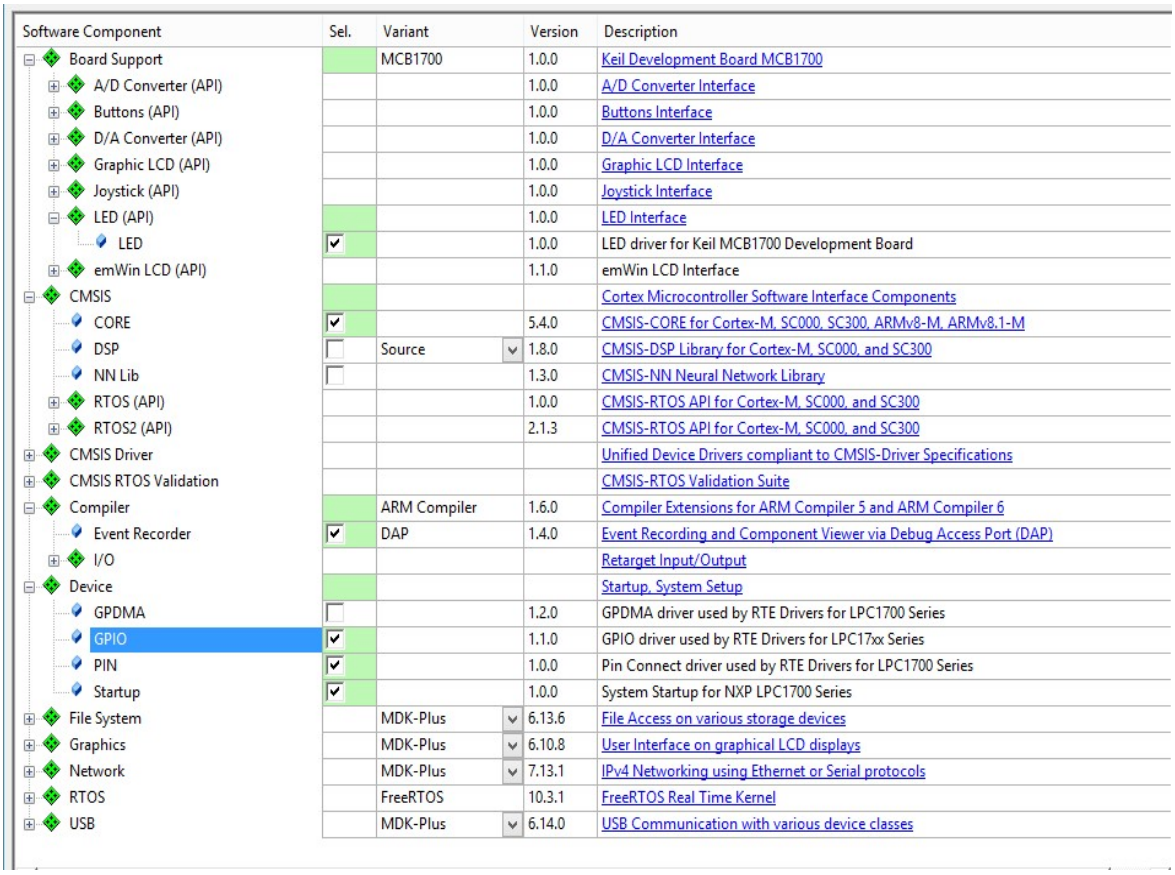

Figure 4

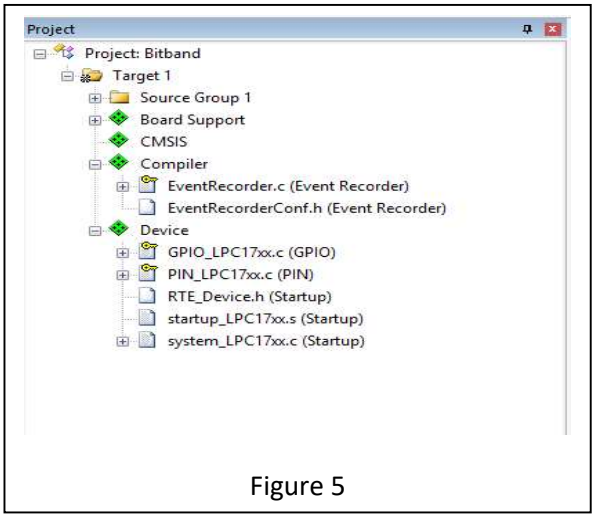

Figure 5

5. Now your Project files should look like the Figure 5 given below.

6. Right click on Source Group1 folder select “Add New Item to Group ‘Source Group 1’ as shown in Figure 6.

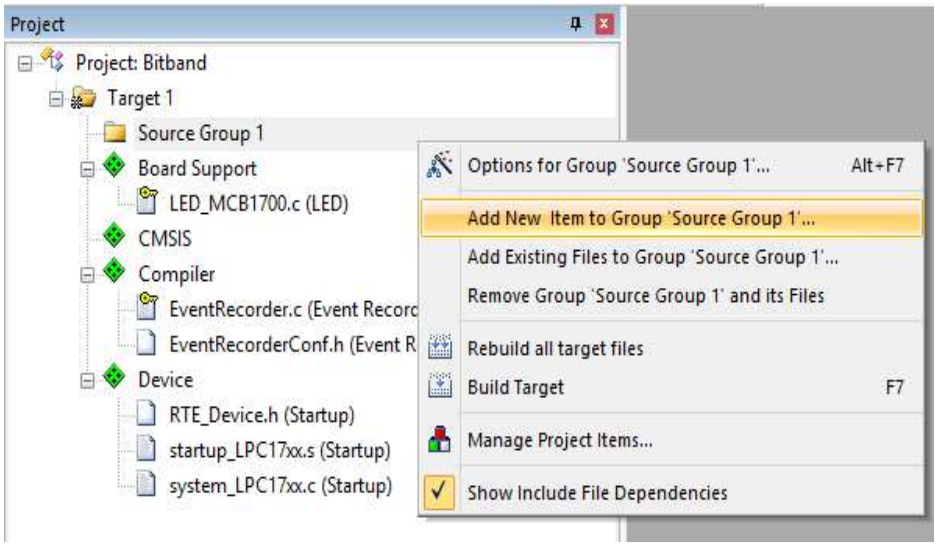

Figure 6

7. Select C File(.c) name it 'bitband' click Add as depicted in Figure 7. The C file will be added to the project.

8. Repeat the Step 7, and this time select Header File (.h) name it 'bitband' click Add as shown in Figure 8. The file bitband.h will be added to your project directory.

