Lecture 24

Laundry Day aka Interrupts for a Real Life MCU Application

Joke of the Day

Why did the programmer die in the shower?

He read the shampoo bottle instructions:

Lather. Rinse. Repeat.

Recap: Multiplying Signed Numbers

Recall, given a number x

- If the number is positive, represent it with the binary numeral for **x**
- If the number is negative, represent it with the binary numeral for $2^{16} - |x|$

Let's multiply a positive and negative number: **x>0** and **y<0**

Binary representations will be **x** and $2^{16} - |y|$

$2^{16} x - x |y|$

And two negative numbers: **x<0** and **y<0**

Binary representations will be $2^{16} - |x|$ and $2^n - |y|$

These are the result in the n-bit register

$2^{32} - 2^{16} (|x| + |y|) + |x| |y|$

⇒ **Multiplication works the same way for signed & unsigned numbers as long as |xy| does not overflow the 16-bit signed number range**

Signed/Unsigned `x_times_y`

```
x_times_y:

; Save afftected core registers on stack - You can add this part last once you
; know which registered are modified
            push.w  R6
            push.w  R10
            push.w  R11

            clr.w   R12             ; R12 will accumulate R5*R6
            clr.w   R10             ; R10 will index bits j = 0, 1, ..., 7
            mov.w   #BIT0, R11      ; R11 has the bitmask to use with tst.w

check_next_bit:
            bit.w   R11, R5         ; Is the jth bit 1?
            jnc     prep_next_bit   ; If not prepare for checking next bit

            add.w   R6, R12         ; Bit j is 1, add

prep_next_bit:
            rla.w   R11             ; Prepare next bitmask
            rla.w   R6              ; Prepare shifted version of R6
            inc.w   R10             ; increase bit index
            cmp.w   #16, R10        ; Are we done with all bits?
            jlo     check_next_bit

; Restore saved core registers from stack
; Watch the order and make sure not to leave anything behind
            pop.w   R11
            pop.w   R10
            pop.w   R6

            ret
```


Multiply all 16 bits, not just 8, but make sure that |xy| does not overflow signed integer range

Quiz 6

Part 1: Coding Task (50 pts)

Your program should start with both LEDs off (i.e., not emitting light), and wait for a push button to be pressed. When either push button is pressed, an interrupt should be triggered on the raising edge. A single interrupt routine serves the interrupts and accomplishes following task:

- Pressing S1 toggles the **green LED**
- Pressing S2 toggles the **red LED**

Toggling an LED means the following: if the LED is off, it is turned on; alternatively, if the LED is on, it is turned off.

Your program should let you press the buttons as many times as you want, and in any order, and exhibit correct behavior.

Solution to Quiz 6 – Main Loop

```
;-------------------------------------------------------------------------------
; Main loop here
;-------------------------------------------------------------------------------
            ; Configure red LED for output, start with unlit LED
            ; Red LED is connected to P1.0
            bic.b   #BIT0, &P1OUT               ; Red LED off
            bis.b   #BIT0, &P1DIR               ; Direction to outpu

            ; Configure green LED for output, start with unlit LED
            ; Green LED is connected to P9.7
            bic.b   #BIT7, &P9OUT
            bis.b   #BIT7, &P9DIR

            ; Configure push buttons S1 and S2 for input
            ; S1 is connected to P1.1, S2 is connected to P1.2
            bis.b   #BIT1|BIT2, &P1REN          ; Resistor enabled
            bis.b   #BIT1|BIT2, &P1OUT          ; Pullup resistor
            bic.b   #BIT1|BIT2, &P1IES          ; Interrupt on raising-edge
            bis.b   #BIT1|BIT2, &P1IE           ; Enable port interrupts

            ; Disable power lock
            bic.w   #LOCKLPM5, &PM5CTL0

            ; Clear all IFGs in P1 in case they are set during config
            clr.b   &P1IFG

            nop
            eint                                ; Enable general interrupts
            nop

main:       jmp     main
```


No need to configure S1 and S2 separately

Good idea →

No `nop` necessary when there is more code to follow – in a subroutine or ISR

How to Write ISRs?

