University of Waterloo
Electrical and Computer Engineering Department

Digital Computers
ECE-222 Lab manual
For the Texas Instruments Tiva-C

Winter 2017

December 12, 2016

Developed by: Rasoul Keshavarzi, Hiren D. Patel
Assisted by: Roger Sanderson, Eric Praetzel, Gordon B. Agnew

Contents

General Information

This section contains general information about the ECE-222 lab. All lab contents and resource are posted on the University of Waterloo on-line learning system called LEARN. It is a password protected environment and can be accessed here: learn.uwaterloo.ca

In the Fall 2012 term we shifted from the Freescale ColdFire® to the ARM® CPU for ECE-222. The Coldfire CPUs was exceptionally good for teaching the basics but many devices today use an ARM based CPU. Please report typos, errors, or other challenges in this manual the Lab Instructor. We appreciate your feedback and cooperation.

In Fall 2015 we started using the Texas Instruments Tiva-C microcontrollers. At $13 they are a cheap development platform with superior power and instruction performance to Arduino. We currently use the Keil MDK development software but it runs under Windows only. TI Code Composer Studio and Energia work under Linux, Mac or Windows but are not supported for ECE 222. There are many other ARM development kits available – however support for assembly language programming is often lacking.

Lab schedule

There are three hour lab sessions scheduled for this course are listed here:
http://www.adm.uwaterloo.ca/infocour/CIR/SA/under.html

Lab groups

All labs are to be done in groups of two students. Groups of three, or more, students are prohibited. If a lab session contains an odd number of students, every effort will be made to pair-up the single student with another student from other sessions, if students' schedules allow.

It is expected that both members will put equal effort into the lab tasks during the term. Unequal participation or other conflicts in a group should be brought to the lab instructor's attention at the earliest possible time.

Lab marking

The course marking scheme is stated in the Course Outline.

There are three main components related to each lab session. The lab manual for each experiment will tell you what you will need to submit for that component.

- **Prelab.** It is designed to get you started with the task. Once you accomplish what is asked in this section, you will be ready to start coding in assembly language.
- **Lab session/Demo**. You will present your work to a lab staff to be marked for that section. Some questions will be posed to students regarding the contents, procedures, debugging, and techniques used to get the code working correctly.
- **Lab report**. You will submit a report which often contains your assembly language code, and a TA will mark your report.

Different labs carry different marks allocated to them. Marking sheets are in the lab manual.

Warning: Failure to complete ALL labs may result in an Incomplete mark for the course. This means each student is expected to attend all lab sessions.

Due dates and on-time delivery

Lab reports and lab demonstration sessions will carry some marks associated with each experiment. They should be treated like examination sessions. Students should not miss them without a legitimate reason, otherwise they will lose some marks.

If you have an interview scheduled during a lab demonstration session, or if you have to miss a lab demonstration session because of another legitimate reason, please inform the Lab Instructor to avoid being recorded as 'Absent'. They will try to assign you to another session for that particular lab.

Details about deadlines and penalties are included in the Course Outline.

Electronic lab report submission is done through LEARN (learn.uwaterloo.ca).

Lab-0: Introduction to the ARM platforms in the lab

Objective

We will familiarize ourselves with the basics of the ARM board used in the ECE-222 lab. Here is a short list of what we will do in this session:

- Introduction to the ARM board
- Introduction to µVision4 software
    - How to create or open a project
    - How to build, or assemble, a target
    - How to download object code into memory on the target board
    - How to debug code
    - How to use the simulator

What you do

In this lab you will load, assemble, download, and run some short programs. Each program performs a specific task. For example, one program loads some values into some registers and them adds them up. You will confirm the result by checking the contents of the registers in debug mode.

Pre-lab

N/A

Introduction to hardware and software

In order to get students familiarized with the tools used in the ECE-222 lab, let us take a closer look at the hardware and software used in the lab. More details can be found in Appendix A.

Hardware

Figure 1.1 shows the Tiva board. The board employs a TM4C123GH6PM micro-controller unit (MCU) made by Texas Instruments. There are several input/output peripheral devices available on the board.

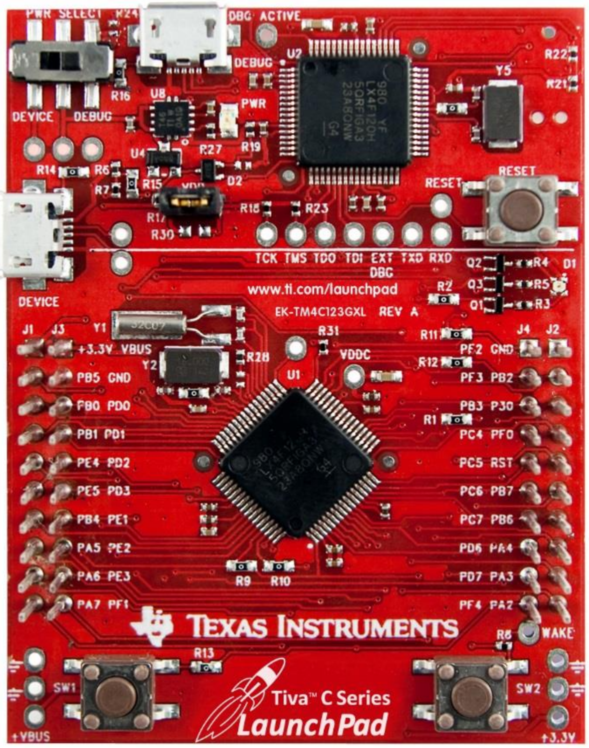


Figure 1.1 – The Tiva C TM4C123Gboard [2]

The heart of the board is the TM4C123G MCU (microcontroller unit), which contains a CPU, on-chip flash memory, RAM (Random Access Memory) and some peripheral blocks.

Software

The software toolchain used to program the Tiva board is µVision® developed by the [Keil](http://www.keil.com).

uVision for ARM is here http://www.keil.com/arm/mdk.asp and can be downloaded here: https://www.keil.com/demo/eval/arm.htm

The µVision® toolchain has been designed for high-level programming languages such as C. However, the board can be used to develop assembly language programs.

To support the Tiva board, install the Keil MDK software on your Windows computer and then install support for the Texas Instruments Tiva board. After installing MDK the “pack” installer will start. Search for Tiva on the left, select it and click on install on the right beside Texas Instruments … Update other packs as suggested. The Stellaris ICDI driver (http://www.ti.com/tool/stellaris_icdi_drivers) also needs to be installed to program and communicate with the Tiva board.

In-lab procedure

First, we will build and run code on the MCU. Then we will review how to debug the code.

Running assembly language code on the MCU

Follow the following steps in order to get yourself familiarized with the µVision toolchain.

1 Run the software by clicking on **Start/All Programs/Keil µVision5**
2 Click on the **Project** tab, and choose **New µVision Project**
3 Select or create a subdirectory on N: drive (like N:/ECE_222/Lab_0), then assign a name to your project (ie Lab0 … it can be different then the folder name), then click on **Save.**. **DO NOT MAKE A DIRECTORY, FILE OR PROJECT NAME WITH A SPACE IN IT!** A space will prevent simulation from working properly.
4 To select a CPU, double click on **Texas Instruments** and select **TM4C123GH6PM**. Click **OK**
5 This step is done outside of the µVision5 software. Copy the provided **Startup.s** file and the sample program **Lab0_tiva.s** from Learn to the folder used for this lab (step 1 above). **DO NOT USE My Documents!**
6 Switch back to the uVision5 screen, and right click on the **Source Group 1** under **Target 1.** Select **Add Existing Files to Group 'Source Group 1' …** . Select **All Files** from '**Files of type'** drop-down menu, which will list all files in the folder. Select **Startup.s,** click **Add**, then select the file **Lab0_tiva.s** , click **Add** then click **Close**.
7 To set the correct debugger right-click on the **TARGET 1** and choose the **Options for Target 'Target 1'** and then click on the **Debug** tab. Choose **Stellaris ICDI** on the right pane, you will need the Tiva board connected to the computer you are working on the debugger window should look like Figure 1.4
8 Now you are ready to assemble your code. This is called 'Build target' in the µVision software. Click on **Project** tab and then on **Build Target** or hit **F7**. The target, or binary code, for programming the Tiva, should assemble with no errors.

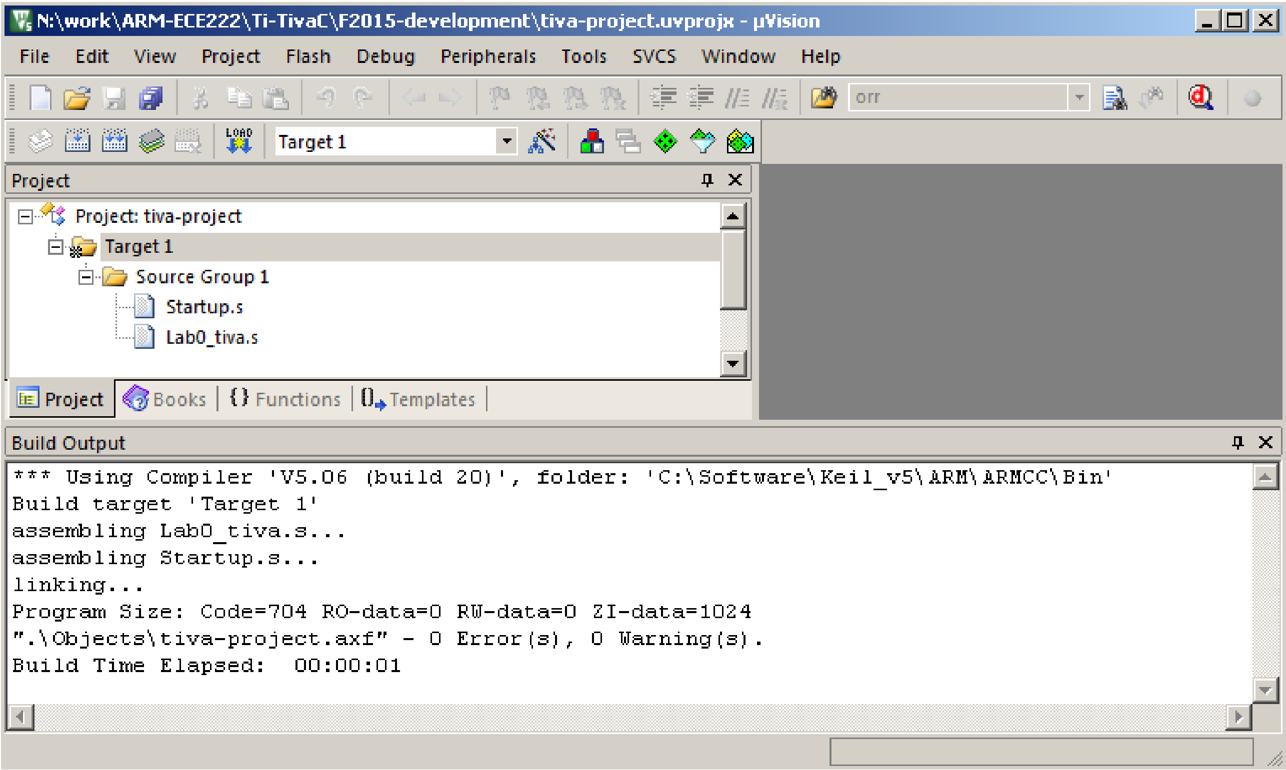


Figure 1.3 – Building the target [5]

9 The next step is to download the program into the Tiva. Click on the **Flash** drop-down menu and select **Download**. To run the code, press the **Reset** button on the board.
   a. Ensure that the power switch in the corner of the Tiva board is set to “Debug”
   b. If, when downloading the program to the device, you receive a **SWD Communication Failure** message push the reset button on the board and try again. Lastly be sure both USB cables are connected to the PC.
   c. To eliminate the need to press the Reset button after every download, right click on your target, **Target 1**, and select **Options for Target ‘Target 1’** and then click the **Utilities** tab. Next click on **Settings**, select the **Use Target Driver for Flash Programing**, click on **Flash Download** tab, and then ensure the checkbox for **Reset and Run** under the **Download Function** section is selected.

Using the simulator

The µVision software comes with a powerful Simulator and it can be used to test code when you do not have access to an ARM board. Here is how to switch between debugging on a physical board and the simulator:

1 Make sure that you are not in the Debug mode. If in Debug mode, simply exit from it by clicking on the Debug button.
2 Right-click on the **TARGET 1** and choose the **Options for Target ‘Target 1’** and then click on the **Debug** tab. You should see Figure 1.4
3 You have the option to choose between the Simulator or the Tiva board. If you click on **Use**

Simulator on the left pane, then you are no longer using the actual board. But if you choose **Stellaris ICDI DEBUGGER** on the right pane, you will need the Tiva board connected to the computer you are working on.