9. Now your Project Folders should look like the Figure 9.

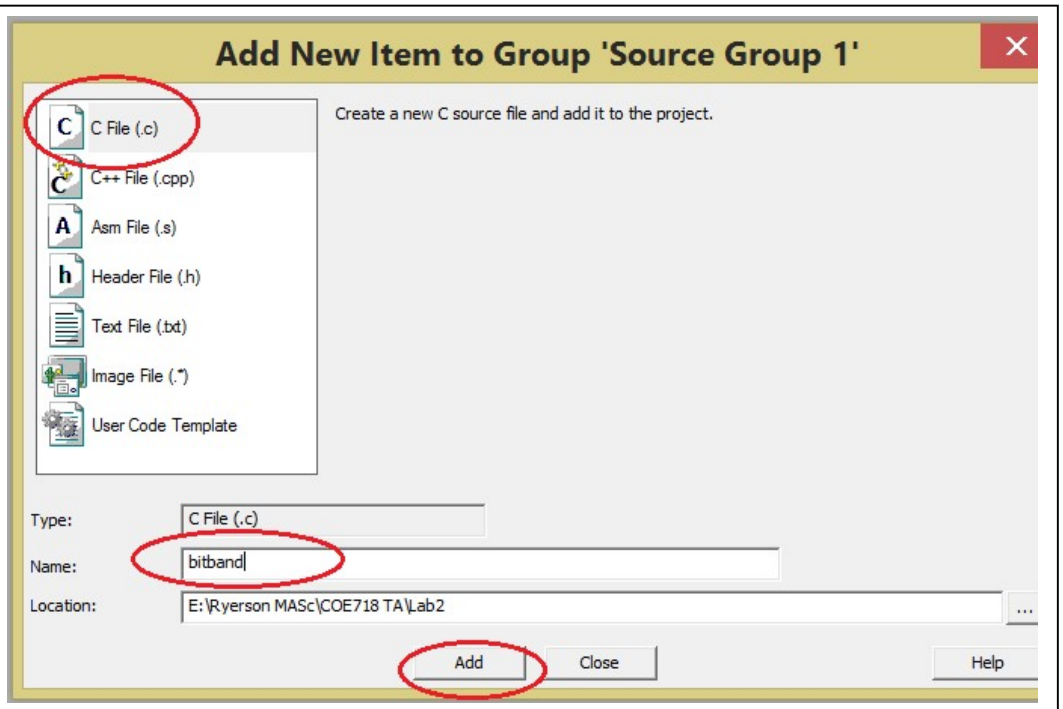

Figure 7

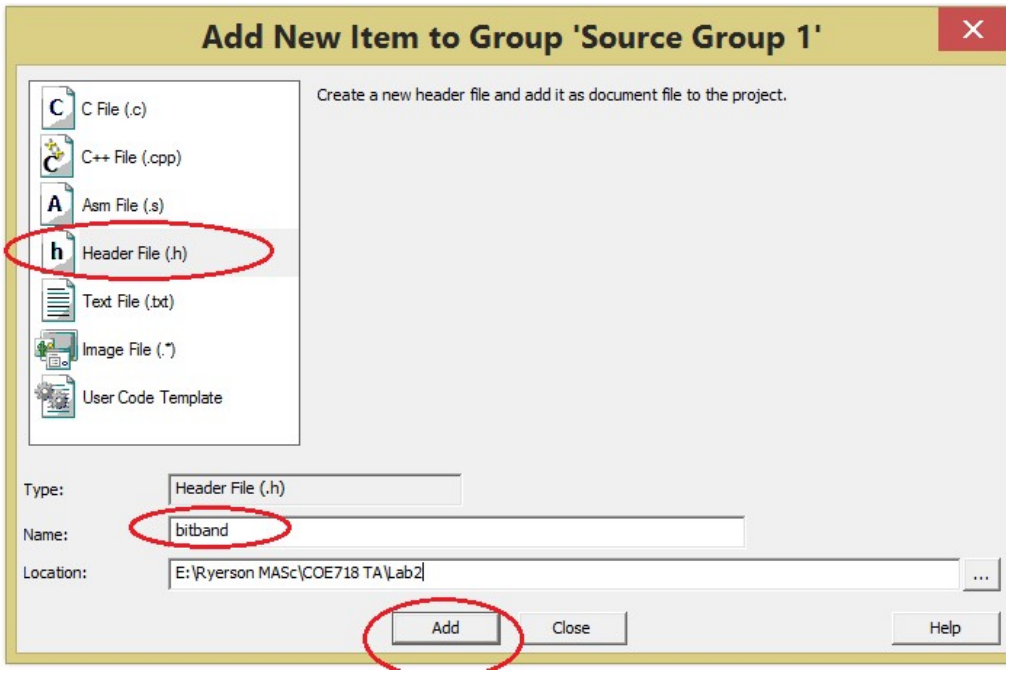

Figure 8

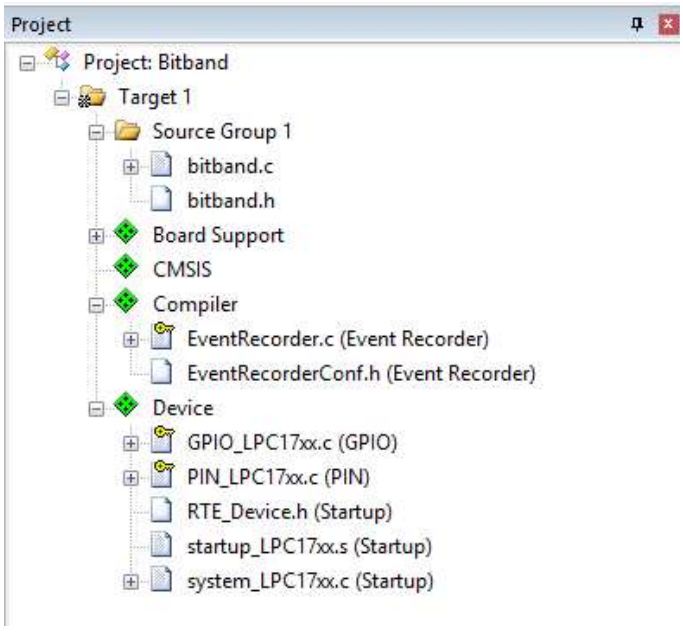

Figure 9

Figure 10

10. Click on the  icon on the top menu as shown in Figure 10.

11. Perform the following changes as shown in Figure 11. Under Target, ARM Compiler> **Use default compiler version 5**. Moreover, check> **Use Micro LIB**, and uncheck**> IRAM2**

