2019

Microcomputers: Lab Report

AHMAD ALLAHHAM & JANUS SILVESTRE

1 CONTENTS

2 LABORATORY 1: FAMILIARISATION WITH AN IDE

2.1 PRE-LABORATORY

1. **What are the differences between Program memory and Data memory?**

   Program Memory stores the program (the actual code being executed) in the form of instructions whilst data Memory stores the data that is operated on by the program. [1]

2. **What is the main difference between a microcontroller and a microprocessor?**

   Microprocessors contains a CPU in its die and have optimised architecture for external data manipulation & storage using queues and caches. Microcontrollers on the other hard are less complex than MPUs, optimised for use when little external devices are connected to it and have peripherals embedded into it.

3. **What are the differences between RISC and CISC architectures?**

   CISC (Complex Instruction Set Computing) have variable commands but sized & multiple clocks and aim is to simplify the program. It does this at the cost of hardware being more complex. RISC (Reduced Instruction Set Computing) aim to simplify its hardware structure with the design focus on simple instructions which result in more complicated and longer programs.

4. **What are the differences between Programmer model and Hardware model?**

   Hardware Model focus on hardware characteristics (understanding physical capabilities & input/output information such as memory and timing) whilst the Software Model focus on instruction set, syntax and execution time[1].

5. **What is the main purpose of an ALU?**

   The main purpose of the ALU is to do arithmetic operations as well as logic operations. [1]

6. **Why should programmers avoid using infinite loops?**

   Infinite loops are very source intensive & consume a lot of power. [1]

7. **What are alternatives to using infinite loops? (Better implementation of infinite loops)**

   Using low power mode with interrupts to awaken the system when the data needs to be processed is better as it saves power whilst keeping the same functionality as long as the right lower power mode has been selected. [1]

8. **What is the main use of functions in C?**

It is used to simplify and reuse code which provides better structure and helps maintain it a clean, less clustered program. It is necessary to do complex operations as you can break down a task into simpler functions which can call. [1]

9. **What is overhead in programming?**

   The resources allocated for a particular program such as the amount of memory used and the execution time are called overhead. [1]

10. **Why are there .c and .h version of libraries available? What is the difference between these?**

    The .c files contain the actual implementation for the functions and the .h files contains the forward declarations required for compiling. [2]

11. **It is possible to mix C programming and assembly language into a single program. Why do programmers do this and what are some of the methods they would use?**

    They mix assembly code with C for hardware specific functions such as interrupts which them more control over how data is stored and code is executed. This can be done either through inlining assembly instructions directly or by intrinsic functions which provide direct access to processor operations which are compiled and optimised to inline code. [1]

2.2 In-Laboratory

Using debugging mode allowed us to view the value inside the registers such as the PC and SP and step through line by line of code to see how they get updated.

In order to get the seven segment display to a particular character, we would use the binary equivalent of that character and either the switches on and off in order of the corresponding binary. To get “0” to display, we would turn off all the switches and to display an “A” (0000 1010 in binary), we would flick on the corresponding switches with the 1.

By looking through the code and by experimentation, we found that turning the switch values to a binary equivalent greater than 26 (or 00011010) would turn result in a blank seven segment display. Any value greater than this would cause the program to write with the value at the index defined for “Blank7Seg” (0x1B) to the seven segment display.

3 LABORATORY 2: INTERFACING WITH PERIPHERAL DEVICES

3.1 PRE-LABORATORY

1. **What is the use of program counter (PC) register?**

The PC register keeps track of the address of the next line of instruction to be executed. [3]

2. **What are Push and Pull operations when talking about the stack pointer?**

The push operation is writing into the stack while the pull operation is reading from the stack. [3]

3. **2 four-bit positive binary numbers have been added together. The result is a four-bit negative number. Has there been an overflow?**

Yes since the sign of the result is different from the sign of the operands. [3]

4. **What do the following flags determine in binary arithmetic operations: C, N, Z**

C: carry is if the last operation has a carry bit, Z: zero is set when the result is zero, N: negative is set when the result is negative. [3]

5. **How can the regular SR flags be used to compare two numbers together?**
   We can see which SR flags are triggered after the subtraction of two numbers to see equal, or one is bigger than the other. If the zero flag is set then they are equal and if the negative flag is set then the first number was smaller than the other. [3]
6. **How many different memory locations can be accessed using 12 bits?**

16^3 locations (ie: 2^12 locations, we multiply the sum of locations by 2 for every new bit that we have)

7. **Briefly describe the Address bus, Data bus, and the Control bus.**

Address bus are the lines used to transport address information. Data bus are set of lines used to transport data and instructions to and from the CPU. Control bus carry signals that regulate system activity such as read/write, synchronisation, etc. [4]

8. **What are the differences between volatile and non-volatile memories?**

Volatile memory is temporary and is erased after the power is turned off whilst non-volatile memory remains in memory after the power is cut off. [5]

9. **What are the unique characteristics of FRAM?**

FRAM is non-volatile memory that is writable at program execution voltages and only consume power while reading or writing and thus requires low energy to operate. Its writing speed is comparable to DRAMS. [4]

10. **In a memory system, data is 16-bit wide. Break the information into bytes with their addresses using little endian and big-endian methods. Starting address is F800 and the data is: E0F2 0041**

Using little endian:

F803	00
F802	41
F801	E0
F800	F2

Using big endian:

F803	41
F802	00
F801	F2
F800	E0

11. **What are the differences between Von Neumann and Harvard architecture?**

In Von Neumann, program and data memory share the same system buses whilst Harvard they have different system buses. [6]

12. **What are look up tables in C and how are they defined?**

A look up table is an initialised array indexed with symbolic names used to define the mapping between the information it holds and its location. They are defined during the program's initialisation stage.

