Brandon Ho 500727531

Lab2: Cortex M3 Features for Performance Efficiency

Introduction:

The ARM architecture optimizes execution time through the implementation of several key performance enhancing features. The first feature is bit-banding, which allows read and write operation to be executed on registers in a single machine cycle without the risk of additional delay from interrupts. This is accomplished through a bit-to-word memory association feature in the ARM architecture. The second feature is the use of predictive logic through the use the "S" suffix and the ITE-block assembly commands to more efficiently implement control flow through use of the PSR flags. Additional, "the barrel shifting feature" allows multiplication to be executed in one machine cycle through bit-shifting saving time in comparison to other methods of multiplication which may require multiple machine cycles to do the same task. These performance enhancing features will be studied and compared through the illumination of LEDs connected to PORT1 PIN 28 and PORT2 PIN2 on the NXP LPC 1768 microcontroller.

Procedure:

1) Create new project by selecting Project>New uVision Project.

2) A window will open prompting the work directory and Project name. Create a new work directory titled "Lab2" using the "New Folder "button and create the project under the name "Bitband".
3) A window will open. Specify the hardware to be used as LPC1768. This can be selecting by the hardware selection tree or found through the search bar.

4) Select the following Packages for the Run-Time-Environment
   Board Support> LED
   CMSIS>CORE
   Complier>Event Recorder
   Device>Startup, GPIO, PIN

5) Select "Source Group 1" from the Project tree, right click and select "Add New Item to Source Group 1".

6) Add the C file and H file titled "bitband" to the "Project Group 1".

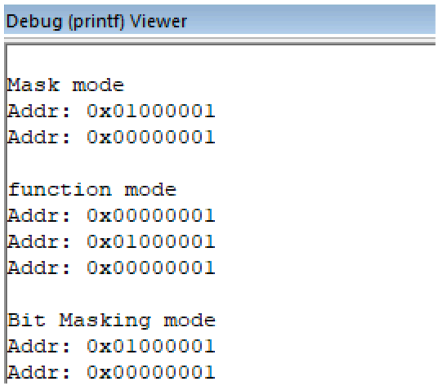
7) Under the "Options for Target" area do the following:
   a) Under Target tab>select "Use MicroLIB", set the ARM complier to "Use default complier version 5" and unselect IRAM2 option.

   b) Under C/C++ tab> select "C99 Mode"

c) Under Debug tab> select "Use Simulator", set the left-side Dialog DLL to DARMP1.DLL, set the associated left-side Parameter to "-LPC1768", set the right-side Dialog DLL to "TARMP1.DLL" and the associated right-side Parameter to "-LPC1768".

8) To execute "Bit Banding" example application, add the contents of the provided template "bitband.h" and "bitband.c" to the previously create uVision C and H files created in Step 6 under the same names.

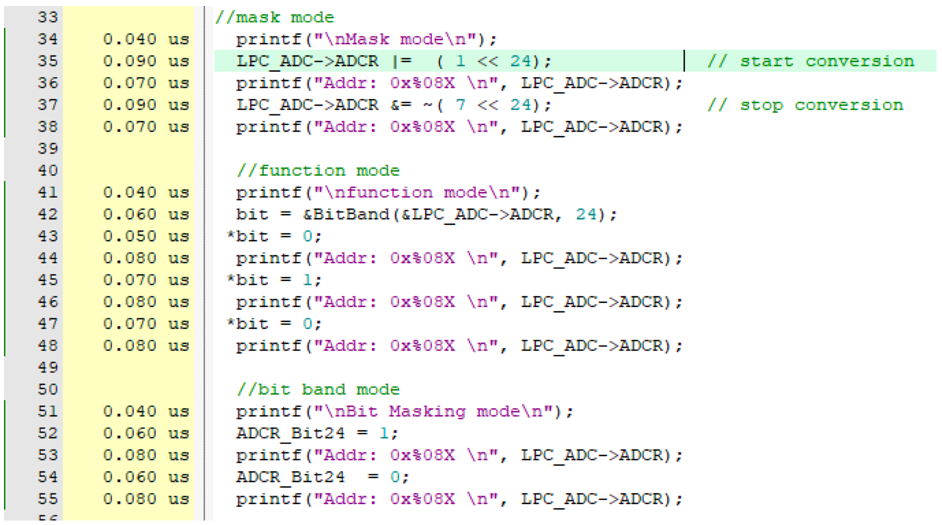
9) Build the project and enter the debugging mode. Under the "View" tab selecting Serial Window>Debug (printf) Viewer. (Refer to Fig1 for Viewer output)



```
Debug (printf) Viewer

Mask mode
Addr: 0x01000001
Addr: 0x00000001

function mode
Addr: 0x00000001
Addr: 0x01000001
Addr: 0x00000001

Bit Masking mode
Addr: 0x01000001
Addr: 0x00000001
```

Fig 1: Debug Viewer Demo (Part 3.4)