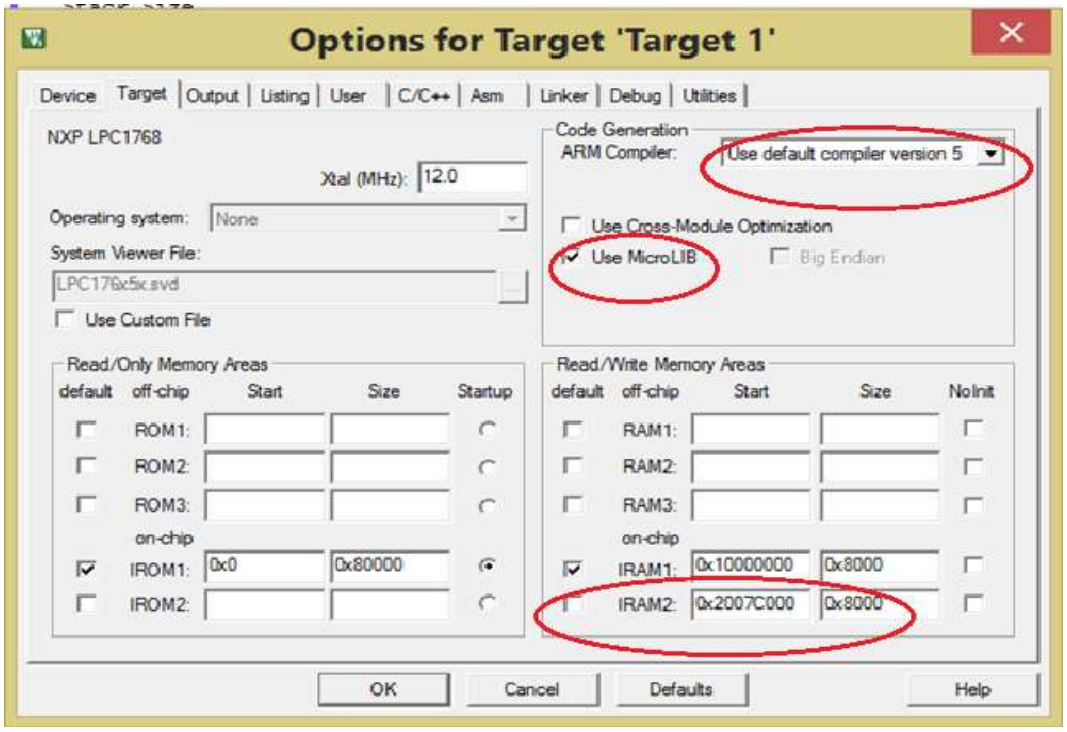

Figure 11

8. Select C/C++ and check **C99 Mode** option as shown in Figure 12.

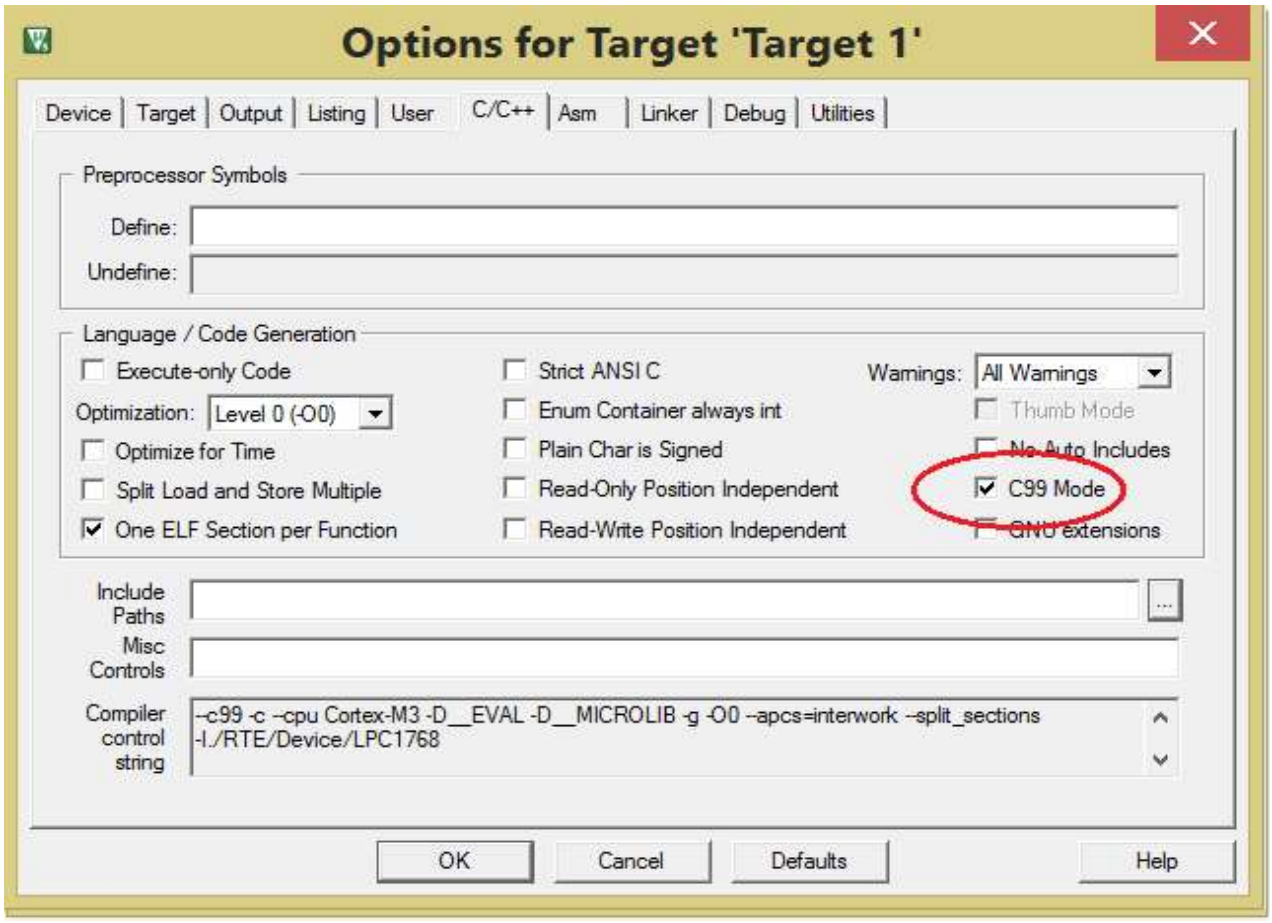

Figure 12

9. Select Debug option and do the following changes: check **Use Simulator** option. Moreover, choose the CPU and DLL parameters as given below.
   a. For CPU DLL Parameter in **Dialog DLL**: choose DARMP1.DLL, **Parameter**: -pLPC1768
   b. For Driver DLL Parameter in **Dialog DLL**: TARMP1.DLL, **Parameter**: -pLPC1768

Debug Window should look like the one in Figure 13, then press **Okay.**
Now the Project Directory is all set for the lab.

3. Bit Banding

3.1 Definition

When implementing applications for embedded systems, there is often a need to clear and set individual bits within peripheral and SRAM registers. For instance, to check when an A/D conversion is complete, it is necessary to check the status flag for completion, obtain the value, and then reset the flag to obtain a new conversion. Consequently, bitwise AND and/or OR masks are needed to check, set, and clear the flags. The Cortex-M processors provide a more efficient implementation to perform these frequent actions, known as bit-banding.

