Objective

This lab project dives into embedded ARM programming by managing GPIO pins through various methods. Each method achieves the same effect, but with different performance characteristics. Further, unique ARM architecture optimizations such as conditional execution and barrel shifting are explored.

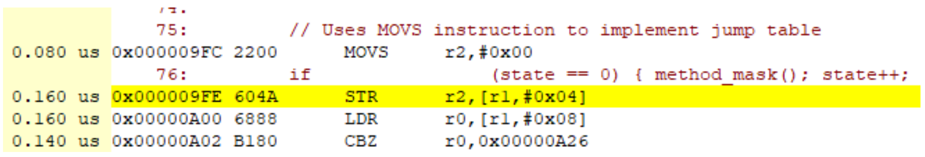
Implementation

Software

Building on the first lab, Lab 2 requires exploring the compiled assembly of the project. To set up the debugging environment, several features were first enabled in the simulator:

- *Execution Profiling > Show Times*: Used to measure CPU time for relevant sections of the code
- *C/C++ > Optimization > (O0/O3)*: Toggles compiler optimizations. Although -O0 is measurably slower, it is useful for inspecting raw instructions coherently before the compiler mangles the output for performance.

Once enabled, the debugger should provide output containing instructions alongside the corresponding c code, notice the instructions with their respective execution time:



```
               75:          // Uses MOVS instruction to implement jump table
0.080 us 0x000009FC 2200        MOVS      r2,#0x00
               76:          if                  (state == 0) { method_mask(); state++;
0.160 us 0x000009FE 604A        STR       r2,[r1,#0x04]
0.160 us 0x00000A00 6888        LDR       r0,[r1,#0x08]
0.140 us 0x00000A02 B180        CBZ       r0,0x00000A26
```

Figure 1: Example of conditional execution through a jump table

Hardware

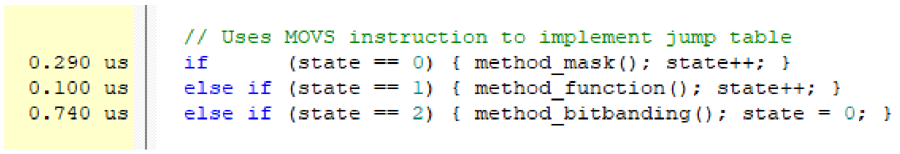
The project requires access to the GPIO pins of the board for LED access, the SysTick implementation for introducing necessary delay, as well as the LCD for debugging purposes. The user manual for the LPC17xx chip was heavily consulted, for gathering memory addresses useful for programming I/O, and configuring the SysTick timer to provide useful interrupts.

Assignment

The program can be logically considered as a small state machine, that is configured to change state every 500ms. Each state represents one of the three methods of accessing GPIO pins, using its execution time to toggle an individual LED on GPIO ports 1 and 2.

The `main()` function in this program is only used for chip configuration, only running a few lines of code before looping indefinitely. After initializing the LED ports, the *SysTick* timer is configured using the internal clock, which will run its interrupt handler periodically. This handler callback function is delegated the actual logic of the program, which will run as the main thread is looping through no-ops. The `SysTick_Handler()` function is used to implement the state transitions: it is designed to run each millisecond, but will accumulate a configured value of *ticks* before actually transitioning.

Each method of addressing is handled within its own function. The decision to wrap the execution of each method within a function was made to present neat boundaries for measuring execution time of each method. An example run is shown:



```
           // Uses MOVS instruction to implement jump table
0.290 us   if      (state == 0) { method_mask(); state++; }
0.100 us   else if (state == 1) { method_function(); state++; }
0.740 us   else if (state == 2) { method_bitbanding(); state = 0; }
```

Figure 2: Runtime profiling the state machine

The compiler is usually able to optimize out any static calculations through static analysis, and each function will likely be inlined in *-O3 mode*, thus, this method provides consistent measurements of the relevant part of each function: writing bits to the specified addresses.

Results

Table 1: Comparing performance of each method and optimization improvements

Method	Time (-O0)	Time (-O3)	% Improvement
Masking	0.510us	0.120us	76.4%
`BitBand()` function	0.180us	0.040us	77.8%
Direct Bit Banding	1.470us	0.250us	82.9%

Masking and direct bit banding offer measurably slower execution once results are normalized. Since they are writing to the GPIO interfaces first, the internal logic of the chip responsible for controlling GPIO is invoked, adding a runtime penalty. The function method bypasses this extraneous GPIO logic – writing directly to the register – and thus offers the best performance.