10) Under the "Debug" tab select the following Execution Profiling>Show Times, to show the execution times of each of the three methods implementing the "bitband.c" script. (Refer to Fig 2 for timing output)



```
              //mask mode
  0.040 us      printf("\nMask mode\n");
  0.090 us      LPC_ADC->ADCR |=  ( 1 << 24);             // start conversion
  0.070 us      printf("Addr: 0x%08X \n", LPC_ADC->ADCR);
  0.090 us      LPC_ADC->ADCR &= ~( 7 << 24);             // stop conversion
  0.070 us      printf("Addr: 0x%08X \n", LPC_ADC->ADCR);

                //function mode
  0.040 us      printf("\nfunction mode\n");
  0.060 us      bit = &BitBand(&LPC_ADC->ADCR, 24);
  0.050 us     *bit = 0;
  0.080 us      printf("Addr: 0x%08X \n", LPC_ADC->ADCR);
  0.070 us     *bit = 1;
  0.080 us      printf("Addr: 0x%08X \n", LPC_ADC->ADCR);
  0.070 us     *bit = 0;
  0.080 us      printf("Addr: 0x%08X \n", LPC_ADC->ADCR);

                //bit band mode
  0.040 us      printf("\nBit Masking mode\n");
  0.060 us      ADCR_Bit24 = 1;
  0.080 us      printf("Addr: 0x%08X \n", LPC_ADC->ADCR);
  0.060 us      ADCR_Bit24  = 0;
  0.080 us      printf("Addr: 0x%08X \n", LPC_ADC->ADCR);
```

Fig 1: Timing for three methods of bit manipulation

11) Compare the execution time between the three methods to register bit assignment. (Refer to Table 1 for total execution time of each method)

Mask Mode: Total time w/o Print statements = 0.090us + 0.090 = 0.18us
Function Mode: Total time w/o Print statements = 0.060 + 0.050 + 0.070 + 0.070 = 0.25us
Bit Masking Mode: Total time w/o Print statements = 0.060 + 0.060 = 0.12us

Table 1: Register Bit Toggle Timing Summation.

The “bitband.c” code partial shown in Figure 1 toggles bit-24 of the LPC_ADC->ADCR register, where the following bit assignment methods are shown below:

1) The code from line 34-38 implements “Mask mode” which toggles bit-24 by using bit-operation.

2) The code from line 41-48 implements “function mode” by implementing a Bitband() function which accepts a target address (such as the register for LPC_ADC->ADCR) and applies Bit-banding to toggle bit-24 based on the formula described in section 3.2 of the lab manual.

3) The code from line 51-55 toggles implements “bit band mode” where bit-24 of address 0x42680060 (address of register LPC_ADC->ADCR) is directly toggled.

Comparing the execution time of “Mask”, “function” and “bit band” mode listed in Table 1, assignment of a bit on a register is most quickly executed using bit-banding directly on the address of a register. It should be noted that if the actual address of the register is unknown, the additional steps required to determine the address may increase the execution time to the point where it is more time efficient to toggle the bit using a mask with bit operations.

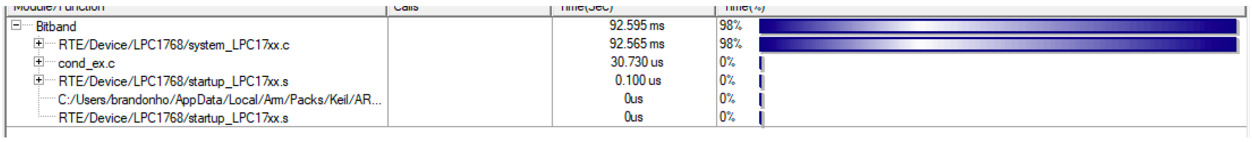
12) To execute “Conditional Execution” example select the “Source Group 1” tab under the Project tree and right-click. Under the option menu select “Manage Project Items”.

13) Remove “bitband.c” and “bitband.h” from the project and add template code “cond_ex.c”.

14) Under the “Options for Target” menu set the following options:
    a) the Under C/C++ tab>select Optimization>set to “-O0”.

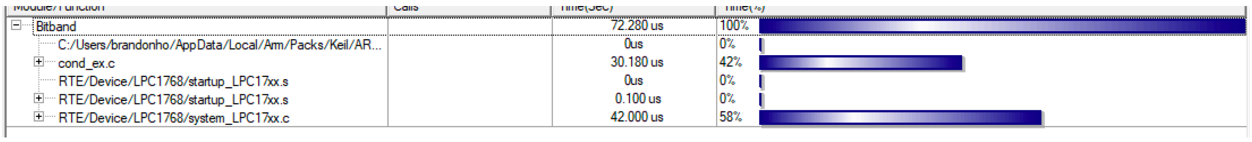
15) Build the project and enter Debug mode. Under “View” tab select Analysis Windows>Performance Analyzer.

16) Run the code and note the execution time in the Performance Analyzer. Repeat Step 14 with Optimization set to "-O3" and select "Optimize for Time". (Refer to Fig 3 and Fig4 for execution time comparison)



Module/Function	Calls	Time(Sec)	Time(%)
Bitband		92.595 ms	98%
RTE/Device/LPC1768/system_LPC17xx.c		92.565 ms	98%
cond_ex.c		30.730 us	0%
RTE/Device/LPC1768/startup_LPC17xx.s		0.100 us	0%
C:/Users/brandonho/AppData/Local/Arm/Packs/Keil/AR...		0us	0%
RTE/Device/LPC1768/startup_LPC17xx.s		0us	0%

