CPE 325 Quiz 3 Study Guide

1) How do stack and stack pointer (SP) work and how is it affected by pushing and popping data? How to retrieve data from the stack by using the stack pointer?

○ The stack is a last-in-first-out structure. When something is moved onto the stack using a PUSH.W instruction, the SP decrements by 2 (lowers / points to the next lowest word in memory) and then stores the value there, so the stack grows toward lower addresses. In the MSP430 that we've been using for labs, the stack starts at address 0x4400. When the first value is pushed, the SP points at address 0x43FE and the value is stored there. Using POP.W takes the value from the address that the SP is currently pointing to, stores it in the destination specified by the pop instruction, and increments the SP, so the stack shrinks toward higher addresses.

```
main:
        push    #swarr          ; Push the software array address onto the stack.
        push    #hwarr          ; Push the hardware array onto the stack.
        push    result          ; Push the result onto the stack.
```

```
mov     6(SP), R5       ; Move the Software array address into R5
mov     4(SP), R6       ; Move the Hardware array address into R6
mov     2(SP), R7       ; Move the result into R7
```

2) How to change some or all of the bits in registers in assembly and C (bis, bic, mov, |=, &= ~, etc)?

```
SetupP2:
            bis.b   #001h, &P1DIR           ; Set P1.0 to output
                                            ; direction (0000_0001)
            bic.b   #001h, &P1OUT           ; Set P1OUT to 0x0000_0001 (ensure
                                            ; LED1 is off)

            bic.b   #002h, &P2DIR           ; SET P2.1 as input for SW1
            bis.b   #002h, &P2REN           ; Enable Pull-Up resister at P2.1
            bis.b   #002h, &P2OUT           ; required for proper IO set up

ChkSW1:     bic.b   #001h, &P1OUT
            bit.b   #002h, &P2IN             ; Check if SW1 is pressed
                                            ; (0000_0010 on P1IN)
            jnz     ChkSW1                  ; If not zero, SW1 is not pressed
                                            ; loop and check again
Debounce:
            mov.w   #2000, R15              ; Set to (2000 * 10 cc = 20,000 cc)
SWD20ms:    dec.w   R15                     ; Decrement R15
            nop
            nop
            nop
            nop
            nop
            nop
            nop
            jnz     SWD20ms                 ; Delay over?
            bit.b   #00000010b, &P2IN       ; Verify SW1 is still pressed
            jnz     ChkSW1                  ; If not, wait for SW1 press

LEDon:      bis.b   #001h, &P1OUT           ; Turn on LED1
SW1wait:    bit.b   #002h, &P2IN            ; Test SW1
            jz      SW1wait                 ; Wait until SW1 is released
            bic.b   #001h, &P1OUT            ; Turn off LED1
            jmp     ChkSW1                  ; Loop to beginning
            nop
```

```
P1SEL |= BIT0;
P2DIR |= BIT2;                          // SMCLK set out to pins
P2SEL |= BIT2;
P7DIR |= BIT7;                          // MCLK set out to pins
P7SEL |= BIT7;
```

```
P2DIR &= ~BIT1;                 // set P2.1 as input (SW1)
P2REN |= BIT1;                  // enable pull-up resistor
P2OUT |= BIT1;
P2IE |= BIT1;                   // enable interrupt at P2.1
P2IES |= BIT1;                  // enable hi->lo edge for interrupt
P2IFG &= ~BIT1;                 // clear any errornous interrupt flag
```

3) How does the hardware multiplier work? What are the registers associated with it and what is the purpose of each one?

The hardware multiplier has two 16-bit operand registers, OP1 and OP2, and three result registers, RESLO, RESHI, and SUMEXT. RESLO stores the low word of the result, RESHI stores the high word of the result, and SUMEXT stores information about the result. The result is ready in three MCLK cycles and can be read with the next instruction after writing to OP2, except when using an indirect addressing mode to access the result. When using indirect addressing for the result, a NOP is required before the result is ready.

