2019

Microcomputers: Logbook

21/10/19 - CMPE2003

AHMAD ALLAHHAM

Lab 1

Pre-labs:

Q1) Program memory: stores the program in the form of instructions
Data Memory: stores the data that is operated on by the program

Q2) MPU: contains cpu, optimised for external data manipulation & storage
MCU: less complex than MPU, optimised for use when little external devices are connected to it

Q3) CISC (Complex Instruction Set Computing):
- Variable commands, bit sizes & multiple clocks
- Aim is to simplify the program (hardware more complex)

RISC (Reduced Instruction Set Computing):
- simple instructions
- Aim is to simplify hardware structure

Q4) Hardware Model: focus on hardware characteristics (understanding physical capabilities & outer input/interaction)
Software Model: focus on instruction set & syntax

Q5) main purpose of ALU is to do arithmetic operations, as well as logic operations

Q6) Infinite loops are very resource intensive & consume a lot of power

Q7) using interrupts, which interrupt the operation of the program for sometime then resume operations when a particular flag is set

Q8) to do compler operations & manipulate data

Q9) the resources allocated for a particular program

Q10) ".c" files are implementations of functions declared in ".h" files

Q11) to have more control of the CPU while avoiding as much overhead as possible.

Lab 2

Q 1) the program counter provides the address of the instruction to be fetched from memory
Q 2) the push operation is writing into the stack while the pull operation is reading from the stack
Q 3) yes since the sign of the result is different from the sign of the operands
Q 4) C: carry, Z: zero, N: negative
Q 5) we can see which SR flags are triggered after the subtraction of two numbers to see equal, or one is bigger than the other
Q 6) 16^3 locations
Q 7)

address bus: the lines used to transport address information
data bus: set of lines used to transport data and instructions to and from the cpu
control bus: he lines that carry signals that regulate system acitivity

Q 8)

volatile memory: rips temporary and is erased after the power is turned off
non volatile memory: remains in memory after the power is cut off

Q 9) FRAM:

can be written to during program execution
requires low energy to operate (only when loading and executing operations)
the writing is speed is comparable to DRAMS

Q 10)
Q 11)

Von Neumann: Program and Data memory share the same system buses
Harvard: they have different system buses

Q 12) a look table is an initialised array that contains precalculated information. They are defined during he program's initialisation stage
Q 13) the master is the one that drives the clock line while the slaves follow the convention defined by the master
Q 14) ACK and NACK

Lab 3

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

Special timer used as a safety device. It resets if it does not receive a signal generated by the program every X time units.

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. Its disadvantage is that In modern technology, wires can electromagnetically interfere with each other, also the timing of signals must be the same which can be difficult with faster connections.

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.

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

Buffered Input: They do not hold input data and the CPU can only read input present at that instant

Latched Input: Hold the data until it is read by the CPU, after which they are usually cleared and ready for another input

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

OpCode: The most significant one, also known as the operating code which defines the operation to be performed.

Operand: Which is the data or registers used in the instruction

Addressing mode: That is the way in which an operand defines the data, for example, an RTN expression may have Rn or (Rn). Operand in both is Rn but the data or datum is found in different places.

5. What is contained in the source file?

assembly instructions

Directives: These are used to control the assembly process and not executed by the CPU

Instructions

Comments

Macro directives: Allow to group a set of assembly instructions into only one line. This simplifies the source for human view

6. What is the purpose of the linker?

the linker combines the file with previously assembled object files and libraries to generate the final executable file

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

Data Transfer

Arithmetic-Logic

Program Control

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

The contents of the program counter change when a jump instruction is executed so that the flow of the program is altered