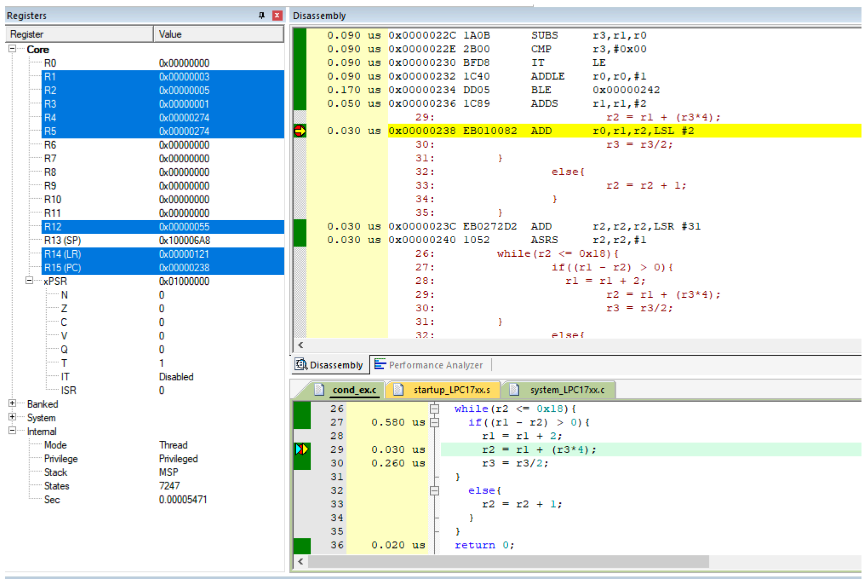
Fig 3: Level 0 Optimization



Module/Function	Calls	Time(Sec)	Time(%)
Bitband		72.280 us	100%
C:/Users/brandonho/AppData/Local/Arm/Packs/Keil/AR...		0us	0%
cond_ex.c		30.180 us	42%
RTE/Device/LPC1768/startup_LPC17xx.s		0us	0%
RTE/Device/LPC1768/startup_LPC17xx.s		0.100 us	0%
RTE/Device/LPC1768/system_LPC17xx.c		42.000 us	58%

Fig 4: Level 3 Optimization

17) To execute the "Barrel Shift" example append the main function to include the code that is commented out and conversely comment out the code that is currently active. Ensure Optimization is set to "-O3" and "Show Times" are selected under the "Execution Profiling" menu. Build the project and run the Debug mode. Take note of the shift, "S" suffix and IT-block commands in the disassembly window. (See Fig 5 for output example)



Register	Value
Core	
R0	0x00000000
R1	0x00000003
R2	0x00000005
R3	0x00000001
R4	0x00000274
R5	0x00000274
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000055
R13 (SP)	0x100006A8
R14 (LR)	0x00000121
R15 (PC)	0x00000238
xPSR	0x01000000
N	0
Z	0
C	0
V	0
Q	0
T	1
IT	Disabled
ISR	0
Banked	
System	
Internal	
Mode	Thread
Privilege	Privileged
Stack	MSP
States	7247
Sec	0.00005471

```
 0.090 us 0x0000022C 1A0B      SUBS     r3,r1,r0
 0.090 us 0x0000022E 2B00      CMP      r3,#0x00
 0.090 us 0x00000230 BFD8      IT       LE
 0.090 us 0x00000232 1C40      ADDLE    r0,r0,#1
 0.170 us 0x00000234 DD05      BLE      0x00000242
 0.050 us 0x00000236 1C89      ADDS     r1,r1,#2
             29:                         r2 = r1 + (r3*4);
 0.030 us 0x00000238 EB010082  ADD      r0,r1,r2,LSL #2
             30:                         r3 = r3/2;
             31:         }
             32:                 else{
             33:                         r2 = r2 + 1;
             34:                 }
             35:         }
 0.030 us 0x0000023C EB0272D2  ADD      r2,r2,r2,LSR #31
 0.030 us 0x00000240 1052      ASRS     r2,r2,#1
             26:         while(r2 <= 0x18){
             27:                 if((r1 - r2) > 0){
             28:                   r1 = r1 + 2;
             29:                         r2 = r1 + (r3*4);
             30:                         r3 = r3/2;
             31:         }
             32:                 else{
```

```
26              while(r2 <= 0x18){
27    0.580 us    if((r1 - r2) > 0){
28                  r1 = r1 + 2;
29    0.030 us      r2 = r1 + (r3*4);
30    0.260 us      r3 = r3/2;
31              }
32                else{
33                  r2 = r2 + 1;
34                }
35              }
36    0.020 us  return 0;
```

Fig 5: ITE / "S" suffix / Barrel shift execution- PSR flag triggered –O3

Summary:

Part 1 Debug (Performance Analysis) -- SEE code Section1.txt

Referring to the submitted code titled Section1.txt, the “masked mode” method directly assigns values to the required registers through the OR-bit operation with the register of interest and a mask to toggle the LEDs per the lab manual. The implementation is shown in the following steps below:

1) The GPIO for PORT1 and PORT2 are configured for output through their respective FIODIR registers, by setting all PIN assignments to bit-value-one by ORing the register FIODIR with a mask featuring all bit-positions set to bit-value-one.

