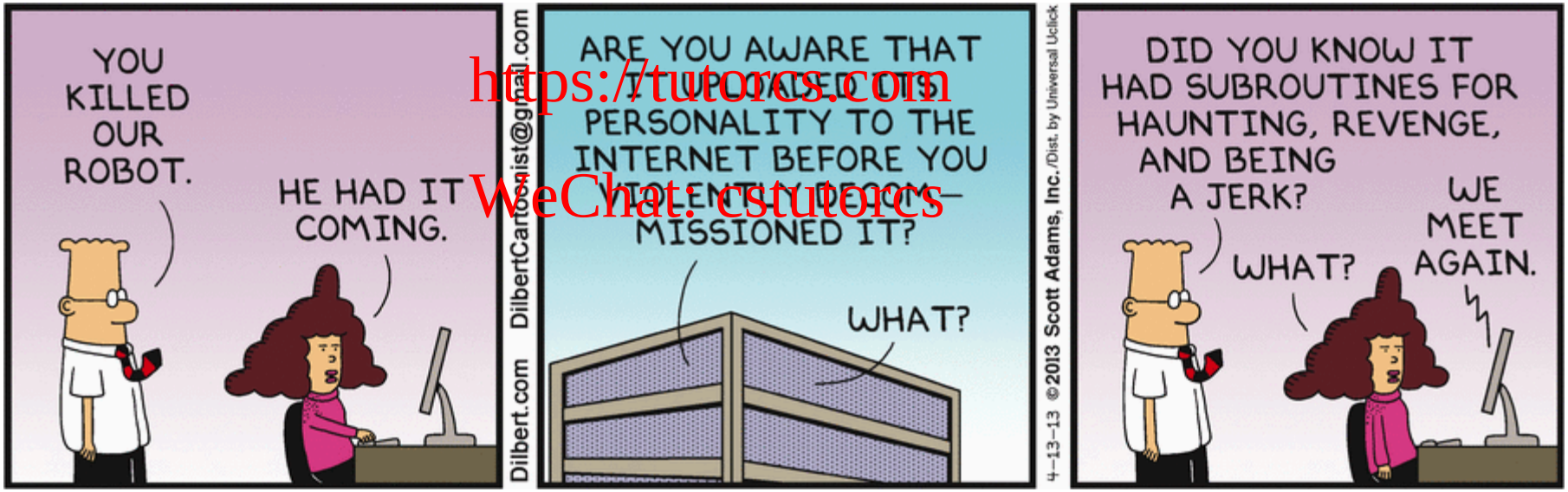
Lecture 13

Subroutines I

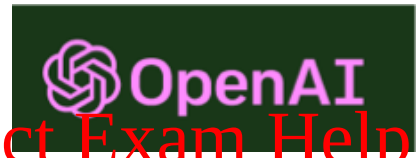


Announcements

Midterm #1 was due today – 4:10 pm

Will post solutions next week, grading will take time – 140+ students

BUT one submission already graded:



ChatGPT receives 0/100 – **zero**!

- The code does not compile
- Not MSP430 assembly: incorrect instructions and incorrect syntax
- Even after fixing those issues: incorrect logic

Upcoming assignments:

Posted a graded anonymous survey: Mid-Semester Class Feedback

Will post Quiz #4 tonight/tomorrow – a short subroutine

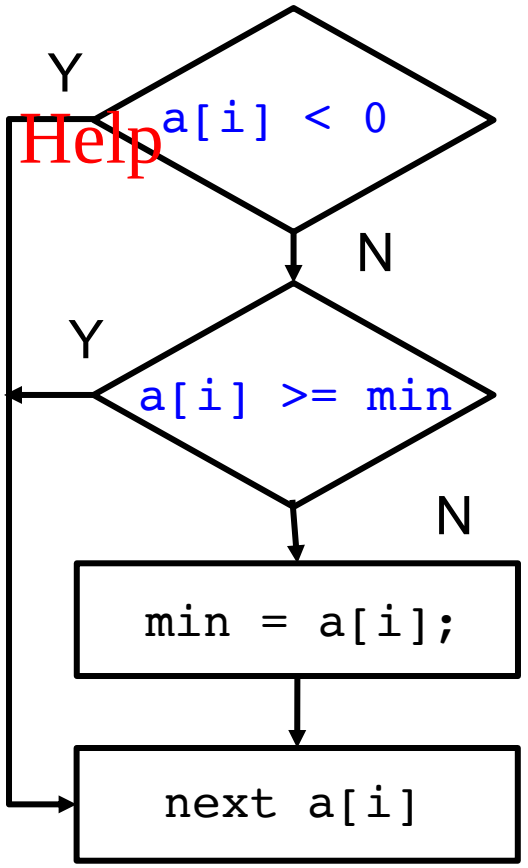
both due Wednesday March 1

Last Time: Compound Conditionals

Task: Given an array of ten signed integers, find the min. nonnegative value

Easy in a high level language once we have a loop that finds the minimum

```
min = infinity;
for (ii = 0; i < length; i++){
    if ( (a[i] >= 0) &&
         (a[i] < min_pos) )
        min = a[i];
}
```