13. **How do Master and Slave work in an I2C bus?**

The master addresses the slave and the slave responds using the I2C bus. The master will send each slave the address of the slave it wants to communicate to and each slave compares their address with the one sent from the master. If it matches, the master can then read or send the data from the slave. [7]

14. **What acknowledgment signals are there in I2C bus?**

ACK and NACK are used for acknowledging the sender that the frame was received successfully or unsuccessfully respectively. [7]

3.2 In-Laboratory

3.2.1 LED and Switches

By bit shifting the BusData to the left by one, we shift the current mapping between a numbered switch and its corresponding numbered LED. As each bit in the register that we are writing to represents a particular LED, we can shift the data being read from the switches such that the physical mapping is off by one bit with the LED it turns on. For example, assuming that the binary equivalent of the value of BusData is 0000 0100 which maps to LED3, shifting this would result in 0000 1000 which then maps Switch3 to LED4 as required.

Relevant code:

```
for(;;){

BusAddress=SwitchesAddr;

BusRead();

BusData=~BusData<<1;

BusAddress=LedsAddr;

BusWrite();

  }

}
```

3.2.2 Switch 2 Segment

The LookUpSeg array contains values used by the seven seg display and indexed by the number that will be displayed in order. By simply changing the index being accessed in the LookUpSeg array, we can cause a bigger number to be displayed to the seven segment displays. Normally, the original code simply reads Temp which is the binary equivalent of which switches have been turned on. This value is then used as the index to a lookup data with the actual data being used to display to the seven segment displays. By simply adding to the index by 1 and 2, we can cause a new value to be retrieved from the look up table which would be correspond to a greater number being displayed.

Relevant code:

BusData=LookupSeg[Temp+1];

BusAddress=SegLow;

BusWrite();

BusData=LookupSeg[Temp+2];

BusAddress=SegHigh;

BusWrite();

3.2.3 LCD

Master mode synchronisation

UCB1CTL0|=(UCMST+UCMODE_3+UCSYNC); // Master mode, I2C, Synchronous

Transmit/ ACLK

UCB1CTL1|=(UCTR+UCSSEL_1); // Transmit, ACLK

Slave address

UCB1I2CSA=0x3E; // Slave address

Writing name on LCD

const char ECE[16]=" Janus " ;

const char CURTIN[16]=" Ahmad ";

3.2.4 Keypad

The keypad program works similarly to the Switch 2 Segment from before in that it uses a lookup table to determine mappings between physical input and display data. By moving all the values down by 1, a lower index (compared to default) is required to retrieve the same data representation for the seven seg display. This would effectively shift the mapping from the keypad to the seven seg. For example, the seven seg value for 3 was 0x24 in the third index of array. By shifting down by one, the second index (which is mapped to 2 on the keypad) now retrieves the seven seg value for 3 and uses this to display a number greater what was pressed on the keypad.

Relevant code:

const char LookupSeg[16]={0x79,0x24,0x30,0x19,0x12,0x02,0x78,0x00,0x18,0x08,0x03,0x46,0x21,0x06,0x0E,0x40};

```
KeyCode=LookupSeg[Count];

    BusAddress=Seg7AddrL;

    BusData=KeyCode;

    BusWrite();

    BusAddress=Seg7AddrH;

    BusData=KeyCode;

    BusWrite();
```

4 LABORATORY 3: MSP430FR5739 DEVELOPMENT BOARD

4.1 PRE-LABORATORY

1. **What is a Watchdog Timer? Why is it a good practice to stop it at the beginning of MSP430 Programs?**

A watchdog timer is a special timer used to as a safety mechanism and prevent it the system from hanging from faults such as infinite loops. It does this by waiting from signals from the program which resets the timer, preventing it from resetting the program. It is disabled at the beginning as we are not doing anything which could cause the problems that are addressed by it and to prevent it from resetting our program. [8]

2. **Compare Parallel and Serial I/O interfaces. What are the advantages and disadvantages of using each?**

Parallel interface means that all the bits composing a single word are communicated simultaneously, so they would need one wire per bit. In the early days, parallel interfaces were more popular due to the ability of transmitting information on multiple wires which was faster than using a single wire. Its disadvantage is that modern systems transmit data at frequencies high enough and wires close enough that they can electromagnetically interfere with each other and cause errors in data transmission.

Serial interfaces require only one wire to transfer the information since they send information one bit at a time. Serial is more economically appealing, as it requires on a single wire. In the early days, using serial was a problem as it was slow to transmit using only a single wire due to slow speeds. [8]

3. **What is the difference between a buffered input and a latched input?**

Buffered input do not hold input data and the CPU can only read input present at that instant. Latched input on the other hand hold the data until it is read by the CPU, after which they are usually cleared and ready for another input. [8]

4. **Explain each of these concepts: OpCode, Operand, Addressing mode**

OpCode also known as the operating code which defines the operation to be performed. Operand is the data or registers that are being used in the instruction. They usually have a source and destination. Addressing mode refers to the way in which an operand defines the data; for example, the data may be the one explicitly given, the data stored in a register or the data stored a memory location. [9]

5. **What is contained in the source file?**

It contains the actual assembly instructions, any directives used that help organize the program through labels and symbolic names and are not compiled into machine code, comments, and macro directives that group instructions into one line for ease of reading. [9]

6. **What is the purpose of the linker?**

The linker combines previously assembled object files and libraries to generate the final executable file which is what is loaded onto the microcontroller's memory. [9

7. **What are the three types of CPU instructions? Briefly explain each of them.**

Data transfer copies data from source to destination without changing the source and the flags in SR are usually not affected by these. Arithmetic-logic do arithmetic and logic operations through the ALU and registers. Program control modifies the flow of execution through jumps and subroutine calls. [9]

8. **How do JUMP instructions affect the contents of program counter?**

The address of the next instruction to be executed is loaded into the program counter when a jump instruction is used. [9]

4.2 In-Laboratory

4.2.1 Enabling the interrupt

Interrupts are used to pause the execution of the software due to a change in hardware state.