2) Toggling the LEDs ON is accomplished by assigning a bit-value-one on PORT1 PIN28 and PORT2 PIN2 by ORing the register FIOSET for each port with a mask specifying the PIN position of the LED to illuminate. For example, ORing the FIOSET PORT1 register with a mask that has bit-position 28 set to bit-value-one and the rest of the bits set to bit-value-zero will illuminate the LED at PORT1 PIN28. Similarly, ORing the FIOSET PORT1 register with a mask that has bit-position-two set to bit-value-one and the rest of the bits set to bit-value-zero will illuminate the LED at PORT2 PIN2.

3) Toggling the LEDs OFF is accomplished similarly to toggling the LEDs ON in Step 2, however the respective masks for each LED of interest must be ORed with register FIOCLR for each port instead.

The “function mode” method toggles the LEDs through manipulation of the registers through bit-banding, where a bit in a certain section of memory (bit-band region) maps to a word in a secondary section of memory (bit-band alias region). Consequently, modifying the word value in the alias region of memory associated with the registers of interest also modifies the respective bit position of the register. The “function mode” method is implemented as follows:

1) As with the masked mode implementation of the LEDs, all the pins associated with PORT1 and PORT2 need to be configured to output through their respective FIODIR registers. These are accessed by their aliased memory address determined automatically by the bitband() function provided in the lab template code. Once the alias addresses for FIODIR registers are determined for all the PINs, setting the value associated with the alias address to word-value-one also sets that respective bit in the FIODIR register to bit-value-one. This is executed by a loop by assigning the alias memory address for each FIODIR register PIN with a word-value-one.

2) Similar to the “masked method”, to toggle the LED ON the bits associated with PIN28 on PORT1 and PIN2 on PORT2 at their respective FIOSET registers must be set to bit-value-one. This is implemented by determining the alias memory address of

the register associated with PIN28 PORT1 and PIN2 PORT2 using the bitband() function and assigning a word-value-one to those alias memory addresses. Consequently, due to bit-banding, the bits on the FIOSET register associated with the alias memory addresses for the register are set to bit-value-one illuminating the LEDs.

3) Toggling the LEDs OFF is accomplished similar to toggling the LEDs ON, except that the FIOCLR register alias memory addresses must be determined by the bitband() function and assigned a word-value-one as opposed to the FIOSET register alias memory address.

The “Bit Masking mode” method logically functions similar to the “function mode” method as both methods modify the registers required to operate the LEDs by use of bit-banding. However, what differentiates the two implementation is that the “Bit Masking mode” method does not use the provided lab template bitband() function to determines the appropriate alias memory addresses to modify. All the alias memory addresses required to implement LED functionality via the FIODIR, FIOSET and FIOCLR registers are predetermined and define at the beginning of the “one.c” program show in Section1.txt. The correct alias memory addresses are determined by first selecting the appropriate Bit-Band-Base address specified in the User manual in Table101, then using the formula provided below, to determine the alias memory address associated with the register of interest. Once the alias memory address is determined, setting that alias memory address to word-value-one associated with the register of interest will also set the bit value of the register associate with that alias memory address to bit-value-one. For example, to set PIN0 on PORT1 to output via register FIODIR, we select the Bit-Band-Base memory address for FIODIR from Table101 of the user manual as 0x2009C020 and apply the following formula provided in the lab manual below:

$$where\ ARM\ M3\ has, \qquad bit\ band\ base\ address = 0x20000000$$
$$and\ , \quad bit\ band\ alias\ base\ address = 0x22000000$$

$$byte\ offset = bit\ of\ interest\ bit\ band\ base\ address - bit\ band\ base\ address$$

$$byte\ offset = bit\ band\ base\ address - 0x20000000$$

$$= 0x2009C020 - 0x20000000$$

$$= 0x0009C020$$

$$bit\ band\ alias\ address = bit\ band\ alias\ base\ address + (byte\ offset * 0x00000020) + (bit\ postion * 0x00000004)$$

$$bit\ band\ alias\ address = 0x22000000 + (byte\ offset * 0x00000020) + (bit\ postion * 0x00000004)$$

$$= 0x22000000 + (0x0009C020 * 0x00000020) + (0 * 0x00000004)$$

$$= 0x23380400$$

Assigning word-value-one to the alias memory address determined above at 0x23380400 consequently assigns the bit position zero on register FIODIR PORT1 to bit-value-one enabling the GPIO PIN0 PORT1 to output. Therefore, to enable all the PINS on PORT1 and PORT2 as output, the alias memory address for each PIN from 0-31 on each PORT must be determined manually and subsequently set to word-value-one. Similarly, to toggle the LEDs ON and OFF the alias memory addresses for PIN28 on PORT1 and PIN2 on PORT2 for the register FIOSET must be set to word-value-one to activate the LED and the register FIOCLR must be set to word-value-one to deactivate the LED.

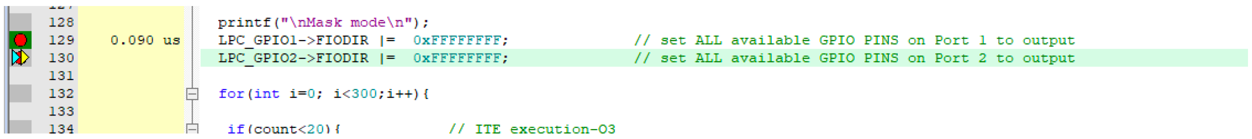
The execution time was compared between the three methods of implementing LED functionality by use of the performance analyzer and breakpoint in accordance to the uVision IDE Debugger execution support document [1]. By placing breakpoint between two common points of references in all three implementation methods and running the “one.c” program file between the breakpoint, accurate timings of each method could be determined and compared.