First thing to do is to check **the source of the interrupt**

```
P1_ISR:
            ; Check the source of the interrupt
Check_S1:   bit.b   #BIT1, &P1IFG
            jnc     Check_S2
            ; Serve P1IFG.1
Check_S2:   bit.b   #BIT2, &P1IFG
            jnc     return_from_P1_ISR
            ; Serve P1IFG.2

return_from_P1_ISR:
            reti
```


Ideally, all unused interrupts **should be disabled**

We did not pay much attention to this – but default settings are disable

Still a good idea to check the source even if there is only one interrupt expected from the source

How to Write ISRs?

The ISR needs to clear the interrupt flag!

BUT do not get carried away and wipe out the entire register

Clear only the flags you have served!!!


```
P1_ISR:
            ; Check the source of the interrupt
Check_S1:   bit.b   #BIT1, &P1IFG
            jnc     Check_S2

            xor.b   #BIT7, &P9OUT

Check_S2:   bit.b   #BIT2, &P1IFG
            jnc     return_from_P1_ISR

            xor.b   #BIT0, &P1OUT

return_from_P1_ISR:
            clr.b   &P1IFG
            reti
```

Might work for the given task BUT not good practice – think *nuking a mosquito*

Solution to Quiz 6 – ISR

```
;-------------------------------------------------------------------
; Interrupt Service Routines
;-------------------------------------------------------------------
P1_ISR:

check_S1:   ; Check source of interrupt: is it P1.1?
            bit.b   #BIT1, &P1IFG
            jnc     check_S2

service_S1:
            xor.b   #BIT7, &P9OUT
            bic.b   #BIT1, &P1IFG

check_S2:
            ; Check source of interrupt: is it P1.2?
            bit.b   #BIT2, &P1IFG
            jnc     return_from_P1_ISR

service_S2:
            xor.b   #BIT0, &P1OUT
            bic.b   #BIT2, &P1IFG

return_from_P1_ISR:

            reti                    ; return from interrupt
```


check

serve

clear

Solution to Quiz 6 – IVT

We add the **label of the ISR** to the Interrupt Vectors (at the end of *.asm)

For Port P1

```
;-----------------------------------------------------------
; Interrupt Vectors
;-----------------------------------------------------------
            .sect   ".int37"
            .short  P1_ISR

            .sect   ".reset"
            .short  RESET
```

Identifies address 0xFFDA

Label of ISR

One Word of Caution

The order of your code can make a big difference !!

```
;-------------------------------------------
; Main loop here
;-------------------------------------------
main:       jmp     main

;-------------------------------------------
; Subroutines
;-------------------------------------------
Sub_1:      ret

;-------------------------------------------
; Interrupt Service Routines
;-------------------------------------------
ISR_1:      reti

;-------------------------------------------
; Stack Pointer definition
;-------------------------------------------
            .global __STACK_END
            .sect   .stack

;-------------------------------------------
; Interrupt Vectors
;-------------------------------------------
            .sect   ".reset"
            .short  RESET
```

If you sandwich ISR between Stack Pointer definition and Interrupt Vectors

your code will crash

Your main.asm needs to end with these two blocks in this order

Laundry Day

What does the MCU of a washing machine do?

- Program selection: Take *user input* and set variables such as: target water temperature, target spin speed, cycle length etc.
- Measure water temperature (sensors), compare against *target* water temperature
- Turn on/off heating element based on outcome of above comparison

⇒ Control water temperature using a closed loop feedback

- Control spin speed
- Set timers to end one cycle segment and proceed to next segment: wash, rinse, spin
- Connect to WiFi ???

Configuring Target Water Temperature

User presses a single button to cycle through possible options:

Tap Cold → Cold → Warm → Hot → Extra Hot → Tap Cold → Cold …

Task: Write assembly code that takes user input through push button S1 and sets the target water temperature (variable `target_temp`)

State machine starts at Warm and cycles through states as shown above

Temperature values are

Tap Cold: no target value, no water temperature control loop

Cold : 30°C	Hot : 60°C
Warm : 40°C	Extra Hot : 95°C

Follow good programming practices **and good problem solving**

- define constants instead of hardcoding values
- write modular code: ISR calls subroutine `set_target_temp`