Relevant Code:

```
//  TA1CCR0 = 0x4000;
//  TA1CTL = TASSEL_1 + MC_1 + TACLR;
//  TA1CCTL0 = CCIE;
```

4.2.2 Stopping the Watch dog

The watchdog timer is a timer that stops the program from going into infinite loops. Since we expect microprocessors such as the MSP430 to go into an infinite loop (i.e. to measure temperature over a long period of time), we stop the watchdog timer as we don't want it to pause the execution of the program if it detects an infinite loop.

Relevant Code:

```
WDTCTL = WDTPW + WDTHOLD;
```

4.2.3 Low Power Mode

Low power mode is used when the microprocessor is an a long (infinite) loop. Placing the microprocessor in low power mode turns off unused operations/capabilities in the microprocessor allowing it to save power and run for a longer period of time.

Relevant Code

```
__low_power_mode_3();
```

4.2.4 Question 4.3: 8 LEDs turn on and off

To allow for all the LEDs to turn on and off at the same time, we need to set the value of the output pin to correspond to the LEDs that need to be turned on and off. For example, since each output pin supports up to 4 LEDs (i.e. 8 bits) if we need to turn on all four LEDs, then the value for the output pin must be equal to 00001111 (for the first four LEDs) or 11110000 (for the last four LEDs. Therefore, we use the '&' bit wise operator to clear out all of the bits except the bits that correspond to the LEDs. For assigning the input, we use the '|' bit wise operator to add each of the bit values to the input pin.

Relevant Code:

```
// Set up one LED to pulse
  P3DIR |= BIT7;
  P3OUT |= BIT7;
// Remove comments below to........Toggle between two LEDs
  P3DIR |= BIT6;
  P3OUT &= BIT6;
  P3DIR |= BIT5;
  P3OUT &= BIT5;
  P3DIR |= BIT4;
  P3OUT &= BIT4;
  PJDIR |= BIT3;
  PJOUT &= BIT3;
  PJDIR |= BIT2;
  PJOUT &= BIT2;
  PJDIR |= BIT1;
  PJOUT &= BIT1;
  PJDIR |= BIT0;
  PJOUT &= BIT0;

  while(1)
  {
          __low_power_mode_3();       // Enter LPM3 w/ interrupt
          P3OUT ^= (BIT7 + BIT6 + BIT5 + BIT4);
          PJOUT ^=  (BIT3 + BIT2 + BIT1 + BIT0);
          counter++;
          __delay_cycles(100000);                   // Delay between transmissions
  }
}
```

4.2.5 Turning on external LED (BLINK WITH OTHER INTERNAL LEDs)

In order to turn on an external LED, we connect it to one of the pins on the MSP430 (here we use the 'P3DIR' pin) and add to it (using the pipeline bitwise operator) the 'BIT0' value (i.e. 00000001). This means that the first LED in the pin (i.e. the external LED) is assigned to that pin. The while loop turns on the external LED, applies a delay then a places the MSP430 in low power mode (effectively shutting down the external LED

Relevant Code:

```
// turning external LED on
  P3DIR |= BIT0;
  P3OUT |= BIT0;

  while(1)
  {
          __low_power_mode_3();       // Enter LPM3 w/ interrupt
          P3OUT ^= BIT0;
```

```
        counter++;
        __delay_cycles(100000);                // Delay between transmissions
  }
```

4.2.6 Turning on LEDs in order and reverse

The LEDs are assigned to the correct pins as discussed in section 4.2.4. Then, in a while loop, the value of the output pin is set equal to a BIT{7-0}, followed by a delay and a low power mode switch, effectively turning on each LED with a time gap between each turn on and off. Then, the operation is reversed (i.e. BIT{0-7}).

Relevant Code:

```
// Set up one LED to pulse
  P3DIR |= BIT7;
  P3OUT |= BIT7;
// Remove comments below to........Toggle between two LEDs
  P3DIR |= BIT6;
  P3OUT &= BIT6;
  P3DIR |= BIT5;
  P3OUT &= BIT5;
  P3DIR |= BIT4;
  P3OUT &= BIT4;
  PJDIR |= BIT3;
  PJOUT &= BIT3;
  PJDIR |= BIT2;
  PJOUT &= BIT2;
  PJDIR |= BIT1;
  PJOUT &= BIT1;
  PJDIR |= BIT0;
  PJOUT &= BIT0;
// turning external LED on
  P3DIR |= BIT0;
  P3OUT |= BIT0;

  while(1)
  {
//      P3OUT ^= BIT0; // turns on external LED
        P3OUT = BIT7;
        __delay_cycles(100000);
        __low_power_mode_3();
        P3OUT = BIT6;
        __delay_cycles(100000);
        __low_power_mode_3();
        P3OUT = BIT5;
        __delay_cycles(100000);
        __low_power_mode_3();
        P3OUT = BIT4;
        __delay_cycles(100000);
        __low_power_mode_3();
        P3OUT &= ~BIT4; // Magically clear BIT
```

```
            PJOUT = BIT3;
            __delay_cycles(100000);
            __low_power_mode_3();
            PJOUT = BIT2;
            __delay_cycles(100000);
            __low_power_mode_3();
            PJOUT = BIT1;
            __delay_cycles(100000);
            __low_power_mode_3();
            PJOUT = BIT0;
            __delay_cycles(100000); // Delay between transmissions
            __low_power_mode_3();        // Enter LPM3 w/ interrupt
            PJOUT = BIT1;
            __delay_cycles(100000);
            __low_power_mode_3();
            PJOUT = BIT2;
            __delay_cycles(100000);
            __low_power_mode_3();
            PJOUT = BIT3;
            __delay_cycles(100000);
            __low_power_mode_3();
            PJOUT &= ~BIT3;
            P3OUT = BIT4;
            __delay_cycles(100000);
            __low_power_mode_3();
            P3OUT = BIT5;
            __delay_cycles(100000);
            __low_power_mode_3();
            P3OUT = BIT6;
            __delay_cycles(100000);
            __low_power_mode_3();
            P3OUT = BIT7;
            __delay_cycles(100000);
            __low_power_mode_3();
            counter++;
  }
}
```