NOTE: To emphasize the performance of optimization level more accurately, the delay function was removed as it was determined -O3 optimization would remove any non-critical loops greatly skewing execution times.

Method	Execution time (-O0) No Delay	Execution time (-O3) No Delay	Performance Improvement
Mask	5.520us	3.860us	30%
BitBand() Function	17.570us	10.230us	41.7%
Direct Bit-Banding	9.490us	7.720us	18.6%

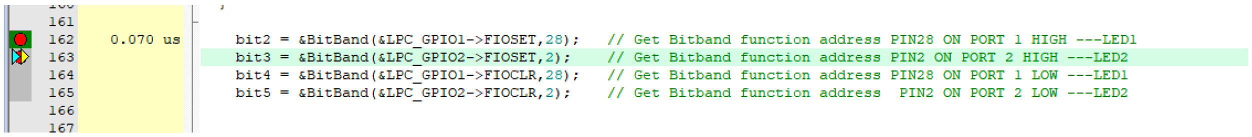
Table 2: Run-time performance (NO DELAY)

Referring to Table2, the “mask mode” method was determined to be the quickest, followed by the “Bit-Mask mode” method and the “function mode” method being the slowest. These results are consistent in both -O0 and -O3 optimization levels. The “masked mode” performed the best executing this specific “one.c” script, however, this may not always be the case as the execution time is dependent on the number of bit values assignments required. This is because the bit operations required to execute an OR operation between a mask and register require multiple cycles (fetch, modify and store) and can be slowed further by interrupt routines. In comparison, bit-banding can execute single instruction fetch and stores and consequently is a much more efficient operation. Referring to Fig 5 and Fig 7, assigning the GPIO PORT1 FIODIR register a bit-value-one requires more execution time using the “Mask mode” method (0.090us) in comparison to the “Bit-Mask mode” method (0.050us), as was originally expected (See Table 1). However, these results conflict with the run-times determined in Table 2 for the complete LED script to execute using each implementation method. Table 2 suggests that for a large amount of bit assignments on a few key registers the “Masked mode” method implementation is most efficient. This makes sense, as the “Mask mode” method re-writes the entire register no matter the number of individual bits required to be changed. However, for single bit assignments on multiple registers the “Mask-bit mode” method is the most efficient. Consequently, the “Mask mode” was most efficient method to implement the “one.c” script because to illuminate the LEDs the bit assignments were almost entirely required on the register FIODIR with sixty-four required bit assignments across the two FIODIR registers. Only four additional bit assignments to toggle the LEDs on and off are required on registers FIOSET and FIOCLR. I comparison, when modifying the FIODIR register the “Bit-mask mode” method must individually assign each bit values one at a time increasing overall execution time. Referring to Table 2, the “function mode” method required the most amount of time to execute the “one.c” script, which can be attributed this implementation containing the most complexity. For each bit assignment, the bitband() function embedded in the “function mode” method needs to determine the appropriate alias memory address significantly increasing complexity and the execution time of this method. Additionally, since this method is also based on bit-banding it features all the disadvantages of the “Bit-Mask mode” method as well. However, referring to Fig 6, it should be noted that the execution time for a single bit assignment with the “function mode” method (0.070us) is still quicker than the “Mask mode” method (0.090us). Therefore, when structuring code on ARM based architecture, bit-banding based methods should be used for individual bit assignments and masking based methods should be used when an entire register needs its bit values re-written.



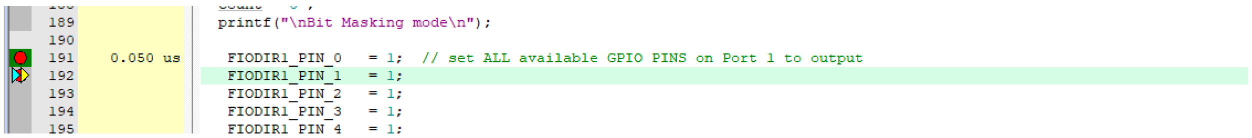
```
              printf("\nMask mode\n");
0.090 us      LPC_GPIO1->FIODIR |=  0xFFFFFFFF;             // set ALL available GPIO PINS on Port 1 to output
              LPC_GPIO2->FIODIR |=  0xFFFFFFFF;             // set ALL available GPIO PINS on Port 2 to output

              for(int i=0; i<300;i++){

               if(count<20){            // ITE execution-O3
```

Fig 5: Mask method execution time



```

0.070 us        bit2 = &BitBand(&LPC_GPIO1->FIOSET,28);   // Get Bitband function address PIN28 ON PORT 1 HIGH ---LED1
                bit3 = &BitBand(&LPC_GPIO2->FIOSET,2);    // Get Bitband function address PIN2 ON PORT 2 HIGH ---LED2
                bit4 = &BitBand(&LPC_GPIO1->FIOCLR,28);   // Get Bitband function address PIN28 ON PORT 1 LOW ---LED1
                bit5 = &BitBand(&LPC_GPIO2->FIOCLR,2);    // Get Bitband function address  PIN2 ON PORT 2 LOW ---LED2
```