Q 4.2

Enabling the interrupt

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

Stop the Watch dog

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

Low Power Mode

```
__low_power_mode_3();
```

Q4.3: 8 LEDs turn on and off

```
// Introduction to the MSP430FR5739 system
// As provided this program will pulse a blue LED on the board
// Removing the necessary line comment will set up two LEDs to toggle between them
// using a software delay to make the display visible
//
// A better setup is to use a timer and interrupts
// Comment out the __delay_cycles function
// Restore all the other lines to facilitate the timer and interrupt operations
// This uses a low power mode to wait for an interrupt to occur.
// CAM 20130213
// ********************************************************************
#include "msp430fr5739.h"
unsigned int counter = 0;
void main(void)
{
  WDTCTL = WDTPW + WDTHOLD;                     // Stop WDT

 // Set up a timer and enable the interrupt
  TA1CCR0 = 0x4000;
  TA1CTL = TASSEL_1 + MC_1 + TACLR;
  TA1CCTL0 = CCIE;                      // interrupt enabled

// 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
 }
}
// The interrupt service routine
 #pragma vector = TIMER1_A0_VECTOR
__interrupt void Timer1_A0_ISR(void)
{
 __low_power_mode_off_on_exit();
}
```

Q4.4: Turning on external LED (BLINK WITH OTHER INTERNAL LEDs)

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

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

Q4.5: Turning on in order and reverse

```
// Introduction to the MSP430FR5739 system
// As provided this program will pulse a blue LED on the board
// Removing the necessary line comment will set up two LEDs to toggle between them
// using a software delay to make the display visible
//
// A better setup is to use a timer and interrupts
// Comment out the __delay_cycles function
// Restore all the other lines to facilitate the timer and interrupt operations
// This uses a low power mode to wait for an interrupt to occur.
// CAM 20130213
// ******************************************************************
```

```
#include "msp430fr5739.h"
unsigned int counter = 0;
void main(void)
{
  WDTCTL = WDTPW + WDTHOLD;                     // Stop WDT

 // Set up a timer and enable the interrupt
  TA1CCR0 = 0x4000;
  TA1CTL = TASSEL_1 + MC_1 + TACLR;
  TA1CCTL0 = CCIE;                      // interrupt enabled

// 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++;
  }
}
// The interrupt service routine
 #pragma vector = TIMER1_A0_VECTOR
```

```
__interrupt void Timer1_A0_ISR(void)
{
 __low_power_mode_off_on_exit();
}
```

Lab 4

Pre-lab

1. TMP36 is an external temperature sensor. Briefly explain how this sensor works.
TMP36 use the fact as temperature increases, the voltage across a diode increases at a known rate. (Technically, this is actually the voltage drop between the base and emitter - the Vbe - of a transistor.) By precisely amplifying the voltage change, it is easy to generate an analog signal that is directly proportional to temperature.

2. How do accelerometer sensors work?
Accelerometers measure proper acceleration, meaning acceleration due to any force, including the force of gravity, unlike other coordinate acceleration, being the acceleration in a fixed coordinate system.

3. Is top of the stack different for push and pull operations?
No, the top of the stack is the same for push and pull operations since it applies the LIFO method (last in, first out)

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.

5. Explain what case A and case B, are when it comes to stack operations.
- Pushing when SP points at TOS
- Popping when SP Point at TOS

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

7. How is a subroutine executed? Explain the roles of stack and program counter.
When reaching a subroutine, the contents of the PC is pushed onto the stack, while maintaining the address for the next instruction. After the subroutine is executed, the next instruction is called and execution is resumed.

8. What is an orthogonal CPU?
A CPU is said to orthogonal if all its registers and addressing modes can be used as operands, except for the immediate mode as a destination

9. What are the advantages and disadvantages of using an interrupt?
- Advantages:
    - Preferred when working with low power mode
    - Using interrupts leads to less CPU cycles and less power consumption
- Disadvantages:
    - susceptible to false triggering of interrupts so in noisy environments it might be better to use polling to get a more accurate functionality
    - If not handled properly, interrupt driven systems may have delayed responses to different IRQs

10. What are maskable and non-maskable interrupts?
Maskable interrupts are interrupts that can be turned off (modified), non-maskable interrupts cannot be turned off (modified)

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 in Step

12. What is a brown-out in electrical circuit?

Pre-lab

1. TMP36 is an external temperature sensor. Briefly explain how this sensor works.
TMP36 use the fact as temperature increases, the voltage across a diode increases at a known rate. (Technically, this is actually the voltage drop between the base and emitter - the Vbe - of a transistor.) By precisely amplifying the voltage change, it is easy to generate an analog signal that is directly proportional to temperature.