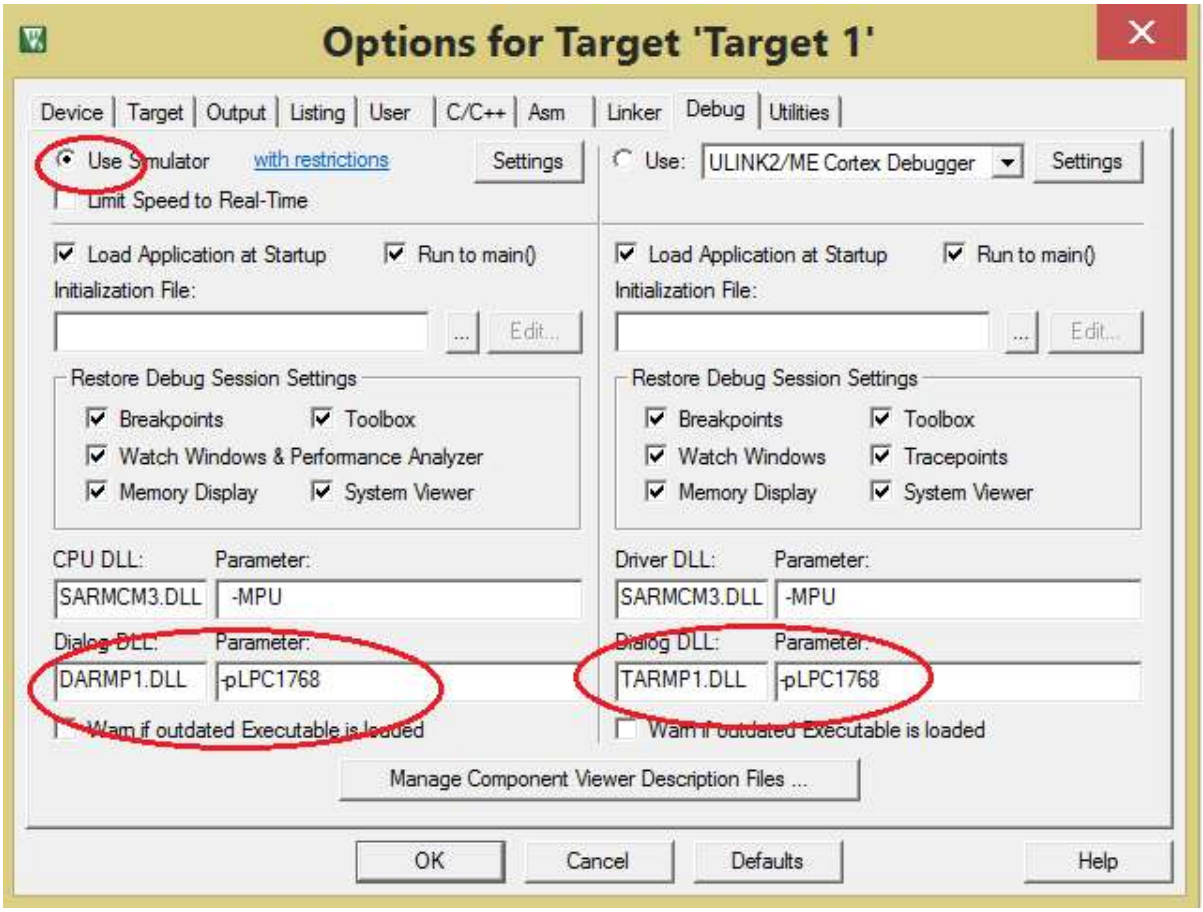

Figure 13

Bit-banding allows *individual* bits in the SRAM and peripheral registers to be read or written to, instead of reading the whole register and masking the desired bits. These SRAM words and peripheral registers are *bit addressable*.

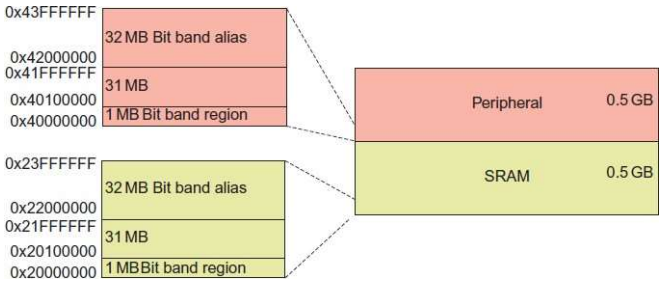

Figure 14: Address Allocation of SRAM and Peripheral Regions

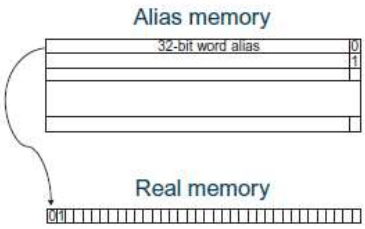

Figure 15: Bit to word Mapping

The Cortex-M3 has a predefined memory map to access its components, with the bit banding portion shown in Figure 14. In total, the LPC1768's memory map consists of 4GB. The first 0.5GB is allocated to code storage. The next GB (starting from 0x20000000) is of particular interest to bit banding; the rest of the memory (2.5 GB) is dedicated to external RAM and devices, private internal accesses, and vendor specific needed memory. As seen in Figure 14®, the first MB of both the SRAM (0x20000000) and peripheral regions (0x40000000) are dedicated bit band regions. Each bit in the ***bit band region*** is aliased to one word (32 bits) in the ***bit band alias*** region as shown in Figure 15®. Therefore, if we were to consider the entire 1MB region of SRAM registers in the Cortex-M3's bit band region, this would actually be aliased to 32MB (1MB*32bits) of virtual word addresses (in the alias region). In other words, each 1 or 0 accessible in a single SRAM or peripheral register is given its own 32-bit word in a separate part of memory (bit band alias region) which start at addresses 0x22000000 and 0x42000000 respectively.

3.2 Calculating the Bit Band Word Address

To use bit banding, the desired (register) bit's bit band address must be calculated based on the bit's real memory address. Once calculated, a pointer in your code is used to reference the calculated address so that it may alias the 32-bit region and modify the bit directly (without masking). The formula to calculate the address of the bit's alias is as follows:

$$\textit{Bit Band Word Address} = \textit{Bit Band Alias Base Address} + (\underline{\textit{Byte Offset}} * 32) + (\textit{Bit Number} * 4) \quad (1)$$