Appendix 4 – HW_Mult.asm

```
;-------------------------------------------------------------------------------
; File       :  HW_mult.asm
; Function   :  To perform a hardware multiplication on predefined values
; Description:  This file will take in values and use hardware multiplication to
;                                find the powers from 1-5
; Input      :  R4 and R7 from main and calc_power.asm
; Output     :  See memory browser
; Author     :  N. Anderson  npa0002@uah.edu
; Date       :  09/30/2020
;-------------------------------------------------------------------------------
            .cdecls C,LIST,"msp430.h"      ; Include device header file
            .def HW_Mult
            .text
HW_Mult:

        mov     R4, &MPY        ; move R4 into the unsinged 16 bit multiplication register
        mov     R7, &OP2        ; move R4 into the general purpose operator. multiply by...
        nop
        nop
        nop
        mov     RESLO, R7       ; move the result the R7.
        mov     R7, 0(R6)       ; move result  into R6
        ret
```

```
; 16x16 Unsigned Multiply
   MOV   #01234h,&MPY ; Load first operand
   MOV   #05678h,&OP2 ; Load second operand
;  ...                ; Process results

; 8x8 Unsigned Multiply. Absolute addressing.
   MOV.B #012h,&0130h ; Load first operand
   MOV.B #034h,&0138h ; Load 2nd operand
;  ...                ; Process results

; 16x16 Signed Multiply
   MOV   #01234h,&MPYS ; Load first operand
   MOV   #05678h,&OP2 ; Load 2nd operand
;  ...                ; Process results

; 8x8 Signed Multiply. Absolute addressing.
   MOV.B #012h,&0132h ; Load first operand
   SXT   &MPYS        ; Sign extend first operand
   MOV.B #034h,&0138h ; Load 2nd operand
   SXT   &OP2         ; Sign extend 2nd operand
                      ; (triggers 2nd multiplication)
;  ...                ; Process results


; 16x16 Unsigned Multiply Accumulate
   MOV   #01234h,&MAC ; Load first operand
   MOV   #05678h,&OP2 ; Load 2nd operand
;  ...                ; Process results

; 8x8 Unsigned Multiply Accumulate. Absolute addressing
   MOV.B #012h,&0134h ; Load first operand
   MOV.B #034h,&0138h ; Load 2nd operand
;  ...                ; Process results

; 16x16 Signed Multiply Accumulate
   MOV   #01234h,&MACS ; Load first operand
   MOV   #05678h,&OP2 ; Load 2nd operand
;  ...                ; Process results

; 8x8 Signed Multiply Accumulate. Absolute addressing
   MOV.B #012h,&0136h ; Load first operand
   SXT   &MACS        ; Sign extend first operand
   MOV.B #034h,R5     ; Temp. location for 2nd operand
   SXT   R5           ; Sign extend 2nd operand
   MOV   R5,&OP2      ; Load 2nd operand
;  ...                ; Process results


; Access multiplier results with indirect addressing
   MOV   #RESLO,R5    ; RESLO address in R5 for indirect
   MOV   &OPER1,&MPY  ; Load 1st operand
   MOV   &OPER2,&OP2  ; Load 2nd operand
   NOP                ; Need one cycle
   MOV   @R5+,&xxx    ; Move RESLO
   MOV   @R5,&xxx     ; Move RESHI
```

4) What are the different data types in assembly and how much memory do they need? (.byte, .int, .string, .usect, .space, etc)

Table 5-1. Directives that Control Section Use

Mnemonic and Syntax	Description	See
.bss *symbol, size in bytes*[, *alignment*]	Reserves *size* bytes in the .bss (uninitialized data) section	.bss topic
.data	Assembles into the .data (initialized data) section	.data topic
.intvec	Creates an interrupt vector entry in a named section that points to an interrupt routine name.	.intvec topic
.sect "*section name***"**	Assembles into a named (initialized) section	.sect topic
.text	Assembles into the .text (executable code) section	.text topic
symbol **.usect "***section name***",** *size in bytes* [, *alignment*]	Reserves *size* bytes in a named (uninitialized) section	.usect topic

