CPE 325 Quiz 2 Notes
MSP430: 16, 16-bit registers
R0 - Program counter
R1 - Stack pointer (SP)
R2 - Status register (SR)
R3 - Constant generator

1) Make sure you know how to manipulate different ports in BOTH C and Assembly, and what are the functions of different registers (PxDIR, PxOUT, PxIN, etc.)?

- **Port Manipulation:**

  i. Assembly:
  BIS.B #1, &P1DIR
  BIS.B #1, &P1OUT
  ii. C:
  How do you turn a bit off?
  &= **0 -> 0, 1 -> 1 ??**
  How do you toggle a bit?
  ^= **0 -> 1, 1 -> 0**
  How do you turn a bit on?
  |= **0 -> 1, 1 -> 1**

```
P4OUT |= BIT7; // LED1 is ON        // you're setting the P4OUT to BIT7
P4OUT &= ~GREENLED;                 // Turn GREEN LED off
P2DIR &= ~BIT1;                     // Set P2.1 as input for S1 input
P1DIR |= 0x01;                      // Set P1.0 to output direction (0000_0001)
```

- **Register functions:**

- **PxIN** - **input register**, reading it returns the logical values on the pins (determined by the external signals). These registers are read-only and bit value of 0 indicates that the corresponding input is low and bit value of 1 indicates that the input is high.

- **PxOUT** - **output register**, writing it sends the value to the corresponding port pin when the pin is configured for the I/O function with output direction. Bit value of 0 will produce low output voltage and bit value of 1 will produce high output voltage.

- **PxDIR** - **direction register**, configures the direction of the corresponding I/O pins (e.g., P2DIR=0xFC = 111111100b configures bits 1 and 0 of port P2 as input pins and all other pins are outputs).

- **PxSEL** - **selection register**, setting bits in this register allows the user to change the port pin function from the standard digital I/O to its corresponding special function.

MSP430 interfaces the external world predominantly through parallel ports and their default operation is a standard digital input/output (PxSEL=0x00). However, some of the pins have an alternative special function – e.g., they can act as an analog input channel (A0) to the analog-to-digital converter or serial data output of a serial communication interface (TDO). The reference manual specifies special functions for each port pin. They are highly device-specific – developers have to consult the reference manual for the microcontroller they are using.

- **PxREN – enables the pull-up or pull-down resistor configuration** (e.g, P2REN = 0x02 enables the pull-up resistor on P2.1 that is connected in a series with switch SW1 on the MSP-EXP430F5529LP board.)

- Ports P1 and P2 also have the ability to serve as **sources of interrupts** and several registers are associated with this function. These are:
    - **PxIE** – Port x Interrupt Enable register for enabling/disabling interrupts (x=1, 2),
    - **PxIFG** – Port x Interrupt Flag register for tracking pending requests,
    - **PxIES** – Port x Interrupt Edge Select register for selecting type of event that triggers an interrupt – rising edge at the port input (0 -> 1) or falling edge (1 -> 0);
    - **PxIV** – Port x Interrupt Vector Word. All interrupts associated with a single port share a single interrupt service routine. The highest priority enabled pending interrupt request generates a number in the PxIV register. This number can be used by the code in the corresponding interrupt service routine to speed up interrupt processing.

2) How to set a bit to 0 or 1 without affecting other bits (bit masking) in BOTH C and assembly

Bit Masking: Setting a bit to 0 or 1 without affecting any other bit. Set bit in assembly, use the instruction **bis.b**

3) How to interface with the switches and LEDs? What values would they return if they are pressed? What about when they are not pressed?