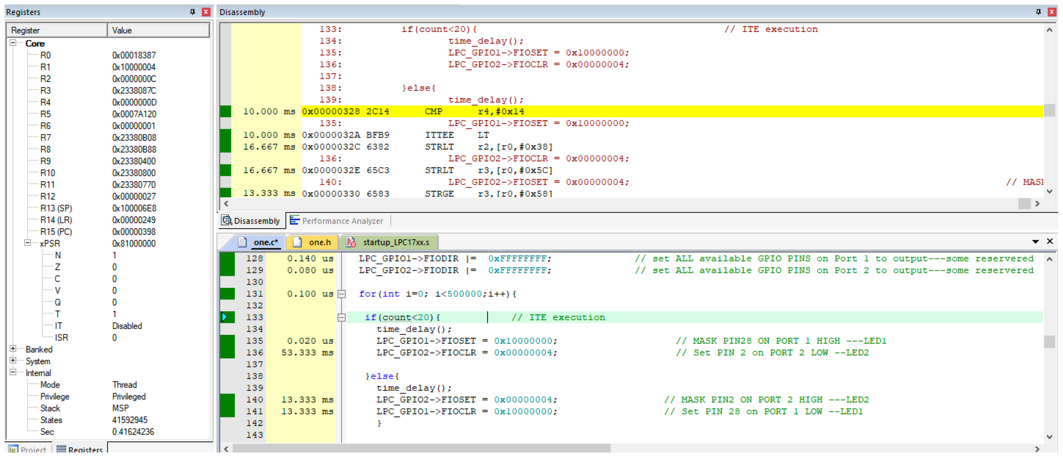
Fig 6: Function() method execution time



```
              printf("\nBit Masking mode\n");

0.050 us       FIODIR1_PIN_0    = 1;  // set ALL available GPIO PINS on Port 1 to output
               FIODIR1_PIN_1    = 1;
               FIODIR1_PIN_2    = 1;
               FIODIR1_PIN_3    = 1;
               FIODIR1_PIN_4    = 1;
```

Fig 7: Bit-Mask method execution time

Referring to Fig 8, when the “demo” version of the “one.c” program in simulated, it can be observed in the -O3 optimization that the “if” statements in the script is executed in assembly using the ITE-block implementation. In comparison, referring to Fig 9 when executing the “one.c” program in simulation at -O0 optimization the same “if” statement in the script is executed non-conditionally using a branch command. No implementation of the “S” suffix operator was observed.