5 LABORATORY 4: MSP430FR5739 EXPERIMENTERS DEMO PROGRAM

5.1 PRE-LABORATORY

1. **TMP36 is an external temperature sensor. Briefly explain how this sensor works.**

TMP36 uses the increasing voltage across the base-emitter junction due to increasing temperature to determine the temperature and generates a linear signal with respect to temperature.[10]

2. **How do accelerometer sensors work?**

Piezoeletric effect accelerometers, the most commonly used type of accelerometer, works by detecting voltage generated from microscopic crystal structures that become stressed due to accelerative forces. [11]

3. **Is top of the stack different for push and pull operations?**

Yes. The TOS for a push operation is the next available address that data can be stored at and it is the last address that was written to in a pull operation. [12]

4. **How is the memory cell notation different for stack compared to a regular memory bank?**

The SP usually starts pushing the top address of RAM, ie the stack grows down.[12]

5. **Explain what case A and case B, are when it comes to stack operations.**

Case A and Case B refers to where the SP is pointing before and after a push/pull operation. For Case A, a push operation consists of first storing the source and then updates the SP. A pull operation updates the SP first and then retrieves the data. The reverse occurs with Case B. In a push operation, the SP is updated first before the data is stored and in a pull operation, the data is retrieved first before updating the SP. [12]

6. **What are the RTN notation for push and pop?**

Push:

(SP) <- src

SP <- SP - N

Pull:

SP <- SP + N

Dst <- (SP)

(Depending on if it is CASE A or B) [12]

7. **How is a subroutine executed? Explain the roles of stack and program counter.**

When calling a subroutine, the current content of the PC is pushed onto the stack. After the subroutine is executed, the address on top of the stack (the content of PC before the subroutine) is pushed to the PC and program continues execution where it left. [12]

8. **What is an orthogonal CPU?**

Orthogonal CPUS can use all its registers and addressing modes as operands, except for the immediate mode as a destination. [12]

9. **What are the advantages and disadvantages of using an interrupt?**

Interrupts have the advantage of using less power and less overhead. However, they are susceptible to false triggering of interrupts in noisy environments where it might be better to use polling to get a more accurate functionality. Interrupt driven systems may also have delayed responses to different IRQs which can cause problems with flow control of the program. [13]

10. **What are maskable and non-maskable interrupts?**

Maskable interrupts are interrupts that can be turned off by clearing the GIE flag. Non-maskable interrupts cannot be disabled and are reserved to system critical events. [13]

11. **Explain the 6 steps for servicing interrupts**

- Finish the instruction being executed.
- Save the current PC value and the SR onto the stack.
- Clear the global interrupt enable flag.
- Load the PC with the address of the ISR to be executed.
- Execute the corresponding ISR.
- Restore the PC and any other register that was saved onto the stack before the ISR and continue execution. [13]

12. **What is a brownout in electrical circuit?**

A brownout is a significant drop in the output of the power supply. [13]

5.2 In-Laboratory

5.2.1 Question 1: Parts of code responsible for capturing temperature:

```
void Mode4(void)
{
  // One time initialization of header and footer transmit package
  TX_Buffer[0] = 0xFA;
  TX_Buffer[6] = 0xFE;

  // variable initialization
  ADCTemp = 0;
  temp = 0;
  WriteCounter = 0;
  active = 1;
  ULPBreakSync = 0;
  counter = 0;

  // One time setup and calibration
  SetupThermistor();
  CalValue = CalibrateADC();
  while((mode == TEMP_MEAS) && (UserInput == 0))
          {
          // Take 1 ADC Sample
          TakeADCMeas();
          if (ADCResult >= CalValue)
          {
          temp = DOWN;
          ADCTemp = ADCResult - CalValue;
          }
          else
          {
          temp = UP;
          ADCTemp = CalValue - ADCResult;
          }

          if((ULP==1) && (UserInput == 0))
          {
          // P3.4- P3.7 are set as output, low
          P3OUT &= ~(BIT4 + BIT5 + BIT6 + BIT7);
   P3DIR |= BIT4 + BIT5 + BIT6 + BIT7;
          // PJ.0,1,2,3 are set as output, low
          PJOUT &= ~(BIT0 + BIT1 + BIT2 + BIT3);
   PJDIR |= BIT0 + BIT1 + BIT2 + BIT3;
          // Transmit break packet for GUI freeze
          if(!(ULPBreakSync))
          {
          TXBreak(mode);
          ULPBreakSync++;
          }

          }
          if((ULP==0) && (UserInput == 0))
```

```
        {
        ULPBreakSync = 0;
        WriteCounter++;
        if(WriteCounter > 300)
        {
        LEDSequence(ADCTemp,temp);
        // Every 300 samples
        // Transmit 7 Bytes
        // Prepare mode-specific data
        // Standard header and footer
        WriteCounter = 0;
        TX_Buffer[1] = 0x04;
        TX_Buffer[2] = counter;
        TX_Buffer[3] = 0x00;
        TX_Buffer[4] = 0x00;
        TX_Buffer[5] = 0x00;
        TXData();
        }
        }
        }
        // turn off Thermistor bridge for low power
        ShutDownTherm();

}
```

The section of the program responsible for switching between the different modes of operation is this switch case:

```
switch(mode)
  {
   case MAX_FRAM_WRITE:
    Mode1();
    break;

   case SLOW_FRAM_WRITE:
    Mode2();
    break;

   case ACCEL_MEAS:
    Mode3();
    break;

   case TEMP_MEAS:
    Mode4();
    break;

   default:
    // This is not a valid mode
    // Blink LED1 to indicate invalid entry
    // Switch S2 was pressed w/o mode select
```

```
      while((mode > 0x08)&& (UserInput == 0))
       {
         P3OUT ^= BIT7;
         LongDelay();
       }
    break;
    }
```

By changing the value of “mode” in the switch case to its symbolic equivalent ACCEL_MEAS or TEMP_MEAS, we can change it to measure acceleration or measure temperature.