Pressed	Not Pressed
0	1

-The LEDs can be turned on and off by writing digital 1 or 0 to the appropriate output port registers, respectively. Therefore, in order to turn a LED on, first the I/O port should be set to output direction, and then either a 0 or a 1 should be written to the output register.
-When trying to interface with **LEDs** essentially you need to # define “name” and then the address area. Next, to actually get the output to the right spot you must choose the **output DIR** and set it to your defined variable. Last, to actually turn off and on the LEDs. you need to mess with the pins. There are LEDs on ports P1.0 and P4.7. You do this with &= (off), |= (set) and ^= (toggle).

```c
 *-----------------------------------------------------------------------------
#include <msp430.h>

void main(void)
{
    WDTCTL = WDTPW + WDTHOLD;    // Stop watchdog timer
    P1DIR |= BIT0;               // Set P1.0 to output direction
    P4DIR |= BIT7;               // Set P4.7 to output direction
    P1OUT &= ~BIT0;              // LED2 is OFF
    P4OUT |= BIT7;               // LED1 is ON
    unsigned int i = 0;
    while(1){                          // Infinite loop
        for (i = 0; i < 50000; i++); // Delay 0.5s
                                       // 0.5s on, 0.5s off => 1/(1s) = 1Hz
        P1OUT ^= BIT0;                 // Toggle LED1
        P4OUT ^= BIT7;                 // Toggle LED1
    }
}
```

-To interface with **switches,** similarly you need to define an name and set it to the input and bit area with: S1 P2IN&BIT1.

```c
#include <msp430.h>

#define S1 P2IN&BIT1

void main(void)
{
    WDTCTL = WDTPW + WDTHOLD;              // Stop watchdog timer
    P1DIR |= BIT0;                         // Set P1.0 to output direction (0000_0001)
    P1OUT &= ~BIT0;                        // LED1 is OFF

    P2DIR &= ~BIT1;                        // Set P2.1 as input for S1 input
    P2REN |= BIT1;                         // Enable the Pull-up resistor at P2.1
    P2OUT |= BIT1;                         // Required for proper IO

    unsigned int i = 0;
    for (;;) {                             // Infinite loop
        if ((S1) == 0) {                   // If S1 is pressed
            for (i = 2000; i > 0; i--);    // Debounce ~20 ms
            if ((S1) == 0)
            {
                P1OUT |= BIT0;             // S1 pressed, turn LED1 on
            }
            while ((S1)==0);               // Hang-on as long as S1 pressed
        } else
            P1OUT &= ~BIT0;
    }
}
```

4) How to calculate delays and how they are affected by different components (clock frequency, loop instruction, loop upper and lower limit, etc.)

- MSP430 operates at ~1 MHz. Therefore, time for one second is 1,000,000 clock cycles.

- A for loop delay takes 10 clock cycles to execute a single loop if there is no body of code execution for it.

- (loop_max * 10 cycles) / 1,000,000 cycles = (125 ms / 1000 ms)

- __delay_cycles (500,000) means delay for 500 milliseconds only because we have a 1 MHz operating frequency which operates at 1M cycles per second.

5) How different assembly instructions work (MOV, JMP, JL, BIS, CMP, etc.) and the difference between word operations (.w) and byte operations (.b)? What happens if you perform a byte operation on a word or vice versa?

- **Word Operations:**
  <u>Example</u>: mov.w #myStr, R4
  Essentially modifying and working with the whole word, not just the last two bytes.

- **Byte Operations:**
  <u>Example:</u> mov.b R5,&P1OUT
  Only performs the operation on the last two bytes.

- **Byte operation on a word:**
  If you did mov.b on a word, then what you will end up with is simply the last two bytes of the word and the rest will become 0.

- **Word operation on a byte:**
  If you did mov.w on a byte, you will simply only move the last two because that is all that is actually there?

				Status Bits			
				V	N	Z	C
* ADC(.B)	dst	dst + C → dst		x	x	x	x
ADD(.B)	src,dst	src + dst → dst		x	x	x	x
ADDC(.B)	src,dst	src + dst + C → dst		x	x	x	x
AND(.B)	src,dst	src .and. dst → dst		0	x	x	x
BIC(.B)	src,dst	.not.src .and. dst → dst		-	-	-	-
BIS(.B)	src,dst	src .or. dst → dst		-	-	-	-
BIT(.B)	src,dst	src .and. dst		0	x	x	x
* BR	dst	Branch to		-	-	-	-
CALL	dst	PC+2 → stack, dst → PC		-	-	-	-
* CLR(.B)	dst	Clear destination		-	-	-	-
* CLRC		Clear carry bit		-	-	-	0
* CLRN		Clear negative bit		-	0	-	-
* CLRZ		Clear zero bit		-	-	0	-
CMP(.B)	src,dst	dst - src		x	x	x	x
* DADC(.B)	dst	dst + C → dst (decimal)		x	x	x	x
DADD(.B)	src,dst	src + dst + C → dst (decimal)		x	x	x	x
* DEC(.B)	dst	dst - 1 → dst		x	x	x	x
* DECD(.B)	dst	dst - 2 → dst		x	x	x	x
* DINT		Disable interrupt		-	-	-	-
* EINT		Enable interrupt		-	-	-	-
* INC(.B)	dst	Increment destination, dst +1 → dst		x	x	x	x
* INCD(.B)	dst	Double-Increment destination, dst+2→dst		x	x	x	x
* INV(.B)	dst	Invert destination		x	x	x	x
JC/JHS	Label	Jump to Label if Carry-bit is set		-	-	-	-
JEQ/JZ	Label	Jump to Label if Zero-bit is set		-	-	-	-
JGE	Label	Jump to Label if (N .XOR. V) = 0		-	-	-	-
JL	Label	Jump to Label if (N .XOR. V) = 1		-	-	-	-
JMP	Label	Jump to Label unconditionally		-	-	-	-
JN	Label	Jump to Label if Negative-bit is set		-	-	-	-