$$\underline{\textit{Byte Offset}} = \textit{Bit's Bit Band Base Address} - \textit{Bit Band Base Address} \quad (2)$$

where:

- *Byte Offset*:
  - *Bit's Bit Band Base Address* - the base address for the targeted SRAM or peripheral register (i.e., the effective address of the port)
  - *Bit Band Base Address* - for SRAM = 0x20000000, for Peripherals = 0x40000000
- *Bit Band Alias Base Address* - for SRAM = 0x22000000, for Peripherals = 0x42000000
- *Bit Number* - the bit position of the targeted register (i.e., pin of the port)

3.3 Example Calculation of Bit Banding

As an example, let us take the SRAM address 0x2008C000, and try to modify bit 3. We use this address to implement and observe the different techniques (bit banding and masking) to help illustrate the efficiency of the bit banding method. We must first calculate the bit band word address of this bit's bit band base address.

Calculate the Word Address:
Substituting the values in equations (2) and (1):

Byte Offset = 0x2008C000 - 0x20000000
= 0x0008C000

Bit Band Word Address = 0x22000000 + (0x0008C000 * 0x20) + (0x3 * 0x4)
= 0x22000000 + (0x1180000) + 0xC
= 0x2318000C

Define a Pointer to the Address:

```
#define BIT_ADDR= (*(volatile unsigned long *)0x2318000C)
```

Assign a Value to the Port Bit:

```
int
main(void) {
        ...
        BIT_ADDR = 1;
}
```

®Martin, T., "The Designer's Guide to the Cortex-M Processor Family", Elsevier Ltd, 2013.

3.4 Bit Banding Example Application

Go to D2L under Contents>Labs>Lab2 folder you will find bitband.h, bitband.c and cond_ex.c files.

Download all the files.

a. Copy the contents of bitband.c to your project bitband.c file
b. Copy the contents of bitband.h to your project bitband.h file

Once uVision is setup, open bitband.c and examine the code. Follow the steps and logic given in Section 3.1 to 3.3 to understand how the .c code is implementing for bit banding.

Build the project. Enter debug mode.
Click reset. From the main menu select **Debug >> Execution Profiling >> Show Times.** Run the application to completion. **Verify the expected code output to the Debug (printf) viewer.** Examine and analyze the execution times of the 3 different methods. What do you notice?