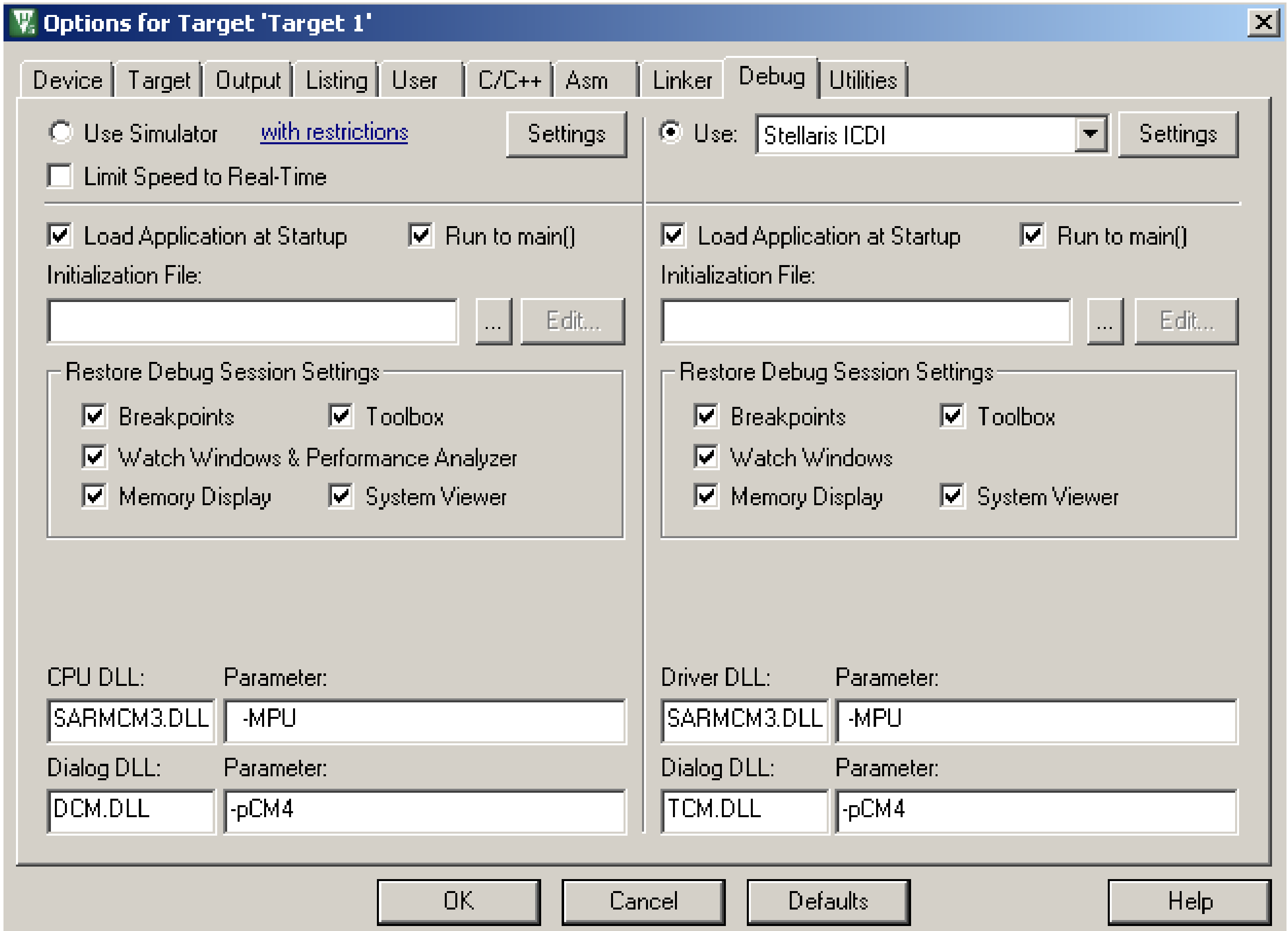


Figure 1.4 – Simulator versus Debugger on the Tiva board [5]

Debugging assembly language code

As you may have noticed, there is no visual difference on the board when the code is running. So, how can we make sure that the code is generating the correct results? This is done by running the code step-by-step and checking the content of registers.

This is called Debug mode. It is a very powerful and useful mode when you want to find a bug in your code. Your code must generate no errors when assembled before you activate the Debug mode.

The Debug mode can be used both with the Simulator or the board itself. When debugging using the board, every instruction will be executed on the Tiva, and the results are communicated over the 'Stellaris ICDI Debugger'. Be sure that your workstation is physically connected to the board via the USB cable, otherwise communication will not be possible.

If the Simulator is chosen, then the board is not used at all during the debug mode.

Follow these instructions in order to step through (debug) your code:

1. Make sure you are using the board and not the simulator for the following steps. (see section **Using the Simulator**)
2. Choose **Start/Stop Debug Session** from the **Debug** drop-down menu.
3. Click **OK** when presented with the message about being in “Evaluation Mode.” Your screen should now resemble Figure 1.5
4. Make note of the following important buttons in the graphical user interface (GUI):

From left to right: **Reset**, **Run**, **Stop**, **Step**, **Step Over**, **Step Out**, **Run to Cursor Line**, **Show Next Statement**, **Command Window**, **Disassembly Window**, **Symbol Window**, **Registers Window**, **Call Stack Window**, **Watch Windows**, **Memory Windows**, **Serial Windows**, **Analysis Windows**, **Trace Windows**, **System Viewer Windows**, **Toolbox**, **Debug Restore Views**

5. Click on the **Reset** button. The arrow should point to the line **B.W Start** ONLY if using the Simulator – but not when using the hardware debugger with flash memory.
6. Set a breakpoint on the first line of code **“MOV R0, #0x5678”** by selecting the line and hitting **F9** or by right clicking on the line. The red circle on the left indicates that a breakpoint is set. Then click on the **Run** button, or use **F5**, to run the program to the breakpoint.
7. Click on the **Step** button or **F11**. The yellow arrow moves down by one line. This means that the first line of code was run and you are now about to run the next line.
8. Click on the **Step** button, or push **F11** button on keyboard, several times until you reach the last line of code “**loop B loop”** before the **END**. In each step look at the register values to make sure that the program is working properly

Lab report

Although there is no mark assigned to this lab, attendance is mandatory and will be checked. You must complete Lab 0 before starting Lab 1.

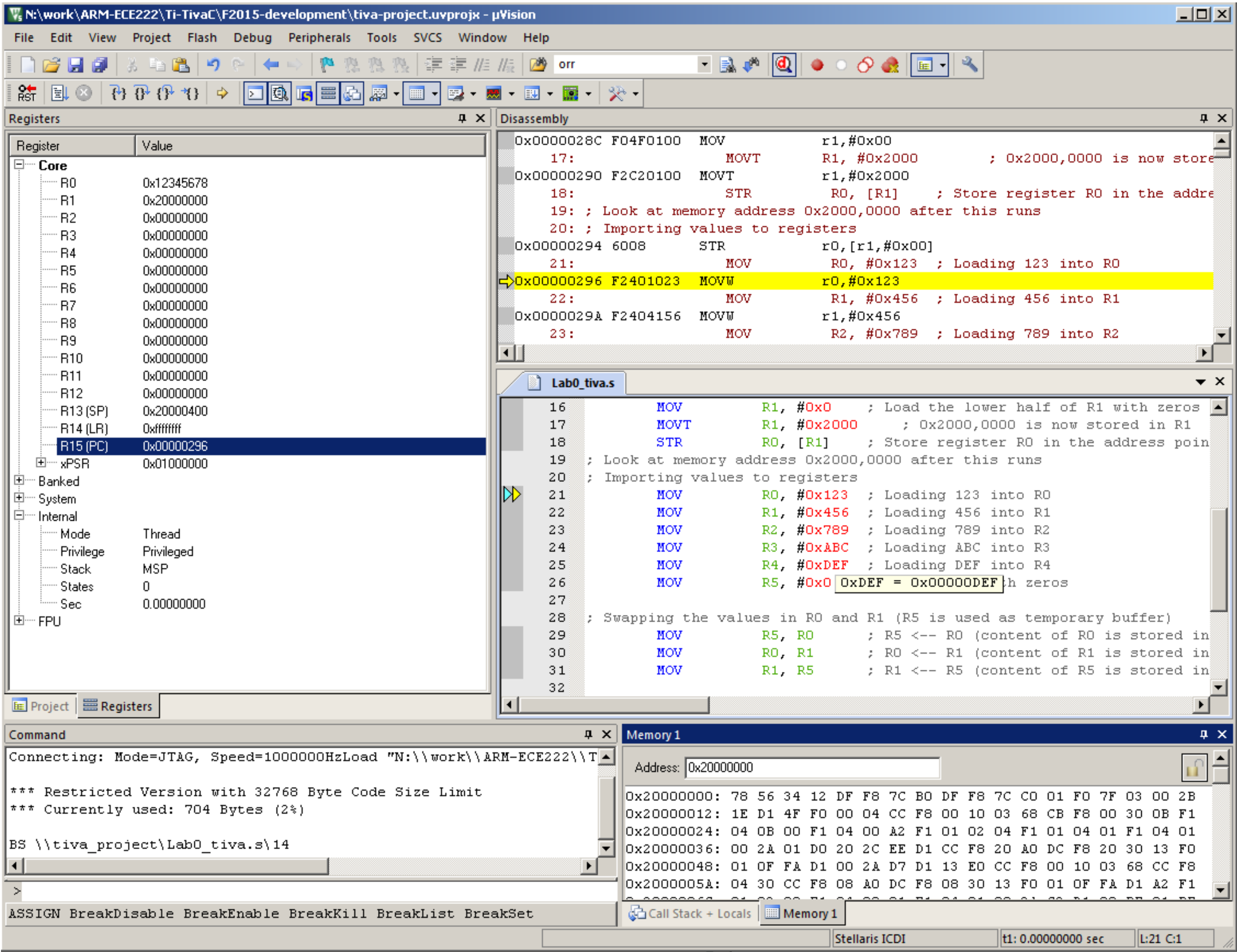


Figure 1.5 – Debug session in µVision software [5]

The assembly language code

```
;*------------------------------------------------------------------------
;* Name:    Lab_0_program.s
;* Purpose: Teaching students how to work with the uVision software
;* Author:       Rasoul Keshavarzi
;*------------------------------------------------------------------------
       AREA    |.text|, CODE, READONLY, ALIGN=2
       THUMB
       EXPORT  Start                 ; The start of the program must be called "Start" as it matches the
Startup.s file
Start

; Store 0x1234,5678 into memory address 0x2000,0000 in order to see how the little-endian
;  writes data into memory
                  MOV            R0, #0x5678    ; Load the lower half of R0 and clear the upper half
                  MOVT           R0, #0x1234    ; Load the upper half of R0
                  MOV            R1, #0x0       ; Load the lower half of R1 with zeros
                  MOVT           R1, #0x2000    ; 0x2000,0000 is now stored in R1
                  STR            R0, [R1]       ; Store register R0 in the address pointed to by R1
(0x2000,0000)
; Look at memory address 0x2000,0000 before and after this runs
; Importing values to registers
                  MOV            R0, #0x123     ; Loading 123 into R0
                  MOV            R1, #0x456     ; Loading 456 into R1
                  MOV            R2, #0x789     ; Loading 789 into R2
                  MOV            R3, #0xABC     ; Loading ABC into R3
                  MOV            R4, #0xDEF     ; Loading DEF into R4
                  MOV            R5, #0x0       ; Loading R5 with zeros

; Swapping the values in R0 and R1 (R5 is used as temporary buffer)
                  MOV            R5, R0         ; R5 <-- R0 (content of R0 is stored in R5)
                  MOV            R0, R1         ; R0 <-- R1 (content of R1 is stored in R0)
                  MOV            R1, R5         ; R1 <-- R5 (content of R5 is stored in R1)

; Adding five values together R5 <-- R0+R1+R2+R3+R4
                  ADD            R5, R0, R1     ; R5 <-- R0 + R1
                  ADD            R5, R2         ; R5 <-- R5 + R2
                  ADD            R5, R3         ; R5 <-- R5 + R3
                  ADD            R5, R4         ; R5 <-- R5 + R4

LOOP              B              LOOP           ; Branch back to this line – an infinite loop

                  END
```

Lab-1: Flashing LED

Objective

The objective of this lab is to complete, assemble and download a simple assembly language program. Here is a short list of what you will do in this session:

- Write some THUMB assembly language instructions
- Use different memory addressing modes
- Test and debug the code on the MCU board

You will flash an LED (Light Emitting Diode) at an approximate 1 Hz frequency.

Background

The Tiva belongs to the Cortex-M4 family of microprocessors which uses the THUMB instruction set. Thumb is a subset of the ARM instruction set.

Conditional instructions are possible via the PSR (Program Status Register). The register can be viewed by expanding xPSR in the Register Window of Keil MDK. There is only one bit, Z, which will be used in this course. If the result of an operation (memory read, test, compare, math, logic), which sets the status bits, is zero this bit will be set to 1. Appendix B details which status bits can be set by which instruction. The bits, https://en.wikipedia.org/wiki/Status_register in brief are:

Z – Zero – was the result zero

N – Negative – the highest bit which may indicate the sign of the number

C – Carry – was a carry (overflow) generated by an operation

V – Overflow (only used for signed math)

Code can be conditionally executed by using an instruction which updates the Z flag (ie compare to 0, ADDS, MOVS) and then branching (see Appendix B) based upon the result of the test. BNE and BEQ are the only branches you will use. BNE will branch if the Z flag is 0 – if Z indicates that the instruction updating the Z flag was Not Equal to zero. BEQ will branch if Z is set or if the instruction set the Z flag and indicates that EQual to zero.