				V	N	Z	C
				Status Bits			
	JNC/JLO	Label	Jump to Label if Carry-bit is reset	-	-	-	-
	JNE/JNZ	Label	Jump to Label if Zero-bit is reset	-	-	-	-
	MOV(.B)	src,dst	src → dst	-	-	-	-
*	NOP		No operation	-	-	-	-
*	POP(.B)	dst	Item from stack, SP+2 → SP	-	-	-	-
	PUSH(.B)	src	SP - 2 → SP, src → @SP	-	-	-	-
	RETI		Return from interrupt TOS → SR, SP + 2 → SP TOS → PC, SP + 2 → SZP	x	x	x	x
*	RET		Return from subroutine TOS → PC, SP + 2 → SP	-	-	-	-
*	RLA(.B)	dst	Rotate left arithmetically	x	x	x	x
*	RLC(.B)	dst	Rotate left through carry	x	x	x	x
	RRA(.B)	dst	MSB → MSB ....LSB → C	0	x	x	x
	RRC(.B)	dst	C → MSB ....LSB → C	x	x	x	x
*	SBC(.B)	dst	Subtract carry from destination	x	x	x	x
*	SETC		Set carry bit	-	-	-	1
*	SETN		Set negative bit	-	1	-	-
*	SETZ		Set zero bit	-	-	1	-
	SUB(.B)	src,dst	dst + .not.src + 1 → dst	x	x	x	x
	SUBC(.B)	src,dst	dst + .not.src + C → dst	x	x	x	x
	SWPB	dst	swap bytes	-	-	-	-
	SXT	dst	Bit7 → Bit8 .... Bit15	0	x	x	x
*	TST(.B)	dst	Test destination	x	x	x	x
	XOR(.B)	src,dst	src .xor. dst → dst	x	x	x	x

Legend:

Legend:	0	The Status Bit is cleared	1	The Status Bit is set
	x	The Status Bit is affected	-	The Status Bit is not affected
	*	Emulated Instructions		

6) Different addressing modes for source and destination and how they work (Refer to Manuals in Canvas or CPE 323 slides)

Addressing modes help set the following table below, or the instruction in memory. Ad is for destination mode, As is source mode, and B/w is byte or word operation. Op-Code’s for different commands are shown below. S-Reg is source register, D-Reg is destination register.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Op-code				S-Reg				Ad	B/W	As		D-Reg			

As	Ad	Addressing Mode	Syntax	Description
00	0	Register Mode	Rn	Register contents are operand
01	1	Indexed Mode	X(Rn)	(Rn + X) points to the operand. X is stored in the next word
01	1	Symbolic Mode	ADDR	(PC + X) points to the operand. X is stored in the next word. Indexed Mode X(PC) is used
01	1	Absolute Mode	&ADDR	The word following the instruction contains the absolute address.
10	-	Indirect Register Mode	@Rn	Rn is used as a pointer to the operand
11	-	Indirect Autoincrement	@Rn+	Rn is used as a pointer to the operand. Rn is incremented afterwards
11	-	Immediate Mode	#N	The word following the instruction contains the immediate constant N. Indirect Autoincrement Mode @PC+ is used