2. How do accelerometer sensors work?
Accelerometers measure proper acceleration, meaning acceleration due to any force, including the force of gravity, unlike other coordinate acceleration, being the acceleration in a fixed coordinate system.

3. Is top of the stack different for push and pull operations?
No, the top of the stack is the same for push and pull operations since it applies the LIFO method (last in, first out)

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.

5. Explain what case A and case B, are when it comes to stack operations.
- Pushing when SP points at TOS
- Popping when SP Point at TOS

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

7. How is a subroutine executed? Explain the roles of stack and program counter.
When reaching a subroutine, the contents of the PC is pushed onto the stack, while maintaining the address for the next instruction. After the subroutine is executed, the next instruction is called and execution is resumed.

8. What is an orthogonal CPU?
A CPU is said to orthogonal if all its registers and addressing modes can be used as operands, except for the immediate mode as a destination

9. What are the advantages and disadvantages of using an interrupt?
- Advantages:
    - Preferred when working with low power mode
    - Using interrupts leads to less CPU cycles and less power consumption
- Disadvantages:
    - susceptible to false triggering of interrupts so in noisy environments it might be better to use polling to get a more accurate functionality
    - If not handled properly, interrupt driven systems may have delayed responses to different IRQs

10. What are maskable and non-maskable interrupts?
Maskable interrupts are interrupts that can be turned off (modified), non-maskable interrupts cannot be turned off (modified)

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 in Step

12. What is a brown-out in electrical circuit?

Lab

Q1: 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();