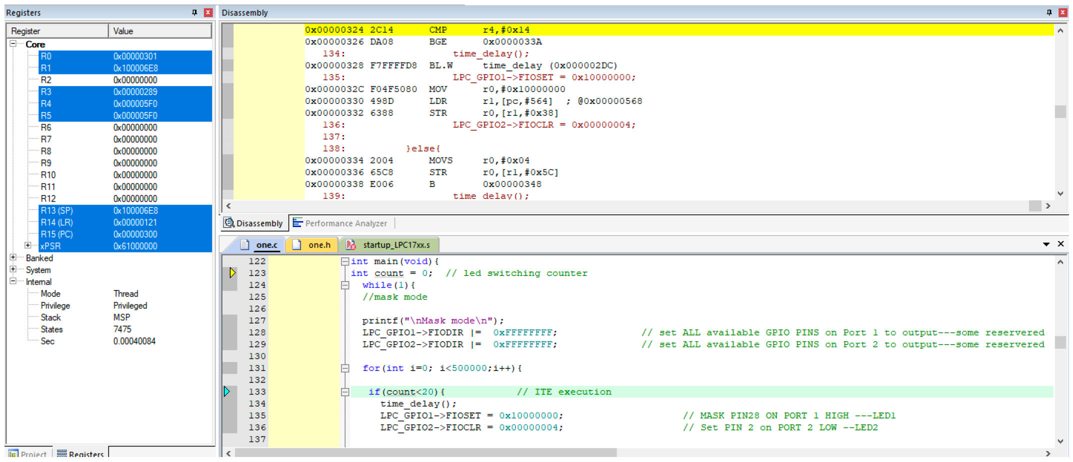


Register	Value
Core	
R0	0x00018387
R1	0x10000004
R2	0x0000000C
R3	0x2338087C
R4	0x0000000D
R5	0x0007A120
R6	0x00000001
R7	0x23380B08
R8	0x23380B88
R9	0x23380400
R10	0x23380800
R11	0x23380770
R12	0x00000027
R13 (SP)	0x100006E8
R14 (LR)	0x00000249
R15 (PC)	0x00000398
xPSR	0x81000000
N	1
Z	0
C	0
V	0
Q	0
T	1
IT	Disabled
ISR	0
Banked	
System	
Internal	
Mode	Thread
Privilege	Privileged
Stack	MSP
States	41592945
Sec	0.41624236

```
                133:          if(count<20){                                    // ITE execution
                134:                time_delay();
                135:                LPC_GPIO1->FIOSET = 0x10000000;
                136:                LPC_GPIO2->FIOCLR = 0x00000004;
                137:
                138:          }else{
                139:                time_delay();
 10.000 ms 0x00000328 2C14      CMP      r4,#0x14
                135:                LPC_GPIO1->FIOSET = 0x10000000;
 10.000 ms 0x0000032A BFB9      ITTEE    LT
 16.667 ms 0x0000032C 6382      STRLT    r2,[r0,#0x38]
                136:                LPC_GPIO2->FIOCLR = 0x00000004;
 16.667 ms 0x0000032E 65C3      STRLT    r3,[r0,#0x5C]
                140:                LPC_GPIO2->FIOSET = 0x00000004;                                // MASI
 13.333 ms 0x00000330 6583      STRGE    r3,[r0,#0x58]
```

```
 0.140 us   LPC_GPIO1->FIODIR |=  0xFFFFFFFF;          // set ALL available GPIO PINS on Port 1 to output---some reservered
 0.080 us   LPC_GPIO2->FIODIR |=  0xFFFFFFFF;          // set ALL available GPIO PINS on Port 2 to output---some reservered

 0.100 us   for(int i=0; i<500000;i++){

             if(count<20){            // ITE execution
               time_delay();
 0.020 us      LPC_GPIO1->FIOSET = 0x10000000;                   // MASK PIN28 ON PORT 1 HIGH ---LED1
53.333 ms      LPC_GPIO2->FIOCLR = 0x00000004;                   // Set PIN 2 on PORT 2 LOW --LED2

             }else{
               time_delay();
13.333 ms      LPC_GPIO2->FIOSET = 0x00000004;                 // MASK PIN2 ON PORT 2 HIGH ---LED2
13.333 ms      LPC_GPIO1->FIOCLR = 0x10000000;                 // Set PIN 28 on PORT 1 LOW --LED1
               }
```

Fig 8: Example Conditional ITE execution --O3 (demo code)



Register	Value
Core	
R0	0x00000301
R1	0x100006E8
R2	0x00000000
R3	0x00000289
R4	0x000005F0
R5	0x000005F0
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 (SP)	0x100006E8
R14 (LR)	0x00000121
R15 (PC)	0x00000300
xPSR	0x61000000
Banked	
System	
Internal	
Mode	Thread
Privilege	Privileged
Stack	MSP
States	7475
Sec	0.00040084

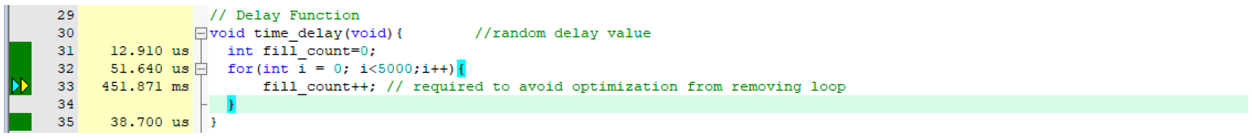
```
0x00000324 2C14      CMP      r4,#0x14
0x00000326 DA08      BGE      0x0000033A
   134:                 time_delay();
0x00000328 F7FFFFD8  BL.W     time_delay (0x000002DC)
   135:                 LPC_GPIO1->FIOSET = 0x10000000;
0x0000032C F04F5080  MOV      r0,#0x10000000
0x00000330 498D      LDR      r1,[pc,#564]  ; @0x00000568
0x00000332 6388      STR      r0,[r1,#0x38]
   136:                 LPC_GPIO2->FIOCLR = 0x00000004;
   137:
   138:           }else{
0x00000334 2004      MOVS     r0,#0x04
0x00000336 65C8      STR      r0,[r1,#0x5C]
0x00000338 E006      B        0x00000348
   139:                 time_delay();
```

```
int main(void){
int count = 0;  // led switching counter
  while(1){
  //mask mode

  printf("\nMask mode\n");
  LPC_GPIO1->FIODIR |=  0xFFFFFFFF;             // set ALL available GPIO PINS on Port 1 to output---some reservered
  LPC_GPIO2->FIODIR |=  0xFFFFFFFF;             // set ALL available GPIO PINS on Port 2 to output---some reservered

  for(int i=0; i<500000;i++){

   if(count<20){            // ITE execution
     time_delay();
     LPC_GPIO1->FIOSET = 0x10000000;                   // MASK PIN28 ON PORT 1 HIGH ---LED1
     LPC_GPIO2->FIOCLR = 0x00000004;                   // Set PIN 2 on PORT 2 LOW --LED2

```

Fig 9: Example: Non-Conditional execution CMP via BGE command -O1 (demo-code)