Table 1a: The complete MSP430 instruction set of 27 core instructions

core instruction mnemonics	core instruction binary															
Single-operand arithmetic	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	0	0	0	1	0	0	opcode			B/W	As		source			
RRC Rotate right through carry	0	0	0	1	0	0	0	0	0	B/W	As		source			
SWPB Swap bytes	0	0	0	1	0	0	0	0	1	0	As		source			
RRA Rotate right arithmetic	0	0	0	1	0	0	0	1	0	B/W	As		source			
SXT Sign extend byte to word	0	0	0	1	0	0	0	1	1	0	As		source			
PUSH Push value onto stack	0	0	0	1	0	0	1	0	0	B/W	As		source			
CALL Subroutine call; push PC and move source to PC	0	0	0	1	0	0	1	0	1	0	As		source			
RETI Return from interrupt; pop SR then pop PC	0	0	0	1	0	0	1	1	0	0	0	0	0	0	0	0
Conditional jump; PC = PC + 2×offset	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	0	0	1	condition			10-bit signed offset									
JNE/JNZ Jump if not equal/zero	0	0	1	0	0	0	10-bit signed offset									
JEQ/JZ Jump if equal/zero	0	0	1	0	0	1	10-bit signed offset									
JNC/JLO Jump if no carry/lower	0	0	1	0	1	0	10-bit signed offset									
JC/JHS Jump if carry/higher or same	0	0	1	0	1	1	10-bit signed offset									
JN Jump if negative	0	0	1	1	0	0	10-bit signed offset									
JGE Jump if greater or equal (N == V)	0	0	1	1	0	1	10-bit signed offset									
JL Jump if less (N != V)	0	0	1	1	1	0	10-bit signed offset									
JMP Jump (unconditionally)	0	0	1	1	1	1	10-bit signed offset									
Two-operand arithmetic	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	opcode				source				Ad	B/W	As		destination			
MOV Move source to destination	0	1	0	0	source				Ad	B/W	As		destination			
ADD Add source to destination	0	1	0	1	source				Ad	B/W	As		destination			
ADDC Add w/carry: dst += (src+C)	0	1	1	0	source				Ad	B/W	As		destination			
SUBC Subtract w/ carry: dst -= (src+C)	0	1	1	1	source				Ad	B/W	As		destination			
SUB Subtract; dst -= src	1	0	0	0	source				Ad	B/W	As		destination			
CMP Compare; (dst-src); discard result	1	0	0	1	source				Ad	B/W	As		destination			
DADD Decimal (BCD) addition: dst += src	1	0	1	0	source				Ad	B/W	As		destination			
BIT Test bits; (dst & src); discard result	1	0	1	1	source				Ad	B/W	As		destination			
BIC Bit clear; dest &= ~src	1	1	0	0	source				Ad	B/W	As		destination			
BIS "Bit set" - logical OR; dst |= src	1	1	0	1	source				Ad	B/W	As		destination			
XOR Bitwise XOR; dst ^= src	1	1	1	0	source				Ad	B/W	As		destination			
AND Bitwise AND; dst &= src	1	1	1	1	source				Ad	B/W	As		destination			

Dec	Hex	Bin
0	0	00000000
1	1	00000001
2	2	00000010
3	3	00000011
4	4	00000100
5	5	00000101
6	6	00000110
7	7	00000111
8	8	00001000
9	9	00001001
10	a	00001010
11	b	00001011
12	c	00001100
13	d	00001101
14	e	00001110
15	f	00001111

Hex	Decimal	Octal	Binary
0	0	0	0
1	1	1	1
2	2	2	10
3	3	3	11
4	4	4	100
5	5	5	101
6	6	6	110
7	7	7	111
8	8	10	1000
9	9	11	1001
A	10	12	1010
B	11	13	1011
C	12	14	1100
D	13	15	1101
E	14	16	1110
F	15	17	1111
10	16	20	10000
20	32	40	100000
40	64	100	1000000
80	128	200	10000000
100	256	400	100000000
200	512	1000	1000000000
400	1024	2000	10000000000

7) Binary operations with BIT0, BIT1, etc.