In order to flash an LED, one needs to know how the TM4C123GH6PM microprocessor is connected to the LEDs – the pin configuration and interfacing. A subroutine, PortF_Init, is provided and it details what is involved with setting up a port. A port may be analog or digital and each pin can have one of up to seven different functions. Interrupts and pull up/down resisters may also be enabled.

After the PortF_Init subroutine has run the LEDs on the Tiva board may be turned on and off by turning on or off bits 1, 2 and 3 at address 0x4002 5038. This can be done with the Memory window in Keil MDK 5 while in debug mode.

Pre-lab

Before the lab session, look at the THUMB instruction set in Appendix B. The TM4C123GH6PM is a Cortex-M4 ARM CPU using the THUMB instruction set.

In order to see a flashing LED, implement a delay between the LED “on” and “off” states. Think about implementing a delay in assembly language.

Hint: Increment or decrement a register in a loop until it reaches a certain value.

There is no deliverable as pre-lab for this lab.

In-lab procedure

Complete the given code that is given at the end of Lab-1 manual. Feel free to change any part of the code if you wish.

Please note that the following line in the given program is for the short flowchart where the LED is toggled as opposed to turning on and off.

```
STR         R0, [R1]         ; write data to PortF
```

Try to make a connection between the given code and the flowcharts, and then complete the provided code. Add about five lines of code in the main loop that causes the LED to flash.

- Create a new folder (like N:\ECE222\Lab1) and project as was done in Lab-0
- Ensure that you call the PortF_Init subroutine to turn off all LEDs

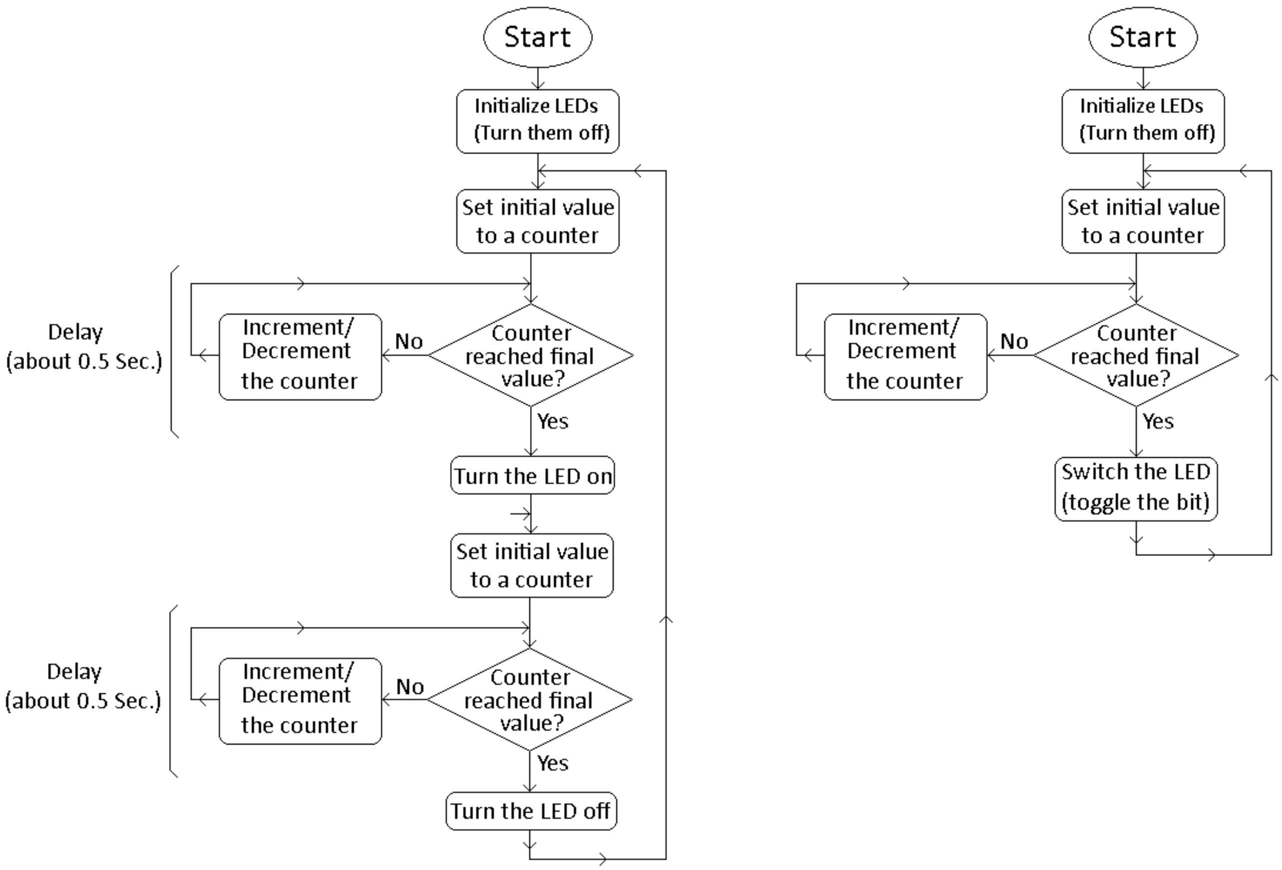


Figure 1.1 – Flowcharts for flashing LED

- Then implement the flashing LED code using an infinite loop which toggles one of the red, green or blue LEDs. Figure 1.1 shows the two different approaches. The shorter flowchart leads to smaller code size and it is more efficient. Your code, when demonstrated for marking, must be using the short flowchart. **But it is strongly recommended that you implement the longer flowchart as the first step.** Once the longer flowchart is working, changing your code to implement the short flowchart.
- Don't forget to insert a 500ms delay in the loop; otherwise the LED blinks too fast to see.
- Assemble the code, download it to the board, and debug it if necessary.

Flash versus RAM memory

Flash memory is non-volatile, meaning that it retains its contents without power and because of this it has become the default. SRAM (Static Random Access Memory) is volatile and the contents written to a RAM are lost as soon as power is lost but the memory never wears out.

Flash memory starts to fail after a few thousands 'Write' operations. At some point downloading will generate memory errors indicating that the Flash memory has failed. This started after three years with the Keil ARM boards.

Coding Goals

The goal is to get the LED flashing at a frequency close to 1 Hz. An accurate period of 1,000 ms or 1,000,000 µs is NOT the goal of this lab – something like 20% accuracy is good enough.

All code should be well commented and all documentation must be within the program. The number of lines of code does NOT affect your mark.

Lab report

Submit a report electronically to LEARN. To understand the deliverables look at the Lab1 Submission form.

Hand assemble, using Appendix G, the instruction below (if you want a challenge select another supported instruction) using the tables in Appendix G. Note that only 8-bit MOV instructions are supported. NOTE: Appendix G is for an OBSOLETE ARM processor and the instructions are not compatible with the ARM processor we are using.

ADD R4, R4, R2.

The assembly instruction should appear as comments at the end of your code.

A snippet from the assembly language code:

```
Start
   BL  PortF_Init                ; initialize input and output pins of Port F
loop
          MOV R0, #RED                              ; load in the value to turn the RED led ON
          LDR R1, =GPIO_PORTF_DATA_R ; pointer to Port F data register
          STR R0, [R1]             ; write data to Port F to turn lights on and off

          LDR R0, =SOMEDELAY          ; R0 = a value to get about a second delay
delay
          SUBS R0, R0, #1              ; R0 = R0 - 1 (count = count - 1) and set N, Z, C status bits
          ; Note: For SUBs the "s" suffix means to set the status bits, without this the loops would not exit

;  five or more lines of code are needed to complete the program

; To turn off the LED(s) simply write 0 to the Port F data register
; Note the program is shorter using a toggle function
; If a toggle function is not used then more than 5 lines of code are required.
; Note: a dedicated register to hold GPIO_PORTF_DATA_R can be used to save re-initializing

          MOV R0, #0                                ; load in the value to turn the RED led OFF
          LDR R1, =GPIO_PORTF_DATA_R ; pointer to Port F data register
          STR R0, [R1]             ; write data to Port F

; watch out - the LED must be turned on - then a delay used and then it must be turned off
;   and another delay used.
; If the delay is too short the LED will look as if it is on constantly and if the delay is too long then the user
might have to wait hours for it to change state

          B          loop
   END                         ; end of file
```

Lab-1 Submission form

Class: 001 □ 002 □	201 □ 204 □	202 □ 205 □	203 □ 206 □	Demo date:

Submission Statement: We (I) are (am) submitting this report for grading in ECE 222. We (I) certify that this report (including any code, descriptions, flowcharts, etc., that are part of the submission) were written by us (me) and have received no prior academic credit at this university or any other institution. **The penalty for copying or plagiarism will be a grade of zero (0).**

Member 1	**Member 2**
Name:	Name:
UW-ID (**NOT** student #)	UW-ID (**NOT** student #)
Signature:	Signature:

Note: Reports submitted without a signed submission statement will receive a grade of zero (0).

		Weight	**Grade**	**Comment**
Part-I	Pre-lab	0	--	
Part-II Lab-demo	Lab completion (short flowchart)	35		
	Questions	35		
Part-III Lab report	Hand Assembly	10		
	Code quality	10		
	Code comments	10		
	Total	**100**		

Lab-2: Subroutines and parameter passing

Objective

In structured programming, big tasks are broken into small routines. A short program is written for each routine. The main program calls these short subroutines.

In most cases when a subroutine is called, some information, parameters, must be communicated between the main program and the subroutine. This is called parameter passing.

In this lab, you will use subroutines and parameter passing by implementing a Morse code system.

What you do

In this lab you will turn one LED into a Morse code transmitter. You will cause one LED to blink in Morse code for a five character word. The LED must be turned on and off with specified time delays until all characters are communicated.

Pre-lab

Think about implementing Lab-1 code using subroutines. Write a subroutine called LED_OFF that turns the red LED off, and another subroutine called LED_ON that turns the red LED on. Write a third subroutine called DELAY that takes one input parameter (register R0) and waits for R0 * 500ms before returning.

There is no deliverable as Pre-lab for this experiment.

In-lab procedure

A template code for lab-2 is available on LEARN. Add to it what you learned in lab-1. Start by initializing the on-board LEDs to off. Then additional functionalities are added to the code as shown in the flowchart depicted in figure 2.1. This is described in the following steps:

- Turn all LEDs off
- Put the initials of the two lab partners together to create a four character word (**Capital letters** only). Add a fifth character of your choice (capital) which is different from the four previous ones. Set the five characters in your program at the InputChar label.
- Write a subroutine called LED_OFF that turns the red LED off
- Write a subroutine called LED_ON that turns the red LED on
- Write a subroutine called DELAY that causes R0 * 500ms before returning to the main program. R0 is passed to subroutine from the main program.
- Write a subroutine called CHAR2MORSE that converts an ASCII code into a Morse pattern. You will use registers for parameter passing between subroutines and the main program.