5.2.2 Question 2: Parts of code responsible for Accelerometer use:

```
void Mode3(void)
{

  // One time initialization of header and footer transmit package
  TX_Buffer[0] = 0xFA;
  TX_Buffer[6] = 0xFE;

  // variable initialization
  active = 1;
  ADCTemp = 0;
  temp = 0;
  WriteCounter = 0;
  ULPBreakSync = 0;
  counter = 0;

  // One time setup and calibration
  SetupAccel();
  CalValue = CalibrateADC();
  while ((mode == ACCEL_MEAS) && (UserInput == 0))
          {
          // Take 1 ADC Sample
          TakeADCMeas();
          if (ADCResult >= CalValue)
          {
          temp = DOWN;
          ADCTemp = ADCResult - CalValue;
          }
          else
          {
          temp = UP;
          ADCTemp = CalValue - ADCResult;
          }
          if((ULP==1) && (UserInput == 0))
          {
          // P3.4- P3.7 are set as output, low
          P3OUT &= ~(BIT4 + BIT5 + BIT6 + BIT7);
          P3DIR |= BIT4 + BIT5 + BIT6 + BIT7;
          // PJ.0,1,2,3 are set as output, low
```

```
        PJOUT &= ~(BIT0 + BIT1 + BIT2 + BIT3);
        PJDIR |= BIT0 + BIT1 + BIT2 + BIT3;
        // Transmit break packet for GUI freeze
        if(!(ULPBreakSync))
        {
        TXBreak(mode);
        ULPBreakSync++;
        }
        }
        if((ULP==0) && (UserInput == 0))
        {
        ULPBreakSync = 0;
        WriteCounter++;
        if(WriteCounter > 300)
        {
        LEDSequence(ADCTemp,temp);
        // Every 300 samples
        // Transmit 7 Bytes
        // Prepare mode-specific data
        // Standard header and footer
        WriteCounter = 0;
        TX_Buffer[1] = 0x03;
        TX_Buffer[2] = counter;
        TX_Buffer[3] = 0x00;
        TX_Buffer[4] = 0x00;
        TX_Buffer[5] = 0x00;
        TXData();
        }
        }
        }
        // end while() loop
        // turn off Accelerometer for low power
        ShutDownAccel();
}
```

5.2.3 Question 3:

In order to turn all 8 of the LEDS one, we need the write to the output registers to specify which ones would be on and off. We found the pattern from the incomplete LED Menu and used it to fill the rest of the look up table. 0x80, 0xC0, 0xE0, and 0xF0 are 1000000, 11000000, 11100000, 11110000 in binary respectively. Using this we filled the table with hex equivalent of turning the other the other LEDS on as seen from below:

const unsigned char LED_Menu[] = {0x80,0xC0,0xE0,0xF0, 0xF8, 0xFC, 0xFE, 0xFF};

The ISR below is responsible for dealing with the push buttons. The key details are when the PJOUT and the P3OUT registers are being written to by values from the look up table we made. We used the MSP430 board guide to determine what registers were being used for turning on the internal LEDS and these were found to be PJOUT and P3OUT. When the push button is pressed, SwitchCounter is incremented by one which shifts the index of the look up to a value with “an extra one” ie. the next LED would be turned on. An if-statement is used to prevent the index from going out of bounds of the array and causing a segmentation fault. When SwitchCounter reaches 7 (all the LEDS are turned on), we reset the value back to 0 (0x80 in the look up table which only turns on 1 LED). The reversed is used when turning the LEDS back down. Decrementing the index of the array would cause a value that has one less one which effectively turns an LED off when written to the output registers. For example, going from 0xFE to 0xFC is equivalent to going from 11111110 to 11111100 with the 1s representing which LEDS are turned on.

An alternative way to do this would be through bit manipulation. A bit mask which shifts position among the 8 bits could be used to turn an LED either on or off. The value in the register would be initialised to 0000 0000. Every time button 1 would be pushed, the bit mask is shifted to the right by one and then OR'd with the output registers which effectively adds 1 to that position and turn on that LED. For example, say the current value of the register is 1000 0000 and the bit mask is currently 1000 0000. Shifting this by one would result in 0100 0000 and the OR with 1000 000 would result in 1100 0000, turning the second LED on. Similar logic can be used to turn the LEDS off except the bit mask would be shifted to the left and an XOR would be applied instead. A conditional can be used to check when all the LEDS are either turned on or off to keep the mask within range.

Relevant code:

```
// Interrupt Service Routines
/*************************************************************************//**
 * @brief  Port 4 ISR for Switch Press Detect
 *
 * @param  none
 *
 * @return none
 **************************************************************************/
#pragma vector=PORT4_VECTOR
__interrupt void Port_4(void)
{
  // Clear all LEDs
  PJOUT &= ~(BIT0 +BIT1+BIT2+BIT3);
  P3OUT &= ~(BIT4 +BIT5+BIT6+BIT7);

  switch(__even_in_range(P4IV,P4IV_P4IFG1))
        {
        case P4IV_P4IFG0:              // Button 1
        DisableSwitches();
        Switch2Pressed = 0;
        UserInput = 1;
        P4IFG &= ~BIT0;                         // Clear P4.0 IFG
        PJOUT = LED_Menu[SwitchCounter];
        P3OUT = LED_Menu[SwitchCounter];
        SwitchCounter++; //turn on next led
        if (SwitchCounter>7) //reset
        {
                SwitchCounter =0;
        }
        StartDebounceTimer(0);                  // Reenable switches after debounce
        break;

        case P4IV_P4IFG1:              // Button 2
        DisableSwitches();
        Switch2Pressed = 1;
        UserInput = 1;
        P4IFG &= ~BIT0;                         // Clear P4.0 IFG
        SwitchCounter--; //go back down
        if (SwitchCounter<0)
        {
                SwitchCounter =7;
        }
        PJOUT = LED_Menu[SwitchCounter];
        P3OUT = LED_Menu[SwitchCounter];
        StartDebounceTimer(0);                  // Reenable switches after debounce
        break;

        default:
        break;
        }
}
```