Table 5-4. Directives that Initialize Values (Data and Memory)

Mnemonic and Syntax	Description	See
.bits $value_1$[, ... , $value_n$]	Initializes one or more successive bits in the current section	.bits topic
.byte $value_1$[, ... , $value_n$]	Initializes one or more successive bytes in the current section	.byte topic
.char $value_1$[, ... , $value_n$]	Initializes one or more successive bytes in the current section	.char topic
.cstring {$expr_1$|"$string_1$"}[,... , {$expr_n$|"$string_n$"}]	Initializes one or more text strings	.string topic

Table 5-4. Directives that Initialize Values (Data and Memory) (continued)

Mnemonic and Syntax	Description	See
.double *value*$_1$[, ... , *value*$_n$]	Initializes one or more 64-bit, IEEE double-precision, floating-point constants	.double topic
.field *value*[, *size*]	Initializes a field of *size* bits (1-32) with *value*	.field topic
.float *value*$_1$[, ... , *value*$_n$]	Initializes one or more 32-bit, IEEE single-precision, floating-point constants	.float topic
.half *value*$_1$[, ... , *value*$_n$]	Initializes one or more 16-bit integers (halfword)	.half topic
.int *value*$_1$[, ... , *value*$_n$]	Initializes one or more 16-bit integers	.int topic
.long *value*$_1$[, ... , *value*$_n$]	Initializes one or more 32-bit integers	.long topic
.short *value*$_1$[, ... , *value*$_n$]	Initializes one or more 16-bit integers (halfword)	.short topic
.string {*expr*$_1$|"*string*$_1$"}[,... , {*expr*$_n$|"*string*$_n$"}]	Initializes one or more text strings	.string topic
.ubyte *value*$_1$[, ... , *value*$_n$]	Initializes one or more successive unsigned bytes in the current section	.ubyte topic
.uchar *value*$_1$[, ... , *value*$_n$]	Initializes one or more successive unsigned bytes in the current section	.uchar topic
.uhalf *value*$_1$[, ... , *value*$_n$]	Initializes one or more unsigned 16-bit integers (halfword)	.uhalf topic
.uint *value*$_1$[, ... , *value*$_n$]	Initializes one or more unsigned 32-bit integers	.uint topic
.ulong *value*$_1$[, ... , *value*$_n$]	Initializes one or more unsigned 32-bit integers	.long topic
.ushort *value*$_1$[, ... , *value*$_n$]	Initializes one or more unsigned 16-bit integers (halfword)	.short topic
.uword *value*$_1$[, ... , *value*$_n$]	Initializes one or more unsigned 16-bit integers	.uword topic
.word *value*$_1$[, ... , *value*$_n$]	Initializes one or more 16-bit integers	.word topic

Table 5-5. Directives that Perform Alignment and Reserve Space

Mnemonic and Syntax	Description	See
.align [*size in bytes*]	Aligns the SPC on a boundary specified by *size in bytes*, which must be a power of 2; defaults to word (2-byte) boundary	.align topic
.bes *size*	Reserves *size* bytes in the current section; a label points to the end of the reserved space	.bes topic
.space *size*	Reserves *size* bytes in the current section; a label points to the beginning of the reserved space	.space topic

5) Interrupts: What are the registers used for setting up the interrupts and what is the purpose of each one (PxIE, PxIES, PxIFG, etc.)? How many interrupt service routines can a port have?

```
P2IE  |= BIT1;                  // Enable interrupt at P2.1 for Switch 2
P2IES |= BIT1;                  // enable hi->lo edge for interrupt
P2IFG &= ~BIT1;                 // clear any errornous interrupt flag
```

```
bis.w    #GIE, SR              ; Enable Global Interrupts
bis.b    #002h, &P1IE          ; Enable Port 1 interrupt from bit 1
bis.b    #002h, &P1IES         ; Set interrupt to call from hi to low
bic.b    #002h, &P1IFG         ; Clear interrupt flag
```

6) What are the different clocks in MSP430? How does changing the clock frequency affect the number of clock cycles and execution time?