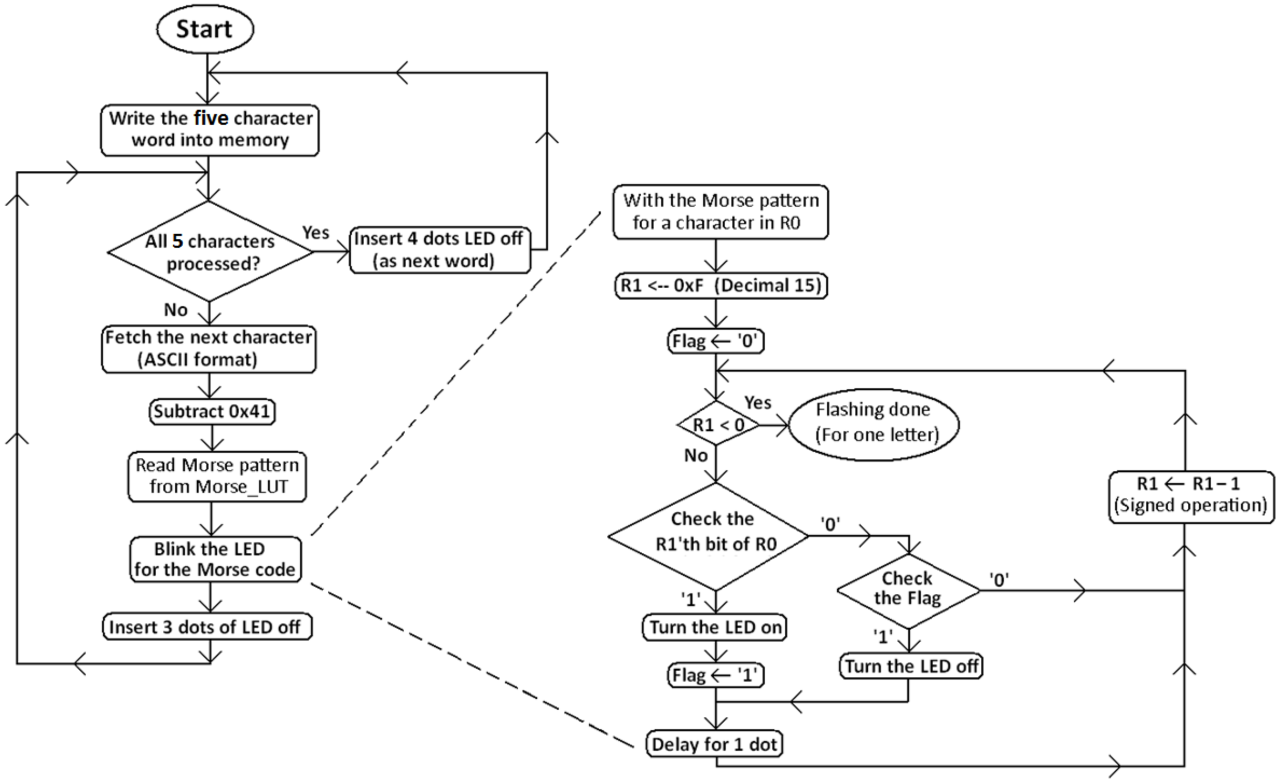


Figure 2.1 – Flowchart for the Morse code transmission using an LED

The following steps are one way to deal with each of the five characters:

- Fetch a character by reading from the memory (from the first to last). It is in ASCII format as shown in table 2.1
- Subtract 0x41 from the ASCII value to get the index for the Morse LUT (look up table)
- Read the Morse pattern from the Morse LUT (using the index)
- Blink the LED for the Morse pattern. Ignore the zeros added to the left of the Morse pattern. This step can be broken into the following sub-steps:
    a) Move the Morse code pattern for a character in register R0
    b) Load the register R1 with 0xF. It will be used to count if all 16 bits of the Morse code pattern in R0 are processed.
    c) Reset a flag, indicating if a '1' bit has been found, to zero. It can be a register, a memory address, or even a single bit.
    d) Check the value in R1. If it is a negative number, then turn off the LED and exit from the sub-steps (you have completed LED flashing for one character). Otherwise continue to the next step.
    e) Check the R1'th bit of register R0 (register R0 has 16 bits numbered from 0 to 15 decimal). If it is '1' then call the LED_ON subroutine, set the Flag variable, and go to step g
    f) Call the LED_OFF subroutine
    g) Call the DELAY subroutine if the Flag is '1'.
    h) Decrement R1 and go back to step d

- Insert long delay (equivalent to three dots) before fetching the next character
- Repeat the above steps for all characters. If the whole string has been processed, then insert another four DELAY intervals and start from the beginning again as a new word

Hint: The step (e) in the above procedure can be completed in several different ways. One option is to use the AND command to test a bit.

```
ANDS    R7, R0, R5    ; R7 <-- R0 and R5
BEQ     ZERO_R7       ; Or alternatively    BNE    NONZERO_R7
```

R5 contains a value of 0x8000 at the beginning (15'th bit set to '1' with the rest of the bits set to '0'). To check the next bit (using LSR R5, #1) the mask, R5, is shifted right. This is simpler than the sample code which checks bit 16 by masking (ANDing) with 0x10000 and shifting the data to the left for the next bit. The looping can be simplified because there are never more than 14 bits used and testing for the mask going to zero is easier than testing if the lower 16 bits of data or zero or if a counter has gone to 0.

Hint: The shift operations (LSR, LSL) shifts the number in a register by a certain number of bits to the left or right. The code template, and flowchart above, is the preferred method if one is dealing with multiple bits at a time. But the shift operations work much better when one shifts only one bit at a time. The _LAST_ bit to get shifted off (or "fall off") the register goes into the C status flag. At that point it's easy to get conditional upon the C status flag using CC (Carry Clear) or CS (Carry Set). Conditional math and logic instructions (ORREQ, ANDCS, …) allow one to write code without conditional branches and this will result in code that is often less than ½ the size and executes more than 2x faster.

Lookup Tables

Lookup tables are often used to provide some data to an assembly language. You should be careful about how to read the data and index into it. Data can be 8-bit (byte), 16-bit (half-word) or 32-bit (word). Data is stored after the program using DCB (Define Constant Byte) or DCW (Define Constant Word – where word means 16-bits).

Here is some simple code to read a character:

```
LDR  R0, =InputChar ; R0 points to the start of the string
; set R1 to 0 or else you read garbage in the next line!
LDRB R2, [R0, R1]   ; Read a character and put it into R2
                    ; R1 is an offset
```

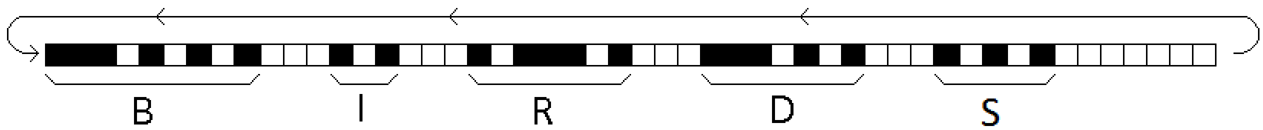
Here is some code to read from an array of 16-bit data:

```
LDR  R3, =Morse_LUT
; init. R4 to 0 or the top 16 bits of R4 will be garbage!
LDRH R4, [R3, R5]   ; Reading the Morse pattern
                         ; R5 is an offset
```

```
      some code here

      ALIGN
InputChar
      DCB   "BIRDS", 0      ; This is the five character word
                            ; to be sent on the LED using Morse

      ALIGN
Morse_LUT
      DCW   0x17, 0x1D5, 0x75D, 0x75 ; A, B, C, D
      DCW   0x1, 0x15D, 0x1DD, 0x55  ; E, F, G, H
      DCW   ...                      ; Use the template code on
                                     ; LEARN for complete list

      ALIGN
LED_ADR    EQU  0x2009C020 ; Address of the memory for the LED

      END
```

In order to work with 16-bit data the LUT for Morse code has to be type DCW, you have to increment by 2 (bytes) to move thru the data and to read the table you need to use LDRH. For Bytes use DCB and LDRB.

Example: Suppose the lab partner's initials plus a fifth letter compose the word BIRDS. Then the program should extract the letters (B I R D S) and create a Morse code pattern like this:



Please note that all five letters are considered to be one word.

Lab report

Submit a report to the electronic drop-box prepared for lab-2 on LEARN. Add enough comments so that your code is clearly understood. Examine the Lab-2 Submission form to see what you will need to deliver.

The Morse code

Table 2.1 shows corresponding Morse codes for English language alphabets.

Note that the “Morse code” pattern does not agree strictly with the “Binary Morse code value”. The first is left justified and the second is right justified.

Table 2.1 – The Morse code

Letter	**ASCII value**	Morse code	Morse code value: Binary	Dec.	Hex
A	**0x41**	■□■■■	0000,0000,0001,0111	23	0x17
B	**0x42**	■■■□■□■□■	0000,0001,1101,0101	469	0x1D5
C	**0x43**	■■■□■□■■■□■	0000,0111,0101,1101	1885	0x75D
D	**0x44**	■■■□■□■	0000,0000,0111,0101	117	0x75
E	**0x45**	■	0000,0000,0000,0001	1	0x 1
F	**0x46**	■□■□■■■□■	0000,0001,0101,1101	349	0x 15D
G	**0x47**	■■■□■■■□■	0000,0001,1101,1101	477	0x 1DD
H	**0x48**	■□■□■□■	0000,0000,0101,0101	85	0x 55
I	**0x49**	■□■	0000,0000,0000,0101	5	0x 5
J	**0x4A**	■□■■■□■■■□■■■	0001,0111,0111,0111	6007	0x 1777
K	**0x4B**	■■■□■□■■■	0000,0001,1101,0111	471	0x 1D7
L	**0x4C**	■□■■■□■□■	0000,0001,0111,0101	373	0x 175
M	**0x4D**	■■■□■■■	0000,0000,0111,0111	119	0x 77
N	**0x4E**	■■■□■	0000,0000,0001,1101	29	0x 1D
O	**0x4F**	■■■□■■■□■■■	0000,0111,0111,0111	1911	0x 777
P	**0x50**	■□■■■□■■■□■	0000,0101,1101,1101	1501	0x 5DD
Q	**0x51**	■■■□■■■□■□■■■	0001,1101,1101,0111	7639	0x 1DD7
R	**0x52**	■□■■■□■	0000,0000,0101,1101	93	0x 5D
S	**0x53**	■□■□■	0000,0000,0001,0101	21	0x 15
T	**0x54**	■■■	0000,0000,0000,0111	7	0x 7
U	**0x55**	■□■□■■■	0000,0000,0101,0111	87	0x 57
V	**0x56**	■□■□■□■■■	0000,0001,0101,0111	343	0x 157
W	**0x57**	■□■■■□■■■	0000,0001,0111,0111	375	0x 177
X	**0x58**	■■■□■□■□■■■	0000,0111,0101,0111	1879	0x 757
Y	**0x59**	■■■□■□■■■□■■■	0001,1101,0111,0111	7543	0x 1D77
Z	**0x5A**	■■■□■■■□■□■	0000,0111,0111,0101	1909	0x 775

Notes:

- A dash is equal to three dots
- The space between parts of the same letter is equal to one dot
- The space between two letters is equal to three dots
- The space between two words is equal to seven dots
- ■ A dot (LED on)
- □ LED off (same length as one dot)

Lab-2 Submission form

201 □ 204 □	202 □ 205 □	203 □ 206 □	Demo date:

Submission Statement: We (I) are (am) submitting this report for grading in ECE 222. We (I) certify that this report (including any code, descriptions, flowcharts, etc., that are part of the submission) were written by us (me) and have received no prior academic credit at this university or any other institution. **The penalty for copying or plagiarism will be a grade of zero (0).**

Member 1	**Member 2**
Name:	Name:
UW-ID (**NOT** student #)	UW-ID (**NOT** student #)
Signature:	Signature:

Note: Reports submitted without a signed submission statement will receive a grade of zero (0).

		Weight	**Grade**	**Comment**
Part-I	Pre-lab	0		
Part-II Lab-demo	Lab completion	40		
	Questions	40		
Part-III Lab report	Code quality	10		
	Code comments	10		
	Total	**100**		

Marking TA:

Lab-3: Input/Output interfacing

Objective

The objective of this lab is to learn how to use peripherals (LEDs, switch) connected to a microprocessor. The ARM CPU is connected to the outside world using Ports and in this lab you will setup, and use, Input and Output ports.

What you do

In this lab you will measure how fast a user responds (reflex-meter) to an event accurate to a 10^{th} of a millisecond. Initially all LEDs are off and after a random amount of time (between 2 to 10 seconds), one LED turns on and then the user presses the push button.

Between the two events of 'Turning the LED on' and 'Pressing the push button', a 32 bit counter is incremented every 10^{th} of a millisecond in a loop. The final value of this 32 bit number will be sent to 8 LEDs in separate bytes with a 2 second delay between them.

Background

Port F contains both the three LEDs on the motherboard as well as the two buttons. Individual bits are connected to each LED and button. The subroutine which configures port F enables pull-up resistors on the buttons so that when a button is NOT PRESSED a "1" will be read. Pressing a button will return a value of "0". The LEDs are also active low – a "0" turns them on and a "1" off.

Pre-lab

There are no deliverable for this part. It is for your information only.

Read section **10.2.1.2 Data Register Operation** to gain an understanding of how special pins on the Ports are protected via a mask for the data embedded in the Port "address". Different processors use different mechanisms and this TI processor embeds a mask into the read/write Port address to protect sensitive pins, such as the debugger, from being accident written to. This is why the provided data address for Ports A, B and E involve a calculation depending upon which exact bits contain LEDs or switches. In the provided code all accesses to the LEDs on Port F use a seemingly cryptic address which has been pre-calculated. In the template for this lab the calculation of the port A, B, E data registers are of this format: **LDR R1, = GPIO_PORTE + (PORT_x_MASK << 2)**. The base address of the port is added to the mask (containing only the bits which have LEDs and switches which we will use) shifted left 2 times.