4. Conditional Execution

4.1 Definition

Now, time to think in terms of assembly code and registers. The program status register (PSR) in the Cortex-M processor contains several CPU status-flags that indicate conditions raised during code execution (ex: "r5 - r5" will raise a zero flag). Figure 16 presents the Cortex M3's PSR. The most important bits to note for this lab are the N (Negative), Z (Zero), C (Carry), V (oVerflow) and If Then (IT) flags. As learned in various lectures and courses, these status flags are essential for branching and general control- flow execution. The Cortex-M family invokes several conditional branching techniques to minimize branch penalties for performance efficiency. One of the primary techniques used is fetching instructions through speculation, using the PSR and compiler optimization.

Figure 16: PSR and Status Flags

4.2 Different Types of Conditional Execution with Cortex-M3

A unique feature of the Cortex-M series is conditional execution: instructions executed by the CPU do not affect the flags in the PSR unless explicitly stated. Therefore, the CPU and ARMv7 ISA uses special instructions suffixed by an 's' to update its PSR flags (ex: SUB**S**, ADD**S** versus SUB, ADD etc.). Conditional branches (and/or conditional instructions- see below) then use the PSR flags to effectively find the next (correct) instruction to execute. The exception to the 's' suffix rule is the CMP and/or TST instructions for updating the PSR (which do not require 's'). An example of the 's' suffix rule is as follows.

```
ADD   r2, r3, #5
SUBS  r1, r1, r2
BEQ   function1
```

The ADD instruction will add 5 to register r3 and place the result in r2 without affecting the PSR. The next instruction will then subtract r1 from r2, and place the difference in register r1 with the ability to modify the PSR flags as SUB**S** is appended with an S. If the Z flag is raised, then the program will branch to function1. Table I presents various condition codes that are exclusive to the ARM instruction set, along with their respective PSR flag tests. ARM also allows non-control flow-based instructions to be appended with conditional codes. Instruction appending allows for more efficient coding and processor performance.

Let's consider the following code:

Conditional Instruction Method Versus Non-Conditional Method

```
CMP r2, #5  //if (a <= 5)
MOVLE r2, #10 //a = 10;
MOVGT r2, #1 //else a = 1;
```

```
    CMP r2, #5
    BGT t_else
    MOV r2, #10
t_else: MOV r2, #1
```

Aside from achieving code density, these conditional instructions allow the processor to avoid unnecessary branch prediction techniques if a condition is not met (i.e., BGT t_else). Therefore, the pipeline does not have to be flushed and/or refilled with the correct instructions if a branch is mis-predicted. With the conditional instructions, the PSR flags may be directly evaluated by the instruction. For example, if the PSR flags of the first instruction listed above (i.e., MOVLE) does not evaluate to true according to the conditional code flags generated by CMP (the LE flag), then the instruction is simply treated as a NOP and the next instruction is executed. Although the NOP instruction may utilize processor time, this technique nonetheless still avoids the need to flush and refill the pipeline as required of typical processors.

Table I: Instruction Condition Codes

Condition Code	xPSR Flags Tested	Meaning
EQ	Z = 1	Equal
NE	Z = 0	Not equal
CS or HS	C = 1	Higher or same (unsigned)
CC or LO	C = 0	Lower (unsigned)
MI	N = 1	Negative
PL	N = 0	Positive or zero
VS	V = 1	Overflow
VC	V = 0	No overflow
HI	C = 1 and Z = 0	Higher (unsigned)
LS	C = 0 or Z = 1	Lower or same (unsigned)
GE	N = V	Greater than or equal (signed)
LT	N! = V	Less than (signed)
GT	Z = 0 and N = V	Greater than (signed)
LE	Z = 1 and N! = V	Less than or equal (signed)
AL	None	Always execute

The second and more efficient conditional execution technique used by the Cortex-M is referred to as IF-THEN (IT) Blocks. In this case, if an IF condition consisting of less than four instructions is present in an IF statement block, ARMv7 compiles all the instructions within the block for conditional execution using its ISA's IT (If-Then) instruction. An IT block may possess many forms such as ITE (If-Then-Else), ITTE (If-Then-Then-Else - meaning that there is an IF condition consisting of 2 instructions if evaluated to true, and one Else instruction if evaluated to false), ITTEE (If-Then-Then-Else-Else - 2 instructions each for then and else clauses), ITEE etc. Let's take a simple IT example which evaluates R0 and R1 in assembly:

```
CMP    R0, R1
ITE    EQ
ADDEQ R4, R3, R2
ASRNE R4, #4
```

Here, if R0 equates to R1, ADDEQ is conditionally executed. If not, ASRNE is executed, and ADDEQ is treated as a NOP. Note that when the ITE instruction is executed, the IT flag will also be raised in the PSR with an "ITE EQ" status.

4.3 Conditional Execution Example Application

1. With the uVision application open, download the cond_ex.c file from D2L and put it in place of bit banding files. Right click on **Source Group 1> Manage Project Items** as shown in Figure 17.