PARAMETER		TEST CONDITIONS	MIN	TYP	MAX	UNIT
$f_{DCO(0,0)}$	DCO frequency (0, 0)[1]	DCORSELx = 0, DCOx = 0, MODx = 0	0.07		0.20	MHz
$f_{DCO(0,31)}$	DCO frequency (0, 31)[1]	DCORSELx = 0, DCOx = 31, MODx = 0	0.70		1.70	MHz
$f_{DCO(1,0)}$	DCO frequency (1, 0)[1]	DCORSELx = 1, DCOx = 0, MODx = 0	0.15		0.36	MHz
$f_{DCO(1,31)}$	DCO frequency (1, 31)[1]	DCORSELx = 1, DCOx = 31, MODx = 0	1.47		3.45	MHz
$f_{DCO(2,0)}$	DCO frequency (2, 0)[1]	DCORSELx = 2, DCOx = 0, MODx = 0	0.32		0.75	MHz
$f_{DCO(2,31)}$	DCO frequency (2, 31)[1]	DCORSELx = 2, DCOx = 31, MODx = 0	3.17		7.38	MHz
$f_{DCO(3,0)}$	DCO frequency (3, 0)[1]	DCORSELx = 3, DCOx = 0, MODx = 0	0.64		1.51	MHz
$f_{DCO(3,31)}$	DCO frequency (3, 31)[1]	DCORSELx = 3, DCOx = 31, MODx = 0	6.07		14.0	MHz
$f_{DCO(4,0)}$	DCO frequency (4, 0)[1]	DCORSELx = 4, DCOx = 0, MODx = 0	1.3		3.2	MHz
$f_{DCO(4,31)}$	DCO frequency (4, 31)[1]	DCORSELx = 4, DCOx = 31, MODx = 0	12.3		28.2	MHz
$f_{DCO(5,0)}$	DCO frequency (5, 0)[1]	DCORSELx = 5, DCOx = 0, MODx = 0	2.5		6.0	MHz
$f_{DCO(5,31)}$	DCO frequency (5, 31)[1]	DCORSELx = 5, DCOx = 31, MODx = 0	23.7		54.1	MHz
$f_{DCO(6,0)}$	DCO frequency (6, 0)[1]	DCORSELx = 6, DCOx = 0, MODx = 0	4.6		10.7	MHz
$f_{DCO(6,31)}$	DCO frequency (6, 31)[1]	DCORSELx = 6, DCOx = 31, MODx = 0	39.0		88.0	MHz
$f_{DCO(7,0)}$	DCO frequency (7, 0)[1]	DCORSELx = 7, DCOx = 0, MODx = 0	8.5		19.6	MHz
$f_{DCO(7,31)}$	DCO frequency (7, 31)[1]	DCORSELx = 7, DCOx = 31, MODx = 0	60		135	MHz
$S_{DCORSEL}$	Frequency step between range DCORSEL and DCORSEL + 1	$S_{RSEL} = f_{DCO(DCORSEL+1,DCO)}/f_{DCO(DCORSEL,DCO)}$	1.2		2.3	ratio
S_{DCO}	Frequency step between tap DCO and DCO + 1	$S_{DCO} = f_{DCO(DCORSEL,DCO+1)}/f_{DCO(DCORSEL,DCO)}$	1.02		1.12	ratio
	Duty cycle	Measured at SMCLK	40%	50%	60%	
df_{DCO}/dT	DCO frequency temperature drift[2]	f_{DCO} = 1 MHz		0.1		%/°C
df_{DCO}/dV_{CC}	DCO frequency voltage drift[3]	f_{DCO} = 1 MHz		1.9		%/V