6 Laboratory 5: MSP430 Assembly

6.1 Pre-Laboratory

1. **What are Mnemonics in Assembly language?**

Mnemonics are names given to the opcode assembly language. Mnemonics end with a suffix of ".b" or ".w" to differentiate between operands that are byte and word sized. No suffix is by default equivalent to ".w" [9]

2. **What is the assembly instruction format for MSP430?**

There are two components to the assembly instruction format with an optional components:

1. The mnemonic as discussed in previously
2. The operands: usually the "Source" and "Destination" of an instruction. An instruction can have two, one or no operands, depending on the needs and purpose of the instruction. They are written in their appropriate addressing mode. [9]
3. Optional label and comment at the start and end respectively

3. **What is the purpose of directives in assembly language? Are they converted to machine code?**

The purpose of directives is to organise the code and create meaningful names for otherwise not so obviously understood code. They are not converted to machine language. They are instructions for the compiler and the assembler which can help optimise the program. [9]

4. **What is the benefit of having labels in assembly language?**

Labels are useful for replacing memory addresses and constants that are difficult to make sense of without previous knowledge of their meaning. They make your code more readable. For instance, they could take the value of the reset vector or interrupt vector, allowing you to use the label in your code rather than the actual value of these vectors, making your code more readable. [9]

5. **What is the difference between core and emulated instructions?**

Core instructions are hardwired commands into the architecture that have machine-specific opcode in machine language syntax.

Emulated instructions make use of labels and are not hard-wired, making them easier to read, make sense of, remember and use. They are translated into their core instruction equivalent when compiled.[12]

6. **The most significant byte of a register is not available in byte instructions. How can we access them if needs be?**

In order to access or change the most significant byte in the register, the "swpb Rn" instruction can be used before or after a byte instruction. [12]

7. **What is the main difference between Absolute Source File and Relocatable Source File?**

In relocatable code, we use segment directives to define or work memory segments, whose starting address is set by the linker. In contrast to this, in absolute source files, the program location counter (PLC) used by the linker to load each instruction and the data at the appropriate address, is controlled with ORG directive and is set with the exact memory map in mind. It cannot be relocated to different architectures unlike relocatable code.[14]

6.2 In-Laboratory

6.2.1 Question 2:

Test.asm:

- Absolute Code
- Keeps LED on while value of `COUNTER` is not equal to 0
- Using `Byte` operations
- Blinks external LEDs (P1OUT)

Test2Blink.asm

- Relocatable Code
- Keeps LED on while value of `COUNTER` is not equal to 0
- Using `Word` operations
- Blinks internal LEDs (PJOUT)

Test2.asm

- Relocatable code
- Keeps LED on while value of `COUNTER` is not equal to 0
- Using `Word` operations
- Blinks external LEDs (P1OUT)

6.2.2 Question 4:

To enable turning the LEDS on and off, the 4 LEDS on the PJDIR register had to be written to with all 1's in order to output to them with each bit representing an LED. This can be seen in the initialisation below where we use bis to set the bits in the register to 1111 (8+4+2+1).

In order to switch between odd and even LEDS, PJOUT was initialised to 1010 which would mean that the first and third LED would be set to high whilst the second and fourth would be set to low. A loop was set by using a counter variable which would be decremented every clock cycle until it hits 0 in which it goes back to the mainloop to do another action. This action was simply inverting the bits in the PJOUT which cause its value to go from 1010 to 0101 and effectively blinks and alternates between the odd and even LEDS.

Relevant code:

```
#include "msp430.h"
LED0   EQU 01h                    ;LED at PJ.0
LED1   EQU 02h                    ;at PJ1.0
LED2   EQU 04h                    ;PJ2.0
LED3   EQU 08h                    ;PJ3.0
DELAY   EQU 50000
#define COUNTER R15               ;R15 as counter
;-------------------------------------------------------------------------;
        RSEG CSTACK               ;Create a stack in RAM
        RSEG CODE                 ;Program goes to code
;-------------------------------------------------------------------------;
RESET   mov.w #SFE(CSTACK), SP           ;Initialize stack
StopWDT mov.w #WDTPW+WDTHOLD,&WDTCTL          ;Stop WDT
```

```
        bis.w #1, &PJDIR                ;PJ.0 output
        bis.w #2, &PJDIR
        bis.w #4, &PJDIR
        bis.w #8, &PJDIR                ;setting up outputs for the rest of leds

        ;PJ0-3
        ;alternate between 0,2 and 1,3
        bis.w #1, &PJOUT
        bis.w #4, &PJOUT

Mainloop inv &PJOUT                     ;Alternating between even & odd
Wait    mov.w #DELAY,COUNTER                    ;Delay to counter
L1      dec.w COUNTER                   ;Decrement counter
        jnz L1                  ;Delay Over?

        jmp Mainloop                    ;Again

;-------------------------------------------------------------------------------
                ;Interrupt Vectors
;-------------------------------------------------------------------------------
        RSEG RESET                      ;Address for Reset Vector
        DW RESET
        END
```

6.2.3 Question 5:

The task was to turn the LEDs on from LED1 to LED4 whilst keeping the previous ones and turn off going backwards. This was achieved by incrementally setting the bits in the PJOUT register by adding powers of 2 (ie. the next position of the LED) using the bis instruction which does a logical OR. It is initially set to 0001 (#1), a delay is waited then OR'd with #2 (0010) which results in the contents of the register to be 0011 (2 LEDS are turned on) as an OR operation is basically an add. This is repeated until all of them are on (PJOUT has 1111). Once all of them are on, we turn them off backwards using an XOR operation with the bit we want to turn off by going from #8 all the way down to #1. The XOR operation results in turning off that LED at that location (1 XOR 1 = 0). Once this is done, the whole loop jumps back to Mainloop which starts whole process all over again.