Part 2 Target (Demo) -- SEE code Section2.txt

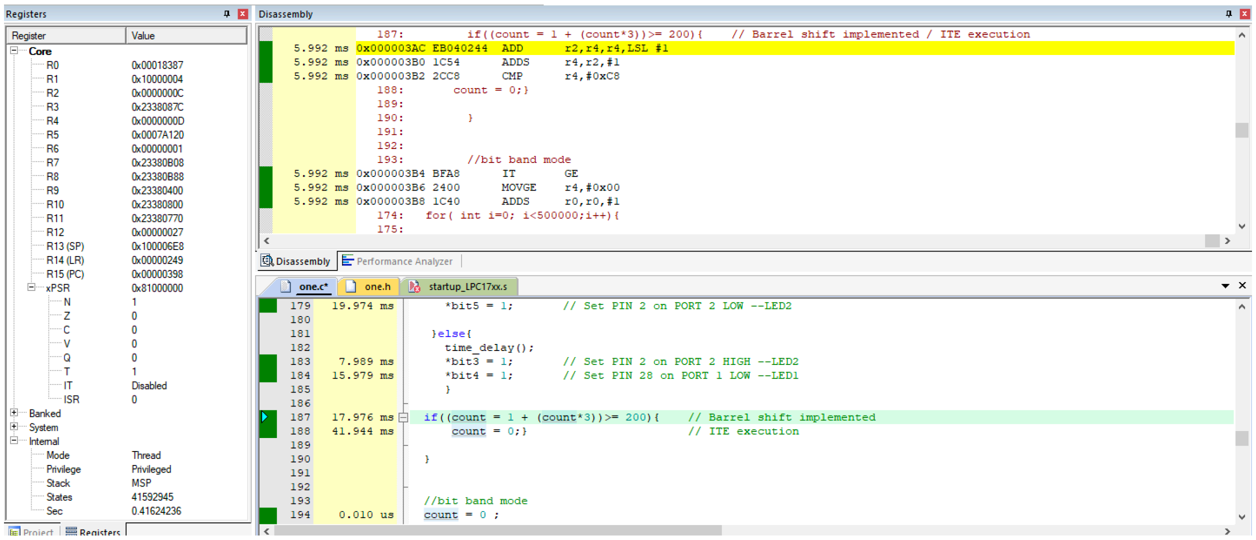
The delay() function required to strobe the LED ON and OFF for Part 2 of the lab is based off a "for loop". It can be observed from Fig 10 by using the execution profiling tool to show the execution times, that the delay adds approximately 451.8ms to the run-time (each time the delay function is called).



```
             // Delay Function
             void time_delay(void){         //random delay value
 12.910 us     int fill_count=0;
 51.640 us     for(int i = 0; i<5000;i++){
451.871 ms        fill_count++; // required to avoid optimization from removing loop
               }
 38.700 us   }
```

Fig 10: Example: Delay Function -O1 only (451.871ms)

The “barrel function” was implemented as a counter to strobe LED1 and LED2 alternatively for each method using conditional statements (See Fig 11). It should be noted, that the “LSL” shift operator only requires one machine cycle to execute. In comparison, the multiply function would require multiple machine cycles to execute. Consequently, shifting is a more time efficient process for multiplication operations.



Registers

Register	Value
Core	
R0	0x00018387
R1	0x10000004
R2	0x0000000C
R3	0x2338087C
R4	0x0000000D
R5	0x0007A120
R6	0x00000001
R7	0x23380B08
R8	0x23380B88
R9	0x23380400
R10	0x23380800
R11	0x23380770
R12	0x00000027
R13 (SP)	0x100006E8
R14 (LR)	0x00000249
R15 (PC)	0x00000398
xPSR	0x81000000
N	1
Z	0
C	0
V	0
Q	0
T	1
IT	Disabled
ISR	0
Banked	
System	
Internal	
Mode	Thread
Privilege	Privileged
Stack	MSP
States	41592945
Sec	0.41624236

Disassembly

```
            187:           if((count = 1 + (count*3))>= 200){     // Barrel shift implemented / ITE execution
5.992 ms 0x000003AC EB040244  ADD       r2,r4,r4,LSL #1
5.992 ms 0x000003B0 1C54      ADDS      r4,r2,#1
5.992 ms 0x000003B2 2CC8      CMP       r4,#0xC8
            188:         count = 0;}
            189:
            190:           }
            191:
            192:
            193:           //bit band mode
5.992 ms 0x000003B4 BFA8      IT        GE
5.992 ms 0x000003B6 2400      MOVGE     r4,#0x00
5.992 ms 0x000003B8 1C40      ADDS      r0,r0,#1
            174:      for( int i=0; i<500000;i++){
            175:
```

```
19.974 ms       *bit5 = 1;        // Set PIN 2 on PORT 2 LOW --LED2

              }else{
                time_delay();
 7.989 ms       *bit3 = 1;        // Set PIN 2 on PORT 2 HIGH --LED2
15.979 ms       *bit4 = 1;        // Set PIN 28 on PORT 1 LOW --LED1
                }

17.976 ms     if((count = 1 + (count*3))>= 200){     // Barrel shift implemented
41.944 ms         count = 0;}                         // ITE execution

              }


              //bit band mode
 0.010 us     count = 0 ;
```

Fig 11: Example Barrel shift -O3 (demo-code)

Comparing the different script versions of “one.c” to evaluate Part 1 and Part 2 of the lab, one key difference was the drastic difference in execution speed between optimization levels and how this interacted with the “for loop” based delay function. It can be observed from Table 2, that as much as a 42% improve in execution times were measured. It should be noted that this improvement in execution time is directly related to the script being executed using LTE-block implementation (-O3) instead of non-conditionally execution via branch statements. Additionally, when executing “one.c” script in -O3 optimization, the “for loop” based delay function is “optimized out” by the complier, essentially removing the delay to further improve execution time of the script. To execute the “demo” of the “one.c” script in -O3 optimization, large modifications are required to the loops implementing the LED illumination to slow down execution. Implementing the delay function using the timer capabilities of the LPC1768 should be used for time critical/sensitive applications.

References:

1) µVISION DEBUGGER: MEASURING EXECUTION TIME, https://www.keil.com/support/docs/971.htm,2020
2) NXP User Manual, https://www.nxp.com/docs/en/user-guide/UM10360.pdf, 2020
3) ARM Keil User Guide, https://www.keil.com/support/man/docs/mcb1700/mcb1700_intro.htm, 2020