There are 6 Bitwise operators namely:

Operators	Meaning of operators
A & B	Bitwise AND
A | B	Bitwise OR
A ^ B	Bitwise XOR
~A	Bitwise Complement
A << B	Shift Left
A >> B	Shift Right

And some more created by there combinations, some of them are listed here:

Operators	Meaning of operators
~ (A & B)	Bitwise NAND
~ (A | B)	Bitwise NOR
~ (A ^ B)	Bitwise XNOR
A >>> B	Bitwise unsigned right shift
A &= B	Bitwise AND assignment
A |= B	Bitwise OR assignment
A ^= B	Bitwise XOR assignment
A <<= B	Bitwise left shift and assignment
A >>= B	Bitwise right shift and assignment
A >>>= B	Bitwise unsigned right shift and assignment

A mask defines which bits you want to keep, and which bits you want to clear.

Masking is the act of applying a mask to a value. This is accomplished by doing:

- **Bitwise ANDing in order to extract a subset of the bits in the value**
- **Bitwise ORing in order to set a subset of the bits in the value**
- **Bitwise XORing in order to toggle a subset of the bits in the value**

Below is an example of extracting a subset of the bits in the value:

```
Mask:   00001111b
Value:  01010101b
```

Applying the mask to the value means that we want to clear the first (higher) 4 bits, and keep the last (lower) 4 bits. Thus we have extracted the lower 4 bits. The result is:

```
Mask:   00001111b
Value:  01010101b
Result: 00000101b
```

Masking is implemented using AND, so in C we get:

```
uint8_t stuff(...) {
  uint8_t mask = 0x0f;   // 00001111b
  uint8_t value = 0x55;  // 01010101b
  return mask & value;
}
```

Here is a fairly common use-case: Extracting individual bytes from a larger word. We define the high-order bits in the word as the first byte. We use two operators for this, `&`, and >> (shift right). This is how we can extract the four bytes from a 32-bit integer:

```
void more_stuff(uint32_t value) {             // Example value: 0x01020304
    uint32_t byte1 = (value >> 24);           // 0x01020304 >> 24 is 0x01 so
                                              // no masking is necessary
    uint32_t byte2 = (value >> 16) & 0xff;    // 0x01020304 >> 16 is 0x0102 so
                                              // we must mask to get 0x02
    uint32_t byte3 = (value >> 8)  & 0xff;    // 0x01020304 >> 8 is 0x010203 so
                                              // we must mask to get 0x03
    uint32_t byte4 = value & 0xff;            // here we only mask, no shifting
                                              // is necessary
    ...
}
```

Notice that you could switch the order of the operators above, you could first do the mask, then the shift. The results are the same, but now you would have to use a different mask:

```
uint32_t byte3 = (value & 0xff00) >> 8;
```

Masking means to keep/change/remove a desired part of information. Lets see an image-masking operation; like- this masking operation is removing any thing that is not skin-

We are doing *AND* operation in this example. There are also other masking operators- *OR*, *XOR*.

Bit-Masking means imposing mask over bits. Here is a bit-masking with *AND*-

```
     1 1 1 0 1 1 0 1   [input]
(&)  0 0 1 1 1 1 0 0    [mask]
------------------------------
     0 0 1 0 1 1 0 0  [output]
```

So, only the middle 4 bits (as these bits are `1` in this mask) remain.

Lets see this with *XOR*-

```
     1 1 1 0 1 1 0 1   [input]
(^)  0 0 1 1 1 1 0 0    [mask]
------------------------------
     1 1 0 1 0 0 0 1  [output]
```

Now, the middle 4 bits are flipped (`1` became `0`, `0` became `1`).

So, using bit-mask we can access individual bits [examples]. Sometimes, this technique may also be used for improving performance. Take this for example-

```
bool isOdd(int i) {
    return i%2;
}
```

This function tells if an integer is odd/even. We can achieve the same result with more efficiency using bit-mask-

```
bool isOdd(int i) {
    return i&1;
}
```

Short Explanation: If the least significant bit of a binary number is `1` then it is odd; for `0` it will be even. So, by doing *AND* with `1` we are removing all other bits except for the least significant bit i.e.:

```
    55  ->  0 0 1 1 0 1 1 1   [input]
(&)  1  ->  0 0 0 0 0 0 0 1    [mask]
---------------------------------------
     1  <-  0 0 0 0 0 0 0 1  [output]
```