Relevant code:

```
RESET   mov.w #SFE(CSTACK), SP          ;Initialize stack
StopWDT mov.w #WDTPW+WDTHOLD,&WDTCTL            ;Stop WDT
        bis.w #1, &PJDIR                ;PJ.0 output
        bis.w #2, &PJDIR
        bis.w #4, &PJDIR
        bis.w #8, &PJDIR                ;setting up outputs for the rest of leds
Mainloop

        bis.w #1, &PJOUT

Wait1   mov.w #DELAY,COUNTER            ;Delay to counter
L1      dec.w COUNTER                   ;Decrement counter
        jnz L1                  ;Delay Over?

        bis.w #2, &PJOUT
```

```
Wait2  mov.w #DELAY,COUNTER          ;Delay to counter
L2     dec.w COUNTER            ;Decrement counter
       jnz L2              ;Delay Over?

       bis.w #4, &PJOUT

Wait3   mov.w #DELAY,COUNTER         ;Delay to counter
L3     dec.w COUNTER            ;Decrement counter
       jnz L3              ;Delay Over?

       bis.w #8, &PJOUT

Wait4   mov.w #DELAY,COUNTER         ;Delay to counter
L4     dec.w COUNTER            ;Decrement counter
       jnz L4              ;Delay Over?

       xor.w #8, &PJOUT

Wait5   mov.w #DELAY,COUNTER         ;Delay to counter
L5     dec.w COUNTER            ;Decrement counter
       jnz L5              ;Delay Over?

       xor.w #4, &PJOUT

Wait6   mov.w #DELAY,COUNTER         ;Delay to counter
L6     dec.w COUNTER            ;Decrement counter
       jnz L6              ;Delay Over?

       xor.w #2, &PJOUT

Wait7   mov.w #DELAY,COUNTER         ;Delay to counter
L7     dec.w COUNTER            ;Decrement counter
       jnz L7              ;Delay Over?

       xor.w #1, &PJOUT

Wait8   mov.w #DELAY,COUNTER         ;Delay to counter
L8     dec.w COUNTER            ;Decrement counter
       jnz L8              ;Delay Over?

       jmp Mainloop
```

Q6:

For this task, we were required to blink an LED three times and move on to the next one, going from LED1 to LED4 and then back again. In order to simplify this whilst being able to demonstrate proof of concept, we only did it with 2 LEDS. To achieve this, we would write a HIGH to the bit (using bis) that represents the position in the LED, wait out a delay and turn it off by doing an XOR operation to turn it back off. This process would effectively blink the LED once. By repeating this bit of code three times, we managed to make it blink three times.

The same process is done for the next position of the LED by using the OR (bis) and XOR operation on the register with the literal #2 (0010) which blink the second LED. To go back down, we would repeat the process for first LED and then jump all the way back to the Mainloop to start the process all over again.

Although this process of hard coding the order of events works, it is rather cluttered and difficult to understand. To simplify this, we could use more variables using other registers which would represent how many times an LED has blinked and use it to create a subloop in which a particular LED would blink 3 times before exiting out of the loop to move on to the next LED. Another improvement that we can do is bit shift the data in the PJOUT register instead of using an OR or XOR operation with the next power of 2.This conceptually also makes more sense as this shift equates to a physical shift of the active LED.

Relevant code:

```
f
RESET   mov.w #SFE(CSTACK), SP          ;Initialize stack
StopWDT mov.w #WDTPW+WDTHOLD,&WDTCTL           ;Stop WDT
        bis.w #1, &PJDIR                ;PJ.0 output
        bis.w #2, &PJDIR
        bis.w #4, &PJDIR
        bis.w #8, &PJDIR                ;setting up outputs for the rest of leds
Mainloop

        bis.w #1, &PJOUT

Wait1   mov.w #DELAY,COUNTER            ;Delay to counter
L1      dec.w COUNTER                   ;Decrement counter
        jnz L1                  ;Delay Over?

        xor.w #1, &PJOUT

Wait2   mov.w #DELAY,COUNTER            ;Delay to counter
L2      dec.w COUNTER                   ;Decrement counter
        jnz L2                  ;Delay Over?

        bis.w #1, &PJOUT

Wait3   mov.w #DELAY,COUNTER            ;Delay to counter
L3      dec.w COUNTER                   ;Decrement counter
        jnz L3                  ;Delay Over?

        xor.w #1, &PJOUT

Wait4   mov.w #DELAY,COUNTER            ;Delay to counter
L4      dec.w COUNTER                   ;Decrement counter
        jnz L4                  ;Delay Over?

        bis.w #1, &PJOUT

Wait5   mov.w #DELAY,COUNTER            ;Delay to counter
L5      dec.w COUNTER                   ;Decrement counter
        jnz L5                  ;Delay Over?

        xor.w #1, &PJOUT

Wait6   mov.w #DELAY,COUNTER            ;Delay to counter
```

```
L6      dec.w COUNTER                ;Decrement counter
        jnz L6                  ;Delay Over?

        bis.w #2, &PJOUT

Wait7   mov.w #DELAY,COUNTER            ;Delay to counter
L7      dec.w COUNTER                ;Decrement counter
        jnz L7                  ;Delay Over?

        xor.w #2, &PJOUT

Wait8    mov.w #DELAY,COUNTER           ;Delay to counter
L8      dec.w COUNTER                ;Decrement counter
        jnz L8                  ;Delay Over?

        bis.w #2, &PJOUT

Wait9   mov.w #DELAY,COUNTER            ;Delay to counter
L9      dec.w COUNTER                ;Decrement counter
        jnz L9                  ;Delay Over?

        xor.w #2, &PJOUT

Wait10   mov.w #DELAY,COUNTER          ;Delay to counter
L10     dec.w COUNTER                ;Decrement counter
        jnz L10                 ;Delay Over?

        bis.w #2, &PJOUT

Wait11mov.w #DELAY,COUNTER            ;Delay to counter
L11     dec.w COUNTER                ;Decrement counter
        jnz L11                 ;Delay Over?

        xor.w #2, &PJOUT

Wait12   mov.w #DELAY,COUNTER          ;Delay to counter
L12     dec.w COUNTER                ;Decrement counter
        jnz L12                 ;Delay Over?

        jmp Mainloop
```