The LEDs can be tested using the Memory explorer in the uVision debugger only AFTER they are initialized by the Port_Init subroutine. The following address and port data will toggle the LEDs:

Port A: 0x4000 4380 write 0xe0 and 00

Port B: 0x4000 50cc write 0x33 and 00

Port E: 0x4002 40c0 write 0x30 and 00

Port F: 0x4002 5038 write 0xe0 and 00 (3 LEDs on the Tiva board at bits 1, 2, 3)

In-lab procedure

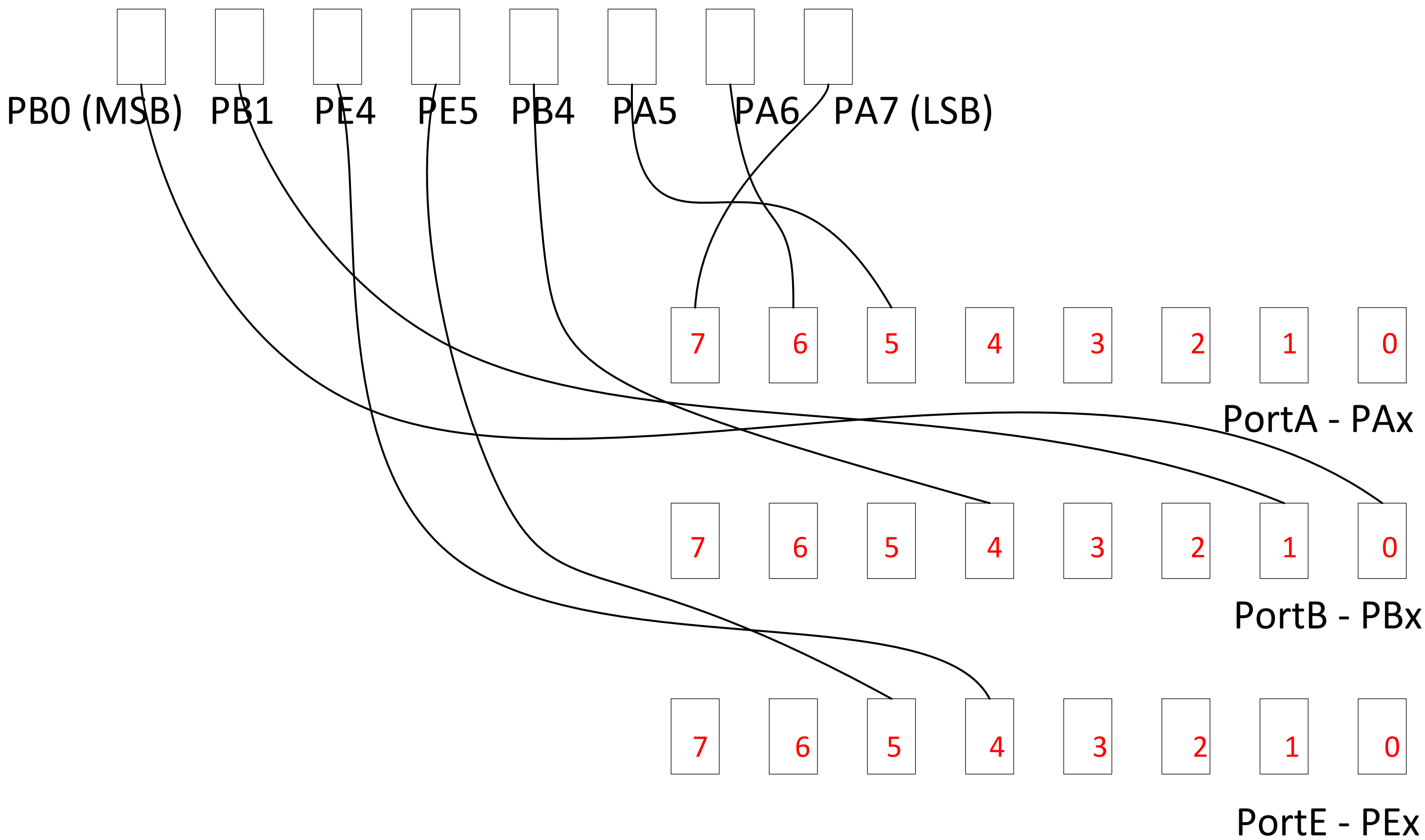
Please note that you will have to demonstrate **two parts** in Lab-3:

- A simple counter subroutine that increments from 0x00 to 0xFF, wraps to 0, and continues counting. This will prove that the bits are displayed in the correct order on the LEDs.
- The reflex-meter.

Here are the suggested steps to implement this program:

1. Write assembly language code for a subroutine which implements a 0.1 millisecond delay. To confirm the duration of 0.1 millisecond, you can do the following steps:
    a) Turn one LED on
    b) Call the subroutine in a loop for 100000 times (#100000 or #0x186A0)
    c) Turn the LED off
    d) Run the code and measure the time that the LED stays on. It must be for 100000x0.1 millisecond = 10 seconds.
2. Create a subroutine to show an 8-bit number on the 8 LEDs. The LEDs modules are active low (a 0 turns the LED on) and from LSB to MSB (right to left) are: PA7, PA6, PA5, PB4, PE5, PE4, PB1, PB0 (the 9^{th} LED PB5 is not used and the 10^{th} LED is not connected).
3. Create a simple counter to generate incrementing numbers from 0 to 255 (0xff) and write these to the 8 LEDs to verify this functionality using a 0.1 second delay between numbers.
4. To implement the reflex-meter project:
    a) The SW1 button is setup as a GPIO input port with a pull-up resistor by the Port_Init subroutine.
    b) **Optionally** keep calling the random number routine until a button press as a way to randomize the sequence of "random" numbers.
    c) Turn off all 8 LEDs
    d) Call the given pseudorandom-number generator subroutine to generate a 16 bit random number, scale it and add an offset, to result in a 2 to 10 second +/-5% delay.
    e) Call a delay function for that amount of time (in 0.1 millisecond)
    f) Turn one LED on and initialize a reaction-time register to 0
    g) Delay for 0.1ms and increment the reaction-time register by 1
    h) Check the status of the SW1 push button using polling
    i) If the button is not pressed go back to step (g)
    j) Send the first 8 bits (least significant part) of the reaction-time register to the LEDs or 7-segment display if using the UW shield
    k) Wait for 2 seconds
    l) Send the next 8 bits and wait again.
    m) Do the above steps two more times until all 32 bits are shown on the LEDs.
    n) Wait 5 seconds and go back to step (j).

LED Mapping



Optional Improvements

Here are some optional ways to improve the program if you have the time and interest:

1- Go to a website like http://www.humanbenchmark.com/tests/reactiontime/ and measure your average reaction time for comparison to your ARM program.
2- Merge the two programs into one by starting with the simple counter subroutine and when the button is pressed the program changes to the reflex-meter.
3- To make the pseudo-random generator more random keep calling it every 100uS, while in the counting subroutine. The variable time delay, while waiting for the user to press the button to exit the counter, ensures a random delay in the game.
4- To enable replaying the game; if the button is pressed during the display of the reaction time - restart the reflex-meter.
5- If the 8 highest bits of the time delay are 0x0 simply perform a 3 second delay (while optionally checking for a key press to restart the game) and then redisplay the time delay again.

Lab report

Submit your commented, well-written code for the simple counter **AND** reflex-meter to the electronic drop-box prepared for Lab-3 on LEARN.

Answer these questions and put them as comments at the end of your programs:

1- If a 32-bit register is counting the number of 10'ths of milliseconds, what is the maximum amount of time which can be encoded in 8 bits, 16-bits, 24-bits and 32-bits?

2- Considering typical human reaction time, which size would be the best for this task (8, 16, 24, or 32 bits)?

Extra Information

Random numbers with Fibonacci linear feedback shift registers at WikiPedia:
http://en.wikipedia.org/wiki/Linear_feedback_shift_register

Lab-3 Submission form

201 □	202 □	203 □	Demo date:
204 □	205 □	206 □	

Submission Statement: We (I) are (am) submitting this report for grading in ECE 222. We (I) certify that this report (including any code, descriptions, flowcharts, etc., that are part of the submission) were written by us (me) and have received no prior academic credit at this university or any other institution. **The penalty for copying or plagiarism will be a grade of zero (0).**

Member 1	Member 2
Name:	Name:
UW-ID (**NOT** student #)	UW-ID (**NOT** student #)
Signature:	Signature:

Note: Reports submitted without a signed submission statement will receive a grade of zero (0).

		Weight	Grade	Comment
Part-I	Pre-lab	0		
Part-II Lab-demo	Simple counter	15		
	Reflex-meter	25		
	Questions	40		
Part-III Lab report	Code quality	6		
	Code comments	6		
	Prove time delay meets 2 to 10 sec +/- 5% spec	6		
	Two questions	2		
Total		**100**		

Lab-4: Interrupt handling

Objective

The objective of this lab is to learn about interrupts. You will enable an interrupt source in the microprocessor, and write an interrupt service routine (ISR) that is triggered when the button SW1 is pressed. The ISR must be very short, in terms of execution time, and it returns to the [halted] main program after handling the interrupt.

What you do

The above goals are achieved by reusing code from lab-3.

The random number generator from lab-3 will be reconfigured to generate a number which gives a time delay of 5.0 to 25.0 seconds with a resolution of 0.1s.

Once the program is started a random integer is generated and stored in R6. The main program then displays this (without a decimal so that 5.6 seconds displays as 56 in binary) on the 8 LEDs. The program delays one second. Then the count in R6 is decrement by the equivalent of 1 second (10) and the new count (time left) displayed. This continues. When the count would go to 0, or less, all decrementing should stop and the R6 should stay fixed at 0 and all LEDs flash on and off (at 1 second rate is fine, but fast – 10Hz is preferred).

The interrupt service routine, triggered by SW1 being pressed, generates a new random number, scales it, and stores it in R6. The main program automatically counts this new random number down.

Pre-lab

There is no deliverable for this part. It is for your information only.

An empty interrupt routine in included in the Lab #4 code template.

An interrupt enabling subroutine is provided with the Lab 4 template code. Only three correct values have to be supplied to enable interrupts.

Information about interrupts is in the Reference [1]. The NVIC (Nested Vector Interrupt Controller) is in section 3.1.2 and controls all of the interrupts.

Individual interrupt numbers (bits) may be enabled or disabled with the EN* And DIS* registers detailed in Table 3-8 on page 134.

Table 2-9 on page 104 details the interrupt assignments – which interrupt number / bit is assigned to which Port or function. Find the Interrupt Number for Port F – it is a SINGLE BIT which must be set to 1. This is required for the INTERRUPT_EN0_OFFSET register.

Read Table 10-4 GPIO Interrupt Configuration Example

You will program the GPIOIS, GPIOIBE, GPIOEV, and GPIOIM registers to configure the type, event and mask of the interrupts for Port F with the SW1 switch. A certain order, detailed in section 10.2.2, starting on page 654, and 10.3 is necessary to prevent generating spurious interrupts and this is setup in the Lab 4 code template.

In-lab procedure

The following steps are suggested for handling the SW1 button as a source of interrupt:

- The button pin has already been configured as GPIO (default) input pin
- Enable the Port F interrupt in the correct EN* register using Table 2-9 (the offset address supplied can be verified via Table 3-8).
- Configure the interrupts for Port F as falling, single, edge using GPIOS/IBE/EV/IM registers. Only a value for the IM register needs to be obtained – it's the bit number that the switch is connected to on port F. The same value is use for the GPIO_ICR_OFFSET register.
- Complete the provided interrupt service routine, for the Port F interrupt. This must:
  - Calculate a random number, scale it and store it in register R6
  - Clear the cause of interrupt (Port F appropriate bit/pin) by writing a 1 to the correct bit / pin of the Port F ICR (Interrupt Clear Register). Do NOT disable the interrupts.

Hint: Putting a breakpoint in the ISR will reveal if the ISR is called or not, and whether it is run only once or more often. This is very helpful when debugging.

Figure 4.1 – Falling/Rising edges when the INT0 push-button is pressed/released

The following steps are suggested for implementing the procedure. Add each step and test the code.

- Revert to the Lab 3 code which used a counter to exercise all 8 LED outputs
- Modify your delay subroutine to have a resolution of 100ms (ie delay R0 * 0.1s)
- Generate a random number by calling the random number generator (code given in lab-3). Mask, scale and offset the output of the random number generator to generate a number (delay) between 5.0 and 25.0.
- Count the random number down, using 1 second delays, until the count will be 0 or less and then display 0 on the LEDs
- Add code to configure a falling edge SW1 interrupt (that's the default – you merely have to clear any possible pending interrupts if you're paranoid) and enable it in the correct EN* register. Your interrupt service routine (ISR) will then be called when SW1 is pressed. For debugging, without breakpoints, toggle the RED LED on the Tiva board when SW1 is pressed.
- Write code so that the ISR generates a new random number, scales it and stores it in R6

Lab report

Submit a report to the electronic drop-box prepared for lab-4 on LEARN. Take a look at the Lab-4 Submission form to understand what you will need to deliver.

Lab-4 Submission form

Class: 001 □ 002 □	201 □ 204 □	202 □ 205 □	203 □ 206 □	Demo date:

Submission Statement: We (I) are (am) submitting this report for grading in ECE 222. We (I) certify that this report (including any code, descriptions, flowcharts, etc., that are part of the submission) were written by us (me) and have received no prior academic credit at this university or any other institution. **The penalty for copying or plagiarism will be a grade of zero (0).**

Member 1	Member 2
Name:	Name:
UW-ID (**NOT** student #)	UW-ID (**NOT** student #)
Signature:	Signature:

Note: Reports submitted without a signed submission statement will receive a grade of zero (0).

		Weight	Grade	Comment
Part-I	Pre-lab	0		
Part-II Lab-demo	Lab completion	40		
	Questions	40		
Part-III Lab report	Code quality	10		
	Code comments	10		
Late show up for demo		**-10**		
Total		**100**		

Appendix A: The TM4C123x microprocessor

Figure A.1 shows block diagram of a MCU – a CPU integrated with I/O. Detailed information for the Tiva can be found in the document **Tiva TM4C123GH6PM Microcontroller** [1].