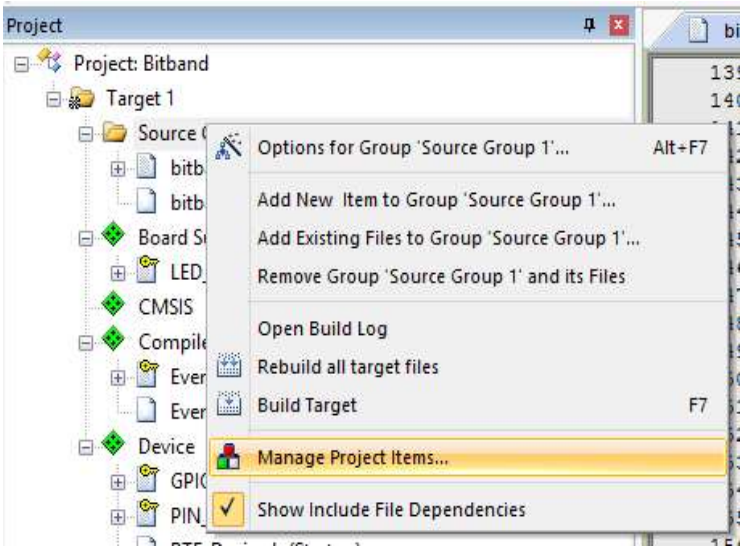

Figure 17

2. Delete bitband.h and bitband.c files as shown in Figure 18.

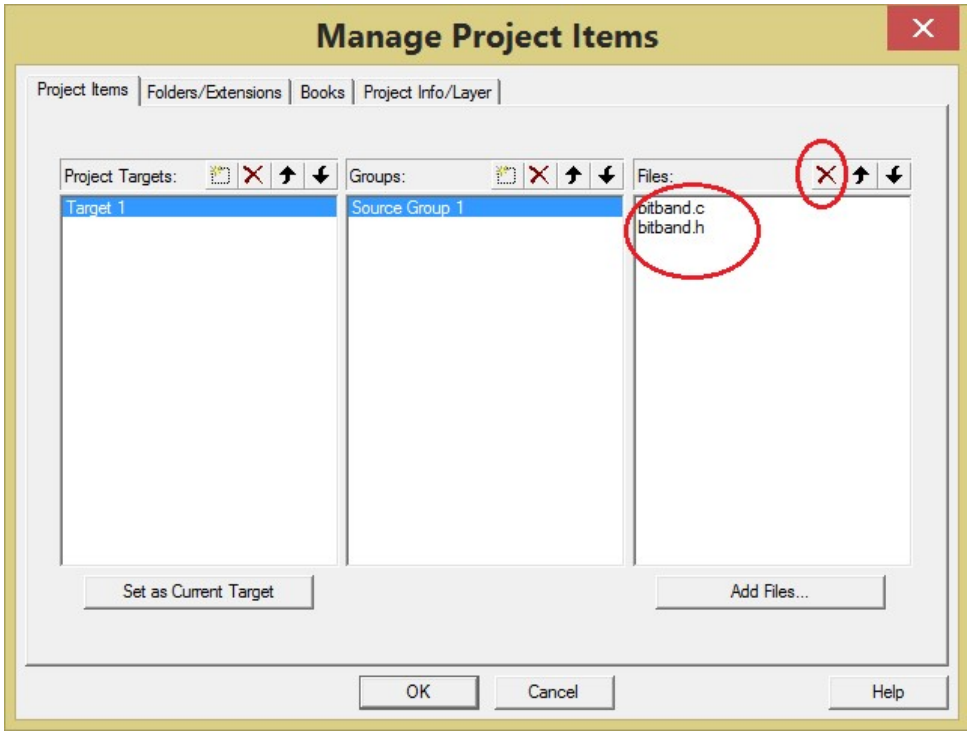

Figure 18

3. Add a new *.c file **'cond_ex.c'** to Source Group 1. Copy and paste the cond_ex.c code from D2L file to this project file.

4. Prior to building the project, in the main menu select Project >> Options for 'Target...' (or  button). Click on the C/C++ tab. In the "Optimization" drop-down box, select Level 0 (-O0) and click OK.

5. Build the project and enter debug mode. Place a breakpoint after *"int main(void) {"*. Expand the "xPSR" register in the "Registers" Window.

6. Reset and run the code. Your code will halt at the breakpoint. From this point, step through the code to see the conditional code generated by the ARM Compiler. This level of optimization employs the 's' suffix technique. Open the Performance Analyzer and note the total execution time of the .c code and entire application with the base -O0 optimization. (60.880us). See Figure 19.

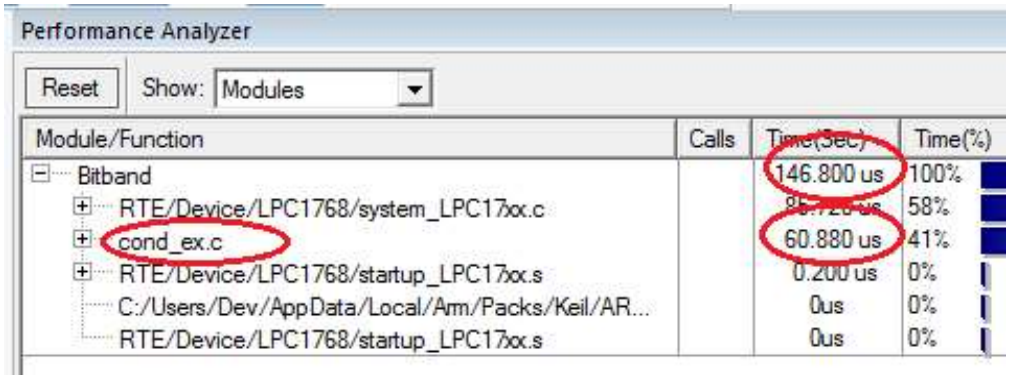

Figure 19

7. Then exit the Debug mode. Take out the breakpoint from the code. In the main menu, once again select Project >> Options for 'Target...' (or ). Click on the C/C++ tab. In the "Optimization" drop-down box, select Level 3(-O3). Underneath the optimization drop-down, there is a checkbox titled "Optimize for Time". Check this option to enable time optimization and click OK.

8. Build the project with the new optimization level and enter debug mode. Place a breakpoint after *"int main(void) {"* and expand the xPSR register. Reset and run the code. Your code will halt at the breakpoint. From this point, step through the code to see the IT conditional code optimized by the compiler in the disassembler. Make note of how the xPSR registers change during execution of the IT block. Note the total execution time of the .c code and entire application with the base ---O3 / time optimization selected. (60.730us). See the performance analyzer in Figure 20.

This conditional execution efficiency is presented on a small scale, and thus as the size and complexity of an application increases, the benefits of the ARM instruction set optimization will also transpire.