}
```

Q2: 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();
}
```

Q3:

```
/*******************************************************************************
 *
 * main.c
 * User Experience Code for the MSP-EXP430FR5739
 *
 *
 * Copyright (C) 2010 Texas Instruments Incorporated - http://www.ti.com/
 *
 *
 *  Redistribution and use in source and binary forms, with or without
 *  modification, are permitted provided that the following conditions
 *  are met:
 *
 *        Redistributions of source code must retain the above copyright
 *        notice, this list of conditions and the following disclaimer.
 *
 *        Redistributions in binary form must reproduce the above copyright
 *        notice, this list of conditions and the following disclaimer in the
 *        documentation and/or other materials provided with the
 *        distribution.
 *
 *        Neither the name of Texas Instruments Incorporated nor the names of
 *        its contributors may be used to endorse or promote products derived
 *        from this software without specific prior written permission.
 *
 *  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
 *  "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
 *  LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
 *  A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
 *  OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
 *  SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
 *  LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
 *  DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
 *  THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
 *  (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
 *  OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 *
 * Created: Version 1.0 04/13/2011
 *        Version 1.1 05/11/2011
 *        Version 1.2 08/31/2011
 *
 *******************************************************************************/
```

```
#include "msp430fr5739.h"
#include "FR_EXP.h"
const unsigned char LED_Menu[] = {0x80,0xC0,0xE0,0xF0, 0xF8, 0xFC, 0xFE, 0xFF};
// These golabal variables are used in the ISRs and in FR_EXP.c
volatile unsigned char mode = 0;
volatile unsigned char UserInput = 0;
volatile unsigned char ULP =0;
volatile unsigned int *FRAMPtr = 0;
volatile unsigned char active = 0;
volatile unsigned char SwitchCounter=0;
volatile unsigned char Switch1Pressed=0;
volatile unsigned char Switch2Pressed=0;
volatile unsigned int ADCResult = 0;
void main(void)
{
  WDTCTL = WDTPW + WDTHOLD;                     // Stop WDT
  SystemInit();                             // Init the Board
  StartUpSequence();                        // Light up LEDs

  // Setup ADC data storage pointer for Modes 3&4
  FRAMPtr = (unsigned int *) ADC_START_ADD;

  while(1)
  {
        // Variable initialization
        active = 0;
        Switch2Pressed = 0;
        ULP = 0;
        // Wait in LPM4 for user input
        __bis_SR_register(LPM4_bits + GIE);         // Enter LPM4 w/interrupt
        __no_operation();                   // For debugger

        // Wake up from LPM because user has entered a mode
//      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;
//          }
  }
}

// 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];
//      P3OUT = LED_Menu[SwitchCounter];
        SwitchCounter++;
        if (SwitchCounter>7)
        {
        SwitchCounter =0;
//      Switch1Pressed++;
        }
        StartDebounceTimer(0);              // Reenable switches after debounce
        break;

        case P4IV_P4IFG1:           // Button 2
        DisableSwitches();
        Switch2Pressed = 1;
        UserInput = 1;
        P4IFG &= ~BIT0;                     // Clear P4.0 IFG
//      P3OUT = LED_Menu[SwitchCounter];
        SwitchCounter--;
        if (SwitchCounter<0)
        {
        SwitchCounter =7;
//      Switch1Pressed++;
        }
        PJOUT = LED_Menu[SwitchCounter];
        P3OUT = LED_Menu[SwitchCounter];
        StartDebounceTimer(0);              // Reenable switches after debounce
        break;

        default:
        break;
        }
}
/*************************************************************************//**
 * @brief  Timer A0 ISR for MODE2, Slow FRAM writes, 40ms timer
 *
 * @param  none
 *
 * @return none
 ****************************************************************************/
#pragma vector = TIMER0_A0_VECTOR
__interrupt void Timer_A (void)
{
  __bic_SR_register_on_exit(LPM4_bits);
}
```

```
/*************************************************************************//**
 * @brief  ADC10 ISR for MODE3 and MODE4
 *
 * @param  none
 *
 * @return none
 ****************************************************************************/
#pragma vector=ADC10_VECTOR
__interrupt void ADC10_ISR(void)
{
  switch(__even_in_range(ADC10IV,ADC10IV_ADC10IFG))
  {
        case ADC10IV_NONE: break;                 // No interrupt
        case ADC10IV_ADC10OVIFG: break;           // conversion result overflow
        case ADC10IV_ADC10TOVIFG: break;          // conversion time overflow
        case ADC10IV_ADC10HIIFG: break;           // ADC10HI
        case ADC10IV_ADC10LOIFG: break;           // ADC10LO
        case ADC10IV_ADC10INIFG: break;           // ADC10IN
        case ADC10IV_ADC10IFG:
        ADCResult = ADC10MEM0;
        *FRAMPtr = ADCResult;
        FRAMPtr++;
        // Pointer round off, once 0x200 locations are written, the pointer
        // rolls over
        if (FRAMPtr > (unsigned int *)ADC_END_ADD)
                FRAMPtr = (unsigned int *) ADC_START_ADD;
        __bic_SR_register_on_exit(CPUOFF);
        break;                    // Clear CPUOFF bit from 0(SR)
        default: break;
  }
}
/*************************************************************************//**
 * @brief Timer A1 ISR for debounce Timer
 *
 * @param  none
 *
 * @return none
 ****************************************************************************/
#pragma vector = TIMER1_A0_VECTOR
__interrupt void Timer1_A0_ISR(void)
{
  TA1CCTL0 = 0;
  TA1CTL = 0;
  EnableSwitches();
```

Lab 5

Pre-Lab

1. What are Mnemonics in Assembly language?
Mnemonics is the name 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"

2. What is the assembly instruction format for MSP430?
There are two components to the assembly instruction format:

1. The Mnemonics: Discussed in (Q1)
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

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 independent of the assembler and rather belong to the cpu

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.

5. What is the difference between core and emulated instructions?
Core instructions are hardwired commands 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.

6. The most significant byte of a register is not available in byte instructions. How can we access them if needs be?
to access or change the most significant byte in the register, we may use "swpb Rn" before or after the instruction.

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

In absolute source file, 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.
In relocatable codes, we use segment directives to define or work memory segments, whose starting address is specified by the linker.

In-Lab