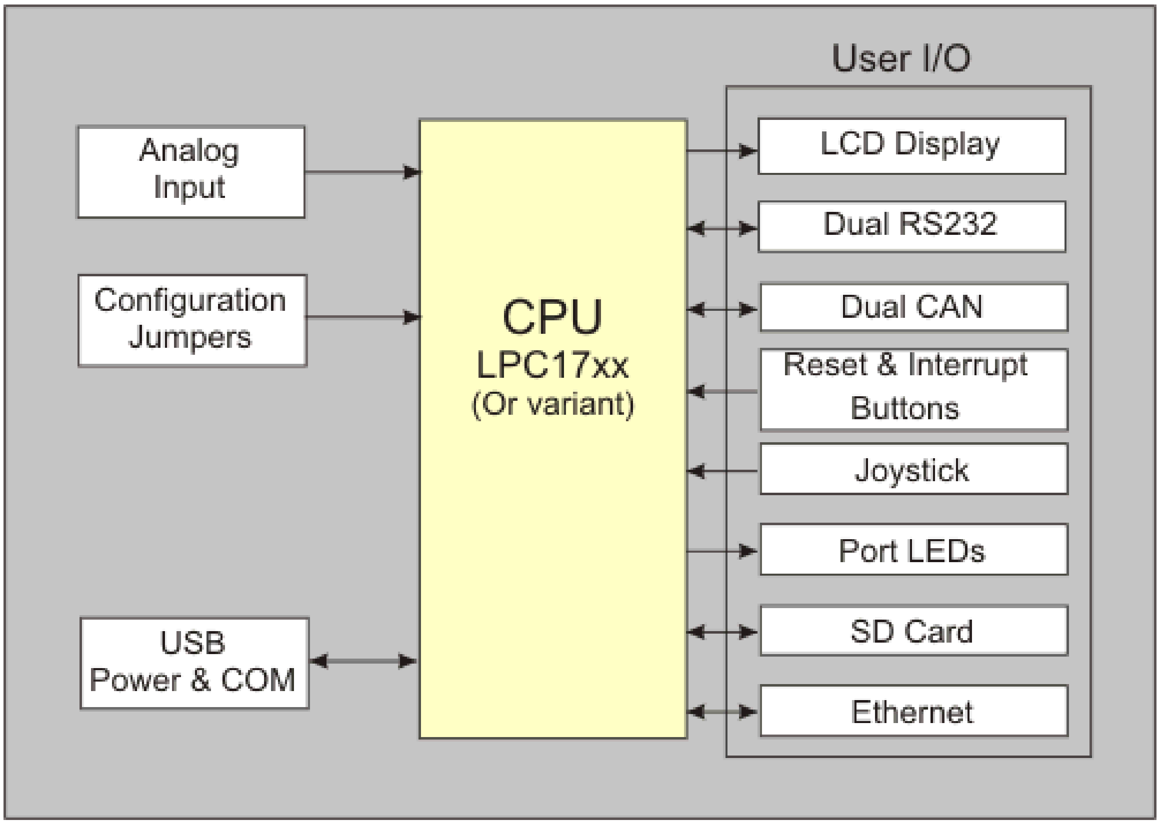


Figure A.1 – Block diagram of LPC1768 [1]

Figure A.2 shows a simplified block diagram of the LPC1768 microprocessor.

As you can see there is no memory block in the above figure. This is because all volatile (RAM) and non-volatile (Flash) memories are on-chip.

Figure A.3 shows some details of the CPU and buses.

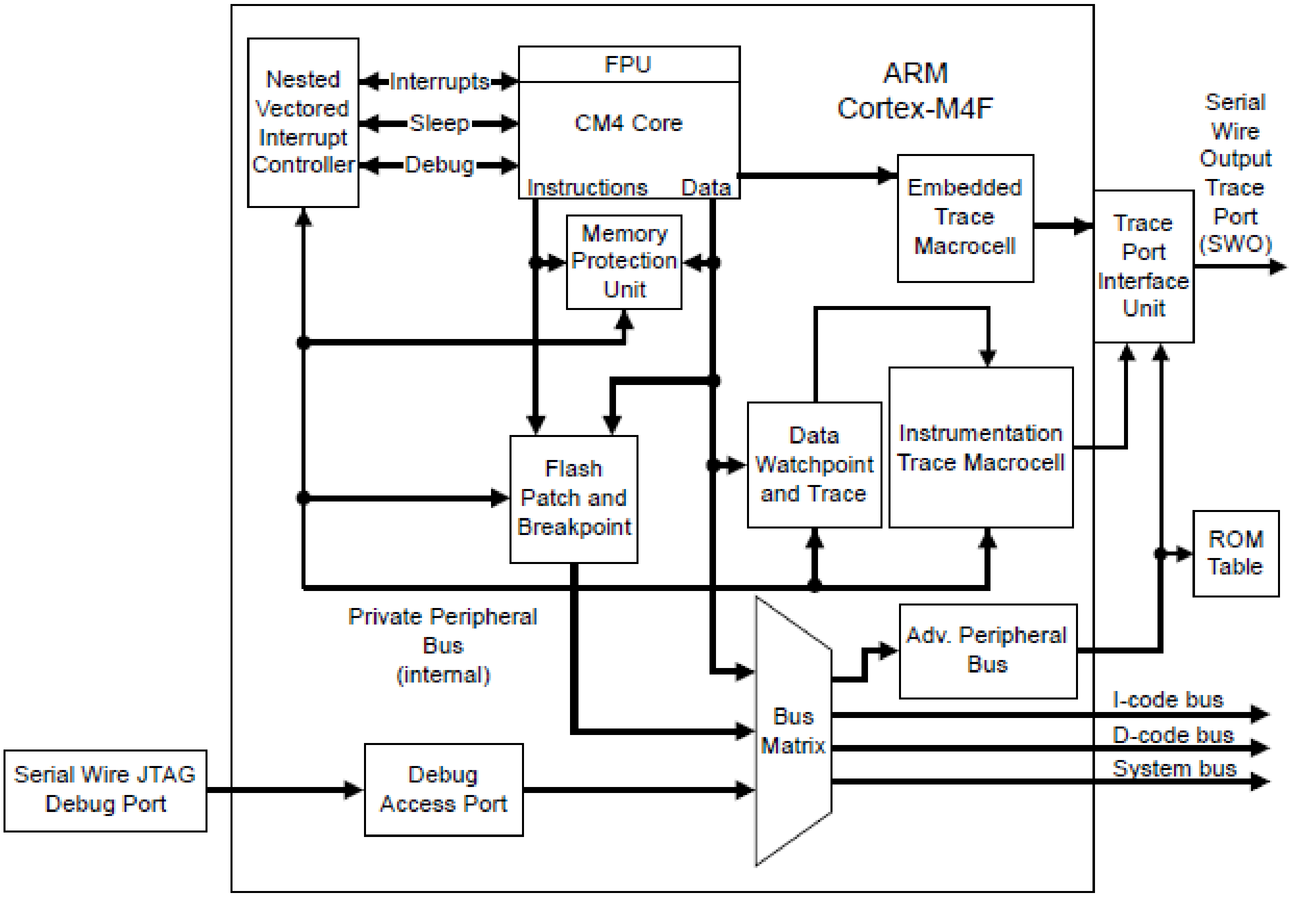


Figure A.2 – Simplified block diagram of TM4C123x CPU [1]

Note that in Figures A.3 and A.4 the ports in the TM4C123GH6PM can be accessed via two buses – the Advanced High Performance Bus (AHB) and the Advanced Peripheral Bus (APB). We will use the APB as that is the default.

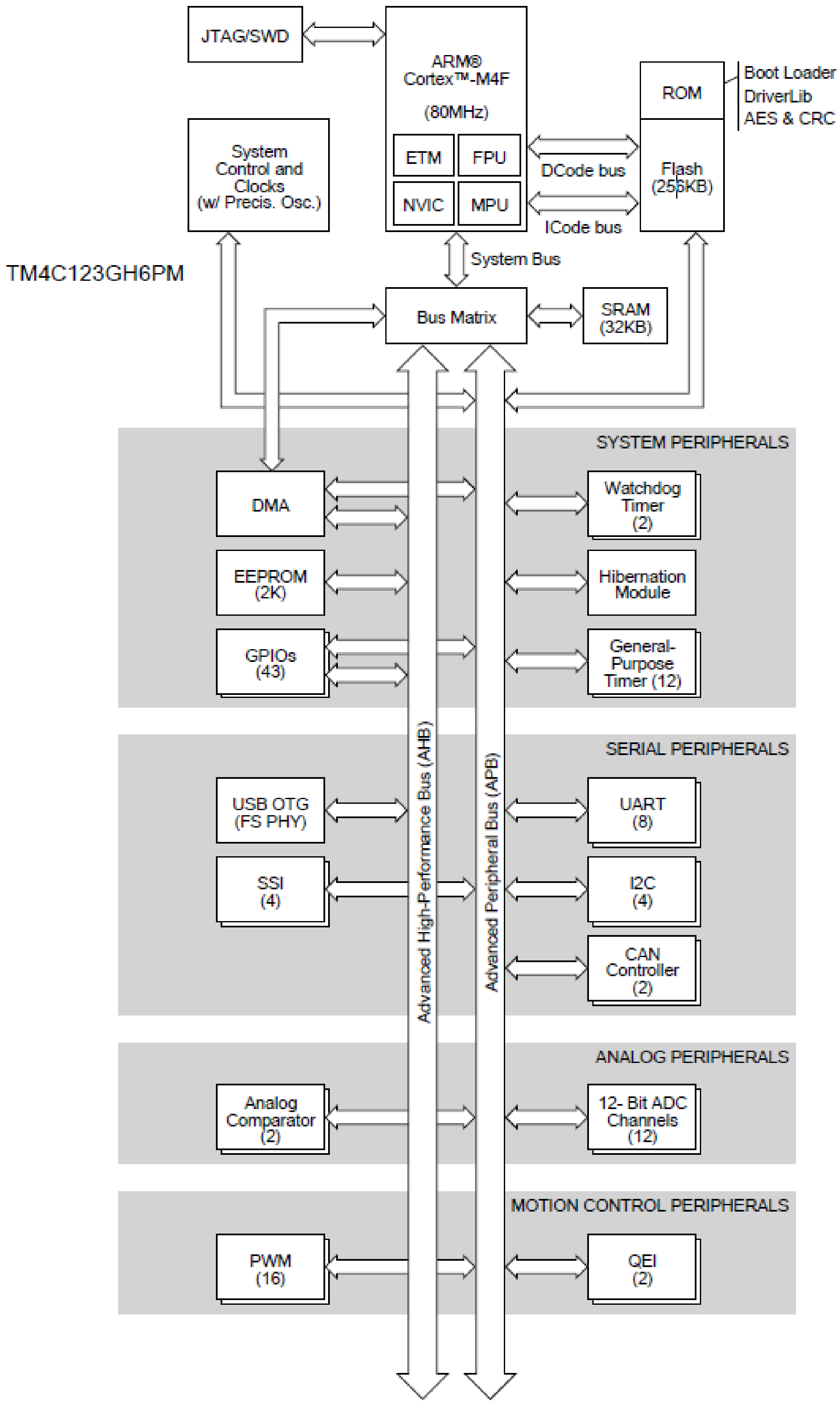


Figure A.3 – TM4C123GH6PM block diagram, CPU, and buses [1]