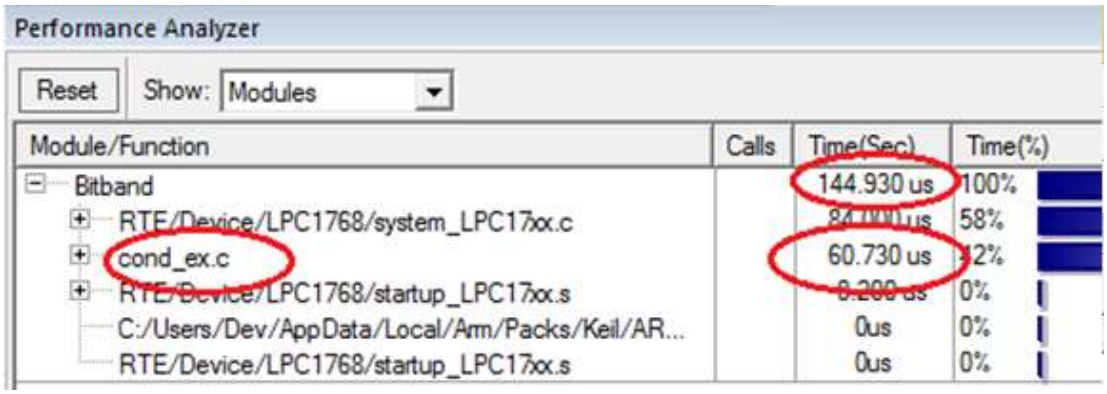

Figure 20

5. Barrel Shifter

5.1 Definition

The Cortex-M3 contains a 32-bit barrel shifter that can rotate/shift an instruction's operand prior to inputting values into the ALU. This technique is extensively used in DSPs and multiply-accumulate units for signal processing. Thus, a line of code such as *a = b + (c*d)* could be executed in a single cycle as:

```
ADD R1, R2, R3 LSL R4        Versus        MUL R1, R2, R3
                                           ADD R5, R1, R4
```

5.2 Barrel Shifting Example

Go back to the code given to you for the conditional execution example. Append the main function so it looks as follows (or alternatively uncomment the bottom portion of the code):

```
int r1 = 1, r2 = 0, r3 = 5;
while (r2 <= 0x18) {
  if ((r1 - r2)> 0) {
        r1 = r1 + 2;
        r2 = r1 + (r3*4);
        r3 = r3/2;
   } else { r2 = r2 + 1;}
}
```

Make sure that the optimization level is set to -O3 and "Optimize for Time" is selected. Compile the project and open Debug mode. Eliminate all the breakpoints. Set a breakpoint in the .c code where barrel shifting would apply. Enable the instruction execution time display in Debug mode by selecting Debug >> Execution Profiling >> Show times.

Reset and run the code. Once the breakpoint is reached in the assembly version, continue stepping through the program. Notice the use of both the 's' suffix and IT blocks in addition to barrel shifting optimization.

6. Optional Lab Assignment (Bonus Marks: 2% of the Course Mark) Submit Lab-2 Report through D2L

Using the knowledge obtained from this lab, create an application which lights-up 2 to 3 LEDs on the MCB1700 using bit banding. You are expected to:

- Use two different methods to light up the LEDs, masking and bit banding mode. These 2 methods will be invoked as one application (i.e., one .c file).
- Calculate the effective address with the aid of Table 101 in the NXP LPC17XX User manual.
  - Use **at least two** LEDs from **two different ports** (will require you to calculate a minimum of two

effective addresses) -- i.e., LED1 = **P1**. 28, LED2 = **P2**.2

- Use a conditional execution method to turn the LEDs on and off for the three different methods. *Explicitly state in your code (with comments) whether you have invoked the 's' suffix or the ITE conditional execution method.*
- Find an application which requires barrel shifting. Find a creative way to integrate it into the above code. Display the equation (and/or method) along with its result printout in the debug window.

Once you have these basics working, then you will need to create a Debug performance analysis version.

1) Debug (Performance Analysis) Version

In Debug mode, implement and analyze the execution times for the 2 different LED bit-banding methods. Write a report explaining how you implemented the 2 different methodologies, the calculations you performed to implement each of the methods, the performance outcomes of each method, along with any other information needed. What technique did you use to measure each bit-banding method's execution time? Include the following table in the report:

Method	Execution Time (-O0)	Execution Time (-O3)	Performance Improvement
Masking			
Bit Banding			

Moreover, discuss the conditional execution method you have used. You need to use Debug mode and its tools for analyzing these application details.

2) Target (Demo) Version

The demo version will require you to turn the LEDs on and off with the two different methods implemented and invoke the barrel shifter function.

In order to explicitly observe the LEDs light up (i.e., Peripheral pin check-box for the three different methods), you will need to invoke a delay function in the methods. Insert these delays in your code each time you turn an LED on or off. Use the print function to display the method currently executing. Use the execution profiling tools to determine the total execution time of one call to your delay function- include this value in your report. Also include a brief description of the barrel shifting method you used in your code and the result obtained.

You are expected to deliver the following:

- Print out of the report, the separate .c codes (i.e., analysis and demo version) and any supporting files that you may have adjusted to implement the lab. Moreover, print out any assembly code that proves your .c code has implemented the required Cortex-M3 features.
- Present your demo, displaying the LEDs flashing on and off i.e., through the peripheral window and printout of the debug window for the bit-banding and barrel shifting functions. Bonus Marks of 2% of the course will be awarded based on your attempt and creativity.

References

1. "The Keil RTX Real Time Operating System and µVision" www.keil.com. Keil an ARM Company.
2. "Keil µVision and Microsemi SmartFusion" - *Cortex-M3 Lab* by Robert Boys www.keil.com.

Acknowledgement

This tutorial has been adapted from introductory notes by Robert Boys "Cortex-M3 Lab" and "The Keil RTX Real Time Operating System and µVision" available at www.keil.com.