7 REFERENCES

[1] KHAKSAR, S. (2019). *MICROCOMPUTER ORGANIZATION : OVERVIEW*. Available at: https://lms.curtin.edu.au/bbcswebdav/pid-7106948-dt-content-rid-36933128_1/courses/2019_2_CMPE2003_V1_L1_A1_INT_670666/2019_2_CMPE2003_V1_L1_A1_INT_670666_ImportedContent_20190724125745/2a%20Microcomputer%20organization%20Overview.pdf [Accessed 23 Oct. 2019].

[2] Stack Overflow. (2019). *What do .c and .h file extensions mean to C?*. [online] Available at: https://stackoverflow.com/questions/1695224/what-do-c-and-h-file-extensions-mean-to-c [Accessed 23 Oct. 2019].

[3] KHAKSAR, S. (2019). MICROCOMPUTER ORGANIZATION: REGISTER DETAILS. Available at: https://lms.curtin.edu.au/bbcswebdav/pid-7106944-dt-content-rid-37126657_1/courses/2019_2_CMPE2003_V1_L1_A1_INT_670666/3a%20Microcomputer%20Organization%20Register%20details%20and%20System%20Buses%281%29.pdf [Accessed 23 Oct. 2019].

[4] KHAKSAR, S. (2019). MICROCOMPUTER ORGANIZATION: BUSES. Available at: https://lms.curtin.edu.au/bbcswebdav/pid-7106944-dt-content-rid-37126658_1/courses/2019_2_CMPE2003_V1_L1_A1_INT_670666/3b%20Microcomputer%20Organization%20Buses.pdf [Accessed 23 Oct. 2019].

[5] GeeksforGeeks. (2019). *Difference between Volatile Memory and Non-Volatile Memory - GeeksforGeeks*. [online] Available at: https://www.geeksforgeeks.org/difference-between-volatile-memory-and-non-volatile-memory/ [Accessed 23 Oct. 2019].

[6] Differencebetween.net. (2019). *Difference between Von Neumann and Harvard Architecture | Difference Between*. [online] Available at: http://www.differencebetween.net/technology/difference-between-von-neumann-and-harvard-architecture/ [Accessed 23 Oct. 2019].

[7] I2C Bus. (2019). *Slave - I2C Bus*. [online] Available at: https://www.i2c-bus.org/slave/ [Accessed 23 Oct. 2019].

[8] KHAKSAR, S. (2019). MICROCOMPUTER ORGANIZATION: I/O SUBSYSTEM ORGANIZATION Available at: https://lms.curtin.edu.au/bbcswebdav/pid-7106938-dt-content-rid-37357126_1/courses/2019_2_CMPE2003_V1_L1_A1_INT_670666/Microcomputer%20Organization%20IO%20Subsystem%20Organization%281%29.pdf

[9] KHAKSAR, S. (2019). MICROCOMPUTER ORGANIZATION: CPU INSTRUCTION SETS (PART1) Available at: https://lms.curtin.edu.au/bbcswebdav/pid-7106935-dt-content-rid-37467108_1/courses/2019_2_CMPE2003_V1_L1_A1_INT_670666/Microcomputer%20Organization%20CPU%20Instruction%20sets%20_Part1_%281%29.pdf

[10] Adafruit Learning System. (2019). *TMP36 Temperature Sensor*. [online] Available at: https://learn.adafruit.com/tmp36-temperature-sensor [Accessed 23 Oct. 2019].

[11] En.wikipedia.org. (2019). *Accelerometer*. [online] Available at: https://en.wikipedia.org/wiki/Accelerometer [Accessed 23 Oct. 2019].

[12] KHAKSAR, S. (2019). MICROCOMPUTER ORGANIZATION: CPU INSTRUCTION SETS (PART2) Available at: https://lms.curtin.edu.au/bbcswebdav/pid-7106932-dt-content-rid-37548072_1/courses/2019_2_CMPE2003_V1_L1_A1_INT_670666/Microcomputer%20Organization%20CPU%20Instruction%20sets%20_Part2_%281%29.pdf

[13] KHAKSAR, S. (2019). INTRODUCTION TO INTERRUPTS AND POLLING. Available at: https://lms.curtin.edu.au/bbcswebdav/pid-7106930-dt-content-rid-36933103_1/courses/2019_2_CMPE2003_V1_L1_A1_INT_670666/2019_2_CMPE2003_V1_L1_A1_INT_670666_ImportedContent_20190724125745/Introduction%20to%20Interrupts%20and%20Polling.pdf

[14] KHAKSAR, S. (2019). MICROCOMPUTER ORGANIZATION: CPU INSTRUCTION SETS (PART3) https://lms.curtin.edu.au/bbcswebdav/pid-7106922-dt-content-rid-38097337_1/courses/2019_2_CMPE2003_V1_L1_A1_INT_670666/Assembly%20language%20Part%203%281%29.pdf