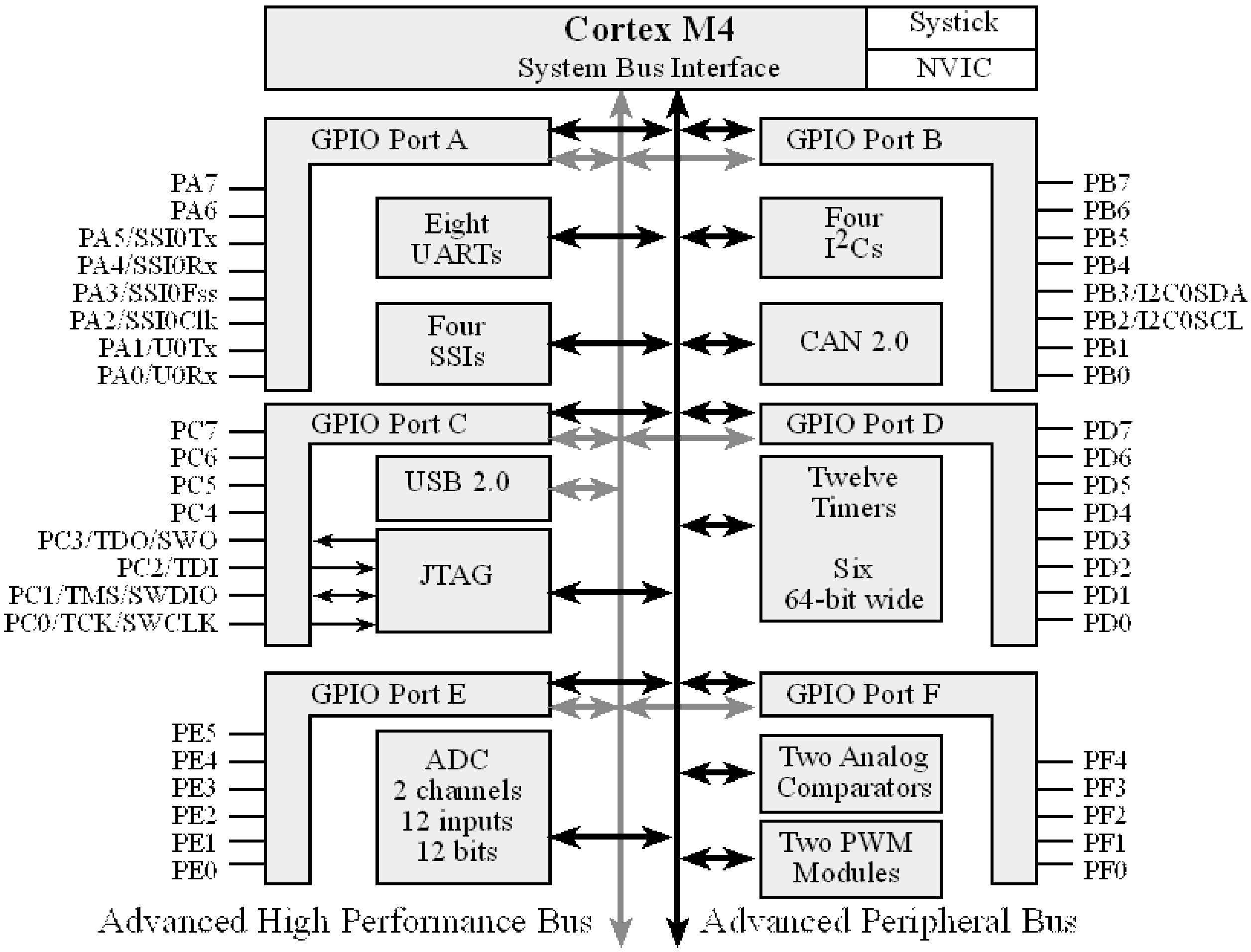


Figure A.4 - GPIO Ports for the TM4C123GH6PM Microcontroller

Appendix B: Instruction set summary

The processor implements a version of the Thumb instruction set. Table B.1 lists the supported instructions [7].

Conditional branching is done with the conditional branch instructions. An "L" can be added to the branches (BLNE, BLHI, …) to save the return address in the Link Register. Some conditional branches are:

Flag	Flag Set	Flag Clear
Z	BEQ – Equal to Zero	BNE – Not Equal to Zero
C	BCS – Carry Set	BCC – Carry Clear
N	BMI – Minus – result negative	BPL – Plus – result positive or zero
N,V	BLO – Lower (unsigned comparison)	BHS – Higher or Same (unsigned comp.)
C,Z	BLS – Lower or Same (unsigned comp.)	BHI – Higher (unsigned comp.)

Avoid using BVC, BVS, BGT, BGE, BLT, BLE as they are for signed number comparisons.

Note: In Table B.1:

- angle brackets, <>, enclose alternative forms of the operand
- an “S” suffix sets condition codes (N,V,Z) for math and logic operations
- braces, {}, enclose optional operands
- the Operands column is not exhaustive
- Op2 is a flexible second operand that can be either a register or a constant
- most instructions can use an optional condition code suffix such as S (set condition codes), H (half-word size), B (byte size), T (register top half).

For more information on the instructions and operands, see the instruction descriptions.

Table B.1. Cortex-M3 instructions [7]

Mnemonic	Operands	Brief description	Flags	Also See
ADC, ADCS	{Rd,} Rn, Op2	Add with Carry	N,Z,C,V	*ADD, ADC, SUB, SBC, and RSB*
ADD, ADDS	{Rd,} Rn, Op2	Add	N,Z,C,V	*ADD, ADC, SUB, SBC, and RSB*
ADD, ADDW	{Rd,} Rn, #imm12	Add	N,Z,C,V	*ADD, ADC, SUB, SBC, and RSB*
ADR	Rd, label	Load PC-relative Address	-	*ADR*
AND, ANDS	{Rd,} Rn, Op2	Logical AND	N,Z,C	*AND, ORR, EOR, BIC, and ORN*
ASR, ASRS	Rd, Rm, <Rs|#n>	Arithmetic Shift Right	N,Z,C	*ASR, LSL, LSR, ROR, and RRX*
B	label	Branch Always	-	*B, BL, BX, and BLX*
BFC	Rd, #lsb, #width	Bit Field Clear	-	*BFC and BFI*
BFI	Rd, Rn, #lsb, #width	Bit Field Insert	-	*BFC and BFI*
BIC, BICS	{Rd,} Rn, Op2	Bit Clear	N,Z,C	*AND, ORR, EOR, BIC, and ORN*
BL	label	Branch with Link	-	*B, BL, BX, and BLX*
BLX	Rm	Branch indirect with Link	-	*B, BL, BrX, and BLX*
BX	Rm	Branch indirect	-	*B, BL, BX, and BLX*

Mnemonic	Operands	Brief description	Flags	Also See
CBNZ	Rn, label	Compare and Branch (forward ONLY) if Non Zero	-	*CBZ and CBNZ*
CBZ	Rn, label	Compare and Branch (forward ONLY) if Zero	-	*CBZ and CBNZ*
CLZ	Rd, Rm	Count Leading Zeros	-	*CLZ*
CMN	Rn, Op2	Compare Negative	N,Z,C,V	*CMP and CMN*
CMP	Rn, Op2	Compare	N,Z,C,V	*CMP and CMN*
EOR, EORS	{Rd,} Rn, Op2	Exclusive OR	N,Z,C	*AND, ORR, EOR, BIC, and ORN*
LDM	Rn{!}, reglist	Load Multiple registers, increment after	-	*LDM and STM*
LDMDB, LDMEA	Rn{!}, reglist	Load Multiple registers, decrement before	-	*LDM rand STM*
LDMFD, LDMIA	Rn{!}, reglist	Load Multiple registers, increment after	-	*LDM and STM*
LDR	Rt, [Rn, #offset]	Load Register with word	-	*Memory access instructions*
LDRB, LDRBT	Rt, [Rn, #offset]	Load Register with byte	-	*Memory access instructions*
LDRD	Rt, Rt2, [Rn, #offset]	Load Register with two bytes	-	*LDR and STR, immediate offset*
LDREX	Rt, [Rn, #offset]	Load Register Exclusive	-	*LDREX and STREX*
LDREXB	Rt, [Rn]	Load Register Exclusive with Byte	-	*LDREX and STREX*
LDREXH	Rt, [Rn]	Load Register Exclusive with Halfword	-	*LDREX and STREX*
LDRH, LDRHT	Rt, [Rn, #offset]	Load Register with Halfword	-	*Memory access instructions*
LDRSB, LDRSBT	Rt, [Rn, #offset]	Load Register with Signed Byte	-	*Memory access instructions*
LDRSH, LDRSHT	Rt, [Rn, #offset]	Load Register with Signed Halfword	-	*Memory access instructions*

Mnemonic	Operands	Brief description	Flags	Also See
LDRT	Rt, [Rn, #offset]	Load Register with word	-	*Memory access instructions*
LSL, LSLS	Rd, Rm, <Rs|#n>	Logical Shift Left	N,Z,C	*ASR, LSL, LSR, ROR, and RRX*
LSR, LSRS	Rd, Rm, <Rs|#n>	Logical Shift Right	N,Z,C	*ASR, LSL, LSR, ROR, and RRX*
MLA	Rd, Rn, Rm, Ra	Multiply with Accumulate, 32-bit result	-	*MUL, MLA, and MLS*
MLS	Rd, Rn, Rm, Ra	Multiply and Subtract, 32-bit result	-	*MUL, MLA, and MLS*
MOV, MOVS	Rd, Op2	Move	N,Z,C	*MOV and MVN*
MOVT	Rd, #imm16	Move Top	-	*MOVT*
MOVW, MOV	Rd, #imm16	Move 16-bit constant	N,Z,C	*MOV and MVN*
MUL, MULS	{Rd,} Rn, Rm	Multiply, 32-bit result	N,Z	*MUL, MLA, and MLS*
MVN, MVNS	Rd, Op2	Move NOT	N,Z,C	*MOV and MVN*
NOP	-	No Operation	-	*NOP*
ORN, ORNS	{Rd,} Rn, Op2	Logical OR NOT	N,Z,C	*AND, ORR, EOR, BIC, and ORN*
ORR, ORRS	{Rd,} Rn, Op2	Logical OR	N,Z,C	*AND, ORR, EOR, BIC, and ORN*
POP	reglist	Pop registers from stack	-	*PUSH and POP*
PUSH	reglist	Push registers onto stack	-	*PUSH and POP*
RBIT	Rd, Rn	Reverse Bits	-	*REV, REV16, REVSH, and RBIT*
REV	Rd, Rn	Reverse byte order in a word	-	*REV, REV16, REVSH, and RBIT*
REV16	Rd, Rn	Reverse byte order in each halfword	-	*REV, REV16, REVSH, and RBIT*
REVSH	Rd, Rn	Reverse byte order in bottom halfword and sign extend	-	*REV, REV16, REVSH, and RBIT*

Mnemonic	Operands	Brief description	Flags	Also See
ROR, RORS	Rd, Rm, <Rs|#n>	Rotate Right	N,Z,C	*ASR, LSL, LSR, ROR, and RRX*
RRX, RRXS	Rd, Rm	Rotate Right with Extend	N,Z,C	*ASR, LSL, LSR, ROR, and RRX*
RSB, RSBS	{Rd,} Rn, Op2	Reverse Subtract	N,Z,C,V	*ADD, ADC, SUB, SBC, and RSB*
SBC, SBCS	{Rd,} Rn, Op2	Subtract with Carry	N,Z,C,V	*ADD, ADC, SUB, SBC, and RSB*
SBFX	Rd, Rn, #lsb, #width	Signed Bit Field Extract	-	*SBFX and rUBFX*
SDIV	{Rd,} Rn, Rm	Signed Divide	-	*SDIV and UDIV*
SMLAL	RdLo, RdHi, Rn, Rm	Signed Multiply with Accumulate (32 x 32 + 64), 64-bit result	-	*UMULL, UMLAL, SMULL, and SMLAL*
SMULL	RdLo, RdHi, Rn, Rm	Signed Multiply (32 x 32), 64-bit result	-	*UMULL, UMLAL, SMULL, and SMLAL*
SSAT	Rd, #n, Rm {,shift #s}	Signed Saturate	Q	*SSAT and USAT*
STM	Rn{!}, reglist	Store Multiple registers, increment after	-	*LDM and STM*
STMDB, STMEA	Rn{!}, reglist	Store Multiple registers, decrement before	-	*LDM and STM*
STMFD, STMIA	Rn{!}, reglist	Store Multiple registers, increment after	-	*LDM and STM*
STR	Rt, [Rn, #offset]	Store Register word	-	*Memory access instructions*
STRB, STRBT	Rt, [Rn, #offset]	Store Register byte	-	*Memory access instructions*
STRD	Rt, Rt2, [Rn, #offset]	Store Register two words	-	*LDR and STR, immediate offset*
STREX	Rd, Rt, [Rn, #offset]	Store Register Exclusive	-	*LDREX and STREX*
STREXB	Rd, Rt, [Rn]	Store Register Exclusive Byte	-	*LDREX and STrREX*
STREXH	Rd, Rt, [Rn]	Store Register Exclusive Halfword	-	*LDREX and STREX*

Mnemonic	Operands	Brief description	Flags	Also See
STRH, STRHT	Rt, [Rn, #offset]	Store Register Halfword	-	*Memory access instructions*
STRT	Rt, [Rn, #offset]	Store Register word	-	*Memory access instructions*
SUB, SUBS	{Rd,} Rn, Op2	Subtract	N,Z,C,V	*ADD, ADC, SUB, SBC, and RSB*
SUB, SUBW	{Rd,} Rn, #imm12	Subtract	N,Z,C,V	*ADD, ADC, SUB, SBC, and RSB*
SXTB	{Rd,} Rm {,ROR #n}	Sign extend a byte	-	*SXT and UXT*
SXTH	{Rd,} Rm {,ROR #n}	Sign extend a halfword	-	*SXT and UXT*
TEQ	Rn, Op2	Test Equivalence	N,Z,C	*TST and TEQ*
TST	Rn, Op2	Test	N,Z,C	*TST and TEQ*
UBFX	Rd, Rn, #lsb, #width	Unsigned Bit Field Extract	-	*SBFX and UBFX*
UDIV	{Rd,} Rn, Rm	Unsigned Divide	-	*SDIV and UDIV*
UMLAL	RdLo, RdHi, Rn, Rm	Unsigned Multiply with Accumulate (32 x 32 + 64), 64-bit result	-	*UMULL, UMLAL, SMULL, and SMLAL*
UMULL	RdLo, RdHi, Rn, Rm	Unsigned Multiply (32 x 32), 64-bit result	-	*UMULL, UMLAL, SMULL, and SMLAL*
USAT	Rd, #n, Rm {,shift #s}	Unsigned Saturate	Q	*SSAT and USAT*
UXTB	{Rd,} Rm {,ROR #n}	Zero extend a Byte	-	*SXT and UXT*
UXTH	{Rd,} Rm {,ROR #n}	Zero extend a Halfword	-	*SXT and UXT*

Appendix C: Memory map

Table C.1 shows a rough memory map. Detailed information can be extracted from chapter 2 of the document **TM4C123GH6PM Microcontroller Data Sheet**.