All methods receive a large, yet roughly similar improvement when compiled with optimizations, which is to be expected.

Appendix

```
/* bitband.c*/

#include "LPC17xx.h"
#include "GLCD.h"

#include <stdio.h>

//------- ITM Stimulus Port definitions for printf ------------------- //
#define ITM_Port8(n)    (*((volatile unsigned char *)(0xE0000000+4*n)))
#define ITM_Port16(n)   (*((volatile unsigned short*)(0xE0000000+4*n)))
#define ITM_Port32(n)   (*((volatile unsigned long *)(0xE0000000+4*n)))

#define DEMCR           (*((volatile unsigned long *)(0xE000EDFC)))
#define TRCENA          0x01000000

struct __FILE { int handle;  };
FILE __stdout;
FILE __stdin;

int fputc(int ch, FILE *f) {
  if (DEMCR & TRCENA) {
    while (ITM_Port32(0) == 0);
    ITM_Port8(0) = ch;
  }
  return(ch);
}
//------------------------------------------------------------------- //

#define __USE_LCD 0 // Uncomment to use the LCD
#define __FI      1 // Font index 16x24

// Bit Band Macros used to calculate the alias address at run time
#define ADDRESS(x)    (*((volatile unsigned long *)(x)))
#define BitBand(x, y)   ADDRESS(((unsigned long)(x) & 0xF0000000) | 0x02000000 |(((unsigned

#ifdef __USE_LCD
static inline void method2lcd(unsigned char* msg) {
  GLCD_DisplayString(6,  8, __FI,  msg);
}
#endif
```

```
// Simple register masking
static void method_mask(){
    LPC_GPIO1->FIOPIN ^= (1 << 28);
    LPC_GPIO2->FIOPIN ^= (1 << 2);
}

// Define pointer with bitband method
static void method_function(){
    volatile unsigned long* bit1 = &BitBand(&LPC_GPIO1->FIOPIN, 29);
    volatile unsigned long* bit2 = &BitBand(&LPC_GPIO2->FIOPIN, 3);

    static _Bool state = 1;

    *bit1 = *bit2 = state;
    state = !state;
}

// Raw bitbanding
static void method_bitbanding() {
    const size_t addr1 = 0x22000000 + (0x2009C034 * 32UL) + (31 * 4);
    const size_t addr2 = 0x22000000 + (0x2009C054 * 32UL) + (4 * 4);

    static _Bool state = 1;

    ADDRESS(addr1) = ADDRESS(addr2) = state;
    state = !state;
}

void SysTick_Handler(void) {
    static size_t tick = 0;
    static size_t state = 0;

    if (tick++ < 500) { return; }
    tick = 0;

    // Uses MOVS instruction to implement jump table
    if          (state == 0) { method_mask(); state++; }
    else if (state == 1) { method_function(); state++; }
    else if (state == 2) { method_bitbanding(); state = 0; }

    #ifdef __USE_LCD
    if          (state == 1) { method2lcd("MASK    "); }
    else if (state == 2) { method2lcd("FUNCTION"); }
    else if (state == 0) { method2lcd("BITBAND "); }
    #endif
```

```
}

int main(void){
    LPC_SC->PCONP     |= (1 << 15);            /* enable power to GPIO & IOCON  */
    LPC_GPIO1->FIODIR |= 0xB0000000;           /* LEDs on PORT1 are output      */
    LPC_GPIO2->FIODIR |= 0x0000007C;           /* LEDs on PORT2 are output          */

    // Configure SysTick with interrupt and internal clock source
    ADDRESS(0xE000E010) = (1 << 0) | (1 << 1) | (1 << 2);

    // Run handler every 1ms
    ADDRESS(0xE000E014) = 99999;

    #ifdef __USE_LCD
    GLCD_Init();                               /* Initialize graphical LCD (if enabled */

    GLCD_Clear(White);                         /* Clear graphical LCD display   */
    GLCD_SetBackColor(Blue);
    GLCD_SetTextColor(Yellow);
    GLCD_DisplayString(0, 0, __FI, "     COE718 Lab 2    ");
    GLCD_SetTextColor(White);
    GLCD_DisplayString(1, 0, __FI, "       bitband.c     ");
    GLCD_DisplayString(2, 0, __FI, "  Watch the LEDs!    ");
    GLCD_SetBackColor(White);
    GLCD_SetTextColor(Blue);
    GLCD_DisplayString(6, 0, __FI, "Method:");
    #endif

    // Let SysTick callback run in background
    while (1) {}
}
```