(1) When selecting the proper DCO frequency range (DCORSELx), the target DCO frequency, f_{DCO}, should be set to reside within the range of $f_{DCO(n,0),MAX} \le f_{DCO} \le f_{DCO(n,31),MIN}$, where $f_{DCO(n,0),MAX}$ represents the maximum frequency specified for the DCO frequency, range n, tap 0 (DCOx = 0) and $f_{DCO(n,31),MIN}$ represents the minimum frequency specified for the DCO frequency, range n, tap 31 (DCOx = 31). This ensures that the target DCO frequency resides within the range selected. It should also be noted that if the actual f_{DCO} frequency for the selected range causes the FLL or the application to select tap 0 or 31, the DCO fault flag is set to report that the selected range is at its minimum or maximum tap setting.

(2) Calculated using the box method: (MAX(–40°C to 85°C) – MIN(–40°C to 85°C)) / MIN(–40°C to 85°C) / (85°C – (–40°C))

(3) Calculated using the box method: (MAX(1.8 V to 3.6 V) – MIN(1.8 V to 3.6 V)) / MIN(1.8 V to 3.6 V) / (3.6 V – 1.8 V)

MCLK - Main clock, CPU uses
SMCLK - submain clock
ACLK - Auxiliary clock

7) How do you turn on or off, or toggle LEDs? How to set or clear flags?

Command	C	Assembly
Toggle	P4OUT ^= BIT7	xor.b #0x80, &P4OUT
On	P4OUT |= BIT7	bis.b #001h, &P1OUT
Off	P1OUT &= ~BIT0	Bic.b #001, &P1OUT

8) How is an interrupt different from a subroutine? What happens to the stack when each one is called? What happens when each one has finished executing?

Subroutines are executed using the CALL instruction which moves the current PC value onto the stack (return address) and then changes the PC to represent the subroutine. To exit, you use RET which sends program control back to where it came from. Similar to a POP PC instruction, it pops whatever is at the bottom of the stack and puts that into the PC. You need to make sure that you have a PUSH and POP instruction for all of the subroutines so that you are always at the correct address.

On the other hand, **Interrupts** are similar to subroutines in that they have control over the program, but they are activated differently. Interrupts can be activated from anywhere in the program and allows the PC to execute the interrupt while it is there.

> *“The ISR handles an interrupt by checking the status of the interrupt, determining why the interrupt occurred and what action needs to be taken. Although the ISR usually will not handle the interrupt itself, it is the "first on the scene," so to speak, and prepares the system for interrupt handling.”*

```
.sect   ".int47"        ; Port 1 vector
.short  SW2_ISR
.sect   ".int42"        ; Port 2 vector
.short  SW1_ISR
```

```
C                         Assembly

_EINT();          or      bis.w    #GIE, SR
                  // Enable global interrupts
PxIE |= BITy;     or      bis.b    #0x__, &PxIE
                  // Enable interrupts at Px.y
PxIES |= BITy;    or      bis.b    #0x__, &PxIES
                  // Switch button press activates interrupt
PxIFG &= ~BITy;   or      bic.b    #0x__, &PxIFG
                  // Clear interrupt flag for Px.y
```

9) What is the purpose of PxSEL? What happens when the bits in that register are set to 0 or 1?

PxSEL selects the functionality of the GPIO (General Purpose) pin as its multiplexed with other functionalities. Decides whether a pin is controlled by PxIN, PxOUT, and PxDIR for generic I/O.

Table 8-1. PxSEL and PxSEL2

PxSEL2	PxSEL	Pin Function
0	0	I/O function is selected.
0	1	Primary peripheral module function is selected.
1	0	Reserved. See device-specific data sheet.
1	1	Secondary peripheral module function is selected.

10) How are signed and unsigned numbers (or positive and negative numbers) represented in binary?