Table C.1 Memory usage for Coretx-M4 and TM4C123GH6PM Microprocessor

Address range	General use	Address range details for our boards	Description
0x0000 0000 – 0x3FFF FFFF	On-chip non-volatile memory	0x0000 0000 – 0x0007 FFFF	512 kB Flash memory
	On-chip SRAM	0x2000 0000 – 0x2000 7FFF	32 kB user program memory
0x4000 0000 – 0x400F FFFF	Peripherals		GPIO Ports, UARTs, I2C, …
0xE000 0000 – 0xFFFF FFFF	Private Peripheral Bus	0xE000 E000 – 0xE000 EFFF	Cortex-M4 functions including NVIC and System Tick Timer

Appendix D: Input / Output ports

Detailed information on this topic can be found in chapter 10 of the document **Tiva TM4C123GH6PM Microcontroller** [1].

The I/O ports are quite complex compared to older MCUs because there each pin is overloaded – there is more functionality within the MCU than can be brought out to external pins. I/O pins can each have upto 7 different functions. Minimizing the number of external pins allows a physically smaller, and cheaper, MCU to be made.

Appendix E: Exception and Interrupts

The TM4C123GH6PM has a large number of possible interrupt sources – external pins, internal counters and much more.

Nested Vectored Interrupt Controller (NVIC) is an integral part of the ARM Cortex CPU handles all interrupts or exceptions. When an interrupt is generated the processor using a table of interrupt points to jump to the appropriate routine. **Section 3.1.2 Nested Vectored Interrupt Controller** of [1] provides more information.

Table 2-9 provides information as to which interrupt source generates which interrupt number / bit. Table 3-8 details the registers which allow one to set (enable) or clear (disable) individual interrupts.

The GPIOIS, GPIOIBE, GPIOEV, and GPIOIM registers allow one to configure the type, event and mask of the interrupts for each GPIO Port. A certain order, detailed in section 10.3, is necessary to prevent generating spurious interrupts.

Appendix G: Hand Assembly

ARM7500 (1995 vintage) instructions are 32-bits long and are documented here: http://infocenter.arm.com/help/topic/com.arm.doc.ddi0050c/DDI0050C_7500_ds.pdf This is the information needed to hand assemble one class of instruction. Note that this is a partial table of what can be hand assembled. For instance the MOV instruction also allows a 16-bit value to be transferred into a register but that is not covered here.

Of note is that all of these instructions can be conditional upon the ZNCV status bits. A modern processor is limited by how well conditional branches can be predicted and conditional instructions (MOV, math, logic) allow one to write code without conditional branches.

Bit Position	31-28	27-26	25	24-21	20	19-16	15-12	11-0
Bit Content	Condition	0 0	RI	OP code	S	Rn	Rd	Operand 2

Here is an abbreviated table of the possible conditions:

Condition Bits	Condition suffix	Name	Condition Tested
0000	EQ	Equal to zero	Z = 1
0001	NE	Not equal to 0	Z = 0
0010	CS	Carry Set	C = 1
0011	CC	Carry Clear	C = 0
0100	MI	Minus (negative)	N = 1
0101	PL	Plus (positive or zero)	N = 0
1110	AL	Always	
1111	NV	Never	

Bit 25 “RI” sets the twelve “Operand 2” bits to Register or Immediate value. Assume the shift and rotate operations are to the left. 0xF1 can be rotated upto 16 times to give 0x00f10000

Bit 25 “RI”	Second Operand	First field	Second field
0	Register	Bits 11-4 (Shift amount)	Bits 3-0 (Register #)
1	Immediate Value	Bits 11-8 (Rotate amount)	Bits 7-0 (Immediate value)

Mnemonic	Op code		Mnemonic	Op code		Mnemonic	Op code
ADD	0100		EOR	0001		ORR	1100
ADC	0101		CMP	1010		AND	0000
SUB	0010		TEQ	1001		MOV	1101
SBC	0110		TST	1000		MVN	1111

The S bit is 1 if the condition flags are to be set, 0 otherwise.

Rn is the operand register. These bits go to a multiplexer to choose one of the 16 registers.

Rd is the destination register. These 4-bits drive a multiplexer to choose the destination register.

The second operand is either a register or immediate value depending upon bit 25 “RI”. An unsigned 8-bit immediate value can be rotated or a register can be shifted.

ORREQ R3, R2, R5

- ORR R2 and R5 and store the result in R3 but ONLY if the last operation to update the Z bit set it to 1 - that is to say that the result of the operation updating the status bits was EQUal to zero

Going thru the tables to get the bits:

Condition is 0000 for EQ

I (now called RI to make it more readable) is "0" because the 2nd operand is a register and not an immediate value.

The Op code is 1100 for ORR.

The S bit is 0 because the status bits are not being set by the ORR operation (that would be ORRS).

Rn is the operand for R2

Rd is the destination or R3

R5 is the second operand and it is not being shifted.

This gives us the bit pattern (broken down into the sections given in the Bit Position table):

0000 00 0 1100 0 0010 0011 00000000 0101

Cond. Opcod Rn Rd shift R5 (second operand)

Slice into 4 bit pieces and convert to hex gives:

0x01823005

Appendix H: ECE Tiva Shield

The ECE Tiva shield is expected to be available in Winter 2017. The LEDs on this shield have a different pinout than the simple LED shield that was used previously.

This shield provides a speaker, eight LEDs and two 7-segment displays. The mapping of those outputs is given below.

NOTE: the decimal point (DP) for the left most 7-segment uses port E1 which is the same as LED D2. DO NOT USE port E1 to drive the DP.

The TI Tiva is based upon the Stellaris and as a side effect there are two zero ohm resistors shorting port pins on the Tiva! On the Tiva R1-2, R9-16, R20 and R26 are all zero ohm resistors. **R9 shorts PD0 and PB6 and R10 shorts PD1 and PB7**.

Red text indicates the part name or name on the circuit board while black is the port signal name on the Tiva board.

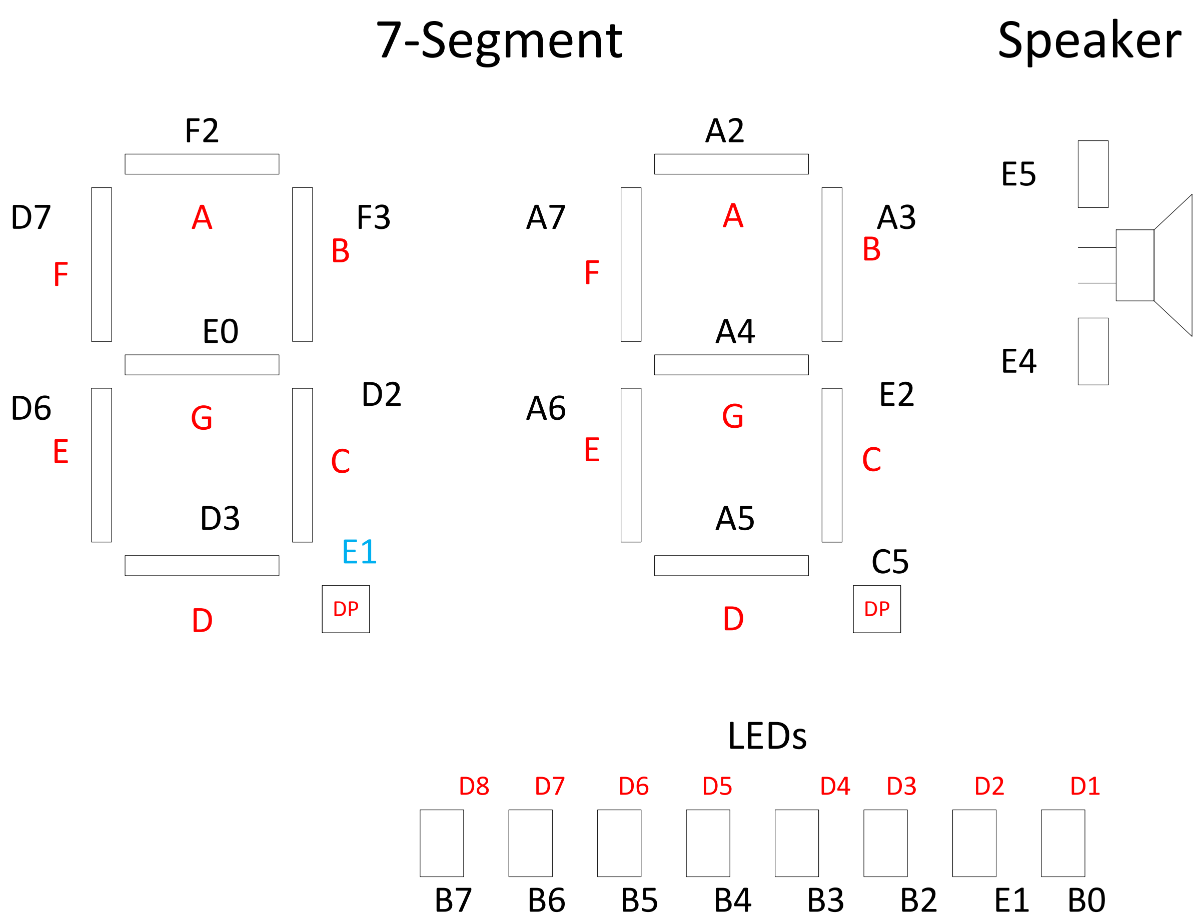


Figure 1 ECE Tiva Shield

References:

[1] ***Tiva TM4C123GH6PM Microcontroller Data Sheet***, Published by Texas Instruments Inc. It can be accessed online http://www.ti.com/lit/gpn/tm4c123gh6pm (Accessed on September 23, 2015)

[2] ***Getting Started, Creating Applications with µVision®5***, Published by Keil®. Can be accessed online: http://www2.keil.com/docs/default-source/default-document-library/mdk5-getting-started.pdf?sfvrsn=2 (Accessed on September 23, 2015)

[5] **Snap-shots** taken from the Keil µVision5 software.

[6] "**Morse Code**", From Wikipedia, the free encyclopedia. Can be accessed online: http://en.wikipedia.org/wiki/Morse_code (Accessed on August 21, 2012)

[7] **Cortex™-M4 Devices**, Generic User Guide, Published by ARM. Can be accessed online: http://infocenter.arm.com/help/topic/com.arm.doc.dui0553a/DUI0553A_cortex_m4_dgug.pdf (Accessed on September 23, 2015)

University Expectations and Policies

The following statements represent university expectations and policies with respect to academic integrity, discipline, grievance, student appeals, and academic accommodations. If you would like more clarification, please contact your course instructor directly.

Academic Integrity

In order to maintain a culture of academic integrity, members of the University of Waterloo community are expected to promote honesty, trust, fairness, respect and responsibility. [Check uwaterloo.ca/academic-integrity for more information.]

Grievance

A student who believes that a decision affecting some aspect of his/her university life has been unfair or unreasonable may have grounds for initiating a grievance. Read Policy 70, Student Petitions and Grievances, Section 4, uwaterloo.ca/secretariat/policies-procedures-guidelines/policy-70. When in doubt please be certain to contact the department's administrative assistant who will provide further assistance.

Discipline

A student is expected to know what constitutes academic integrity [check uwaterloo.ca/academic-integrity] to avoid committing an academic offence, and to take responsibility for his/her actions. A student who is unsure whether an action constitutes an offence, or who needs help in learning how to avoid offences (e.g., plagiarism, cheating) or about "rules" for group work/collaboration should seek guidance from the course instructor, academic advisor, or the undergraduate Associate Dean. For information on categories of offences and types of penalties, students should refer to Policy 71, Student Discipline, uwaterloo.ca/secretariat/policies-procedures-guidelines/policy-71. For typical penalties check Guidelines for the Assessment of Penalties, uwaterloo.ca/secretariat/policies-procedures-guidelines/guidelines/guidelines-assessment-penalties.

Appeals

A decision made or penalty imposed under Policy 70 (Student Petitions and Grievances) (other than a petition) or Policy 71 (Student Discipline) may be appealed if there is a ground. A student who believes he/she has a ground for an appeal should refer to Policy 72 (Student Appeals) uwaterloo.ca/secretariat/policies-procedures-guidelines/policy-72.

Note for Students with Disabilities

Access Ability Services, located in Needles Hall, Room 1132, collaborates with all academic departments to arrange appropriate accommodations for students with disabilities without compromising the academic integrity of the curriculum. If you require academic accommodations to lessen the impact of your disability, please register with Access Ability Services at the beginning of each academic term.