One Solution

```
            .data
min_pos:    .space 2

            .text                           ; Assemble into program memory.
            .retain                         ; Override ELF conditional linking
            .retainrefs                     ; And retain any sections that have

array:      .word   -37, 101, -59, -47, 23, 11, 79, -131, -5, 163
;-------------------------------------------------------------------------------
RESET       mov.w   #__STACK_END,SP         ; Initialize stackpointer
StopWDT     mov.w   #WDTPW|WDTHOLD,&WDTCTL  ; Stop watchdog timer

;-------------------------------------------------------------------------------
; Main loop here
;-------------------------------------------------------------------------------
            mov.w   #0x7FFF, min_pos    ; min_pos = infinity/max. 16-bit signed #
            clr.w   R4

read_next:  tst.w   array(R4)
            jn      proceed             ; skip if negative

non_neg:    cmp.w   array(R4), min_pos  ; if min_pos - array(R4) > 0 replace
            jlo     proceed

            mov.w   array(R4), min_pos

proceed:    incd.w  R4
            cmp.w   #2*10, R4           ; check for end of array
            jlo     read_next           ; break if R4==20

main:       jmp     main
            nop
```

How to Solve a Problem?

Before jumping to the solution …

… take the time to study the problem and understand it well

Let's have a look at 16-bit signed numbers

0000	0	0
0001	1	1
...		
7FFE	32766	32766
7FFF	32767	32767
8000	-32768	32768
8001	-32767	32769
...		
FFFE	-2	65534
FFFF	-1	65535

Key Observation:

Every negative number is larger than every positive number if we do unsigned comparison

⇒ Minimum nonnegative value in array is the minimum value

⇒ No need to check for sign

Better Solution

Use unsigned compare, start with `min_pos = 0xFFFF`

```
            mov.w   #0, R4                  ; Set R4 as 2 to index second value of array
                                            ; Can start at 2nd value because minPos initializ
Repeat:
            cmp.w   array(R4), minPos       ; See if current value is less than value at inde
            jlo     if_NonNeg               ; Use unsigned compare because negative numbers
                                            ; will always be evaluated as higher
                                            ; Assume that there is at least one non Ne
            mov.w   array(R4), minPos       ; Set minPos to value in R4 to record current sma

if_NonNeg:
            incd.w  R4                      ; Increment twice to get to next word in array
            cmp.w   #20, R4                 ; Make sure we are still in array
            jl      Repeat                  ; Loop again if we are still in array


Inf_Loop:
            jmp     Inf_Loop
            nop
```

A Simple Subroutine

In Midterm #1 you had to divide by 16 twice in the code

Do we really need to write the same code twice? No!

We can write a simple **subroutine** to divide by 16


```
output = floor(input/16)
```


We can call this subroutine every time we need to divide by 16

- Allows us to reuse code
- Makes it easier to write, test, and maintain code
- Enables the use of libraries

Good code is _____ . **defect-free** **efficient** **modular**

A Simple Subroutine

Task: Write a simple subroutine **`div_by_16`** to divide a *given input* by 16

What registers are affected by subroutine – if any?

What is the input, output, functionality?

Contract


```
;-------------------------------------------------------------
; Subroutine: div_by_16
; Input:    16-bit signed number in R5 -- modified
; Output:   16-bit signed number in R5 -- R5 = floor(R5/16)
;-------------------------------------------------------------
div_by_16:  rra.w   R5          ; R5 <-- R5/2
            rra.w   R5          ; R5 <-- R5/2
            rra.w   R5          ; R5 <-- R5/2
            rra.w   R5          ; R5 <-- R5/2
            ret
```

Label

to identify the subroutine

`ret` – return to exit from subroutine

A Simple Subroutine

The bigger picture

```
;-------------------------------------------------------------------
; Main loop here
;-------------------------------------------------------------------
            mov.w   #LENGTH-2, R4

read_nxt:   mov.w   array_1(R4), R5
            call    #div_by_16
ret_addr:   mov.w   R5, array_2(R4)

            decd.w  R4
            jhs     read_nxt

main:       jmp     main
            nop

;-------------------------------------------------------------------
; Subroutine: div_by_16
; Input:    16-bit signed number in R5 -- modified
; Output:   16-bit signed number in R5 -- R5 = floor(R5/16)
;-------------------------------------------------------------------
div_by_16:  rra.w   R5          ; R5 <-- R5/2
            rra.w   R5          ; R5 <-- R5/2
            rra.w   R5          ; R5 <-- R5/2
            rra.w   R5          ; R5 <-- R5/2
            ret
```

Main loop

After the ∞-loop

sub-routine

`ret` – return to exit from subroutine

Jumps vs `call`

With a jump:

- The program counter (PC) is updated to the address of the label
- Execution proceeds from that label

```
;-------------------------------------------------------------
; Main loop here
;-------------------------------------------------------------
            mov.w   #LENGTH-2, R4

read_nxt:   mov.w   array_1(R4), R5

            jmp     div_by_16

ret_addr:   mov.w   R5, array_2(R4)

            decd.w  R4
            jhs     read_nxt

main:       jmp     main
            nop

div_by_16:  rra.w   R5          ; R5 <-- R5/2
            rra.w   R5          ; R5 <-- R5/2
            rra.w   R5          ; R5 <-- R5/2
            rra.w   R5          ; R5 <-- R5/2
            jmp     ret_addr
```


Awful coding practice!

For demonstration purposes only!

DO NOT REPLICATE

Jumps vs `call`

With a **`call`** there is more

- The address of the next instruction in the calling program is **saved**

⇒ **Return address**

- The address of the subroutine is loaded into the PC
- The subroutine is executed
- After the **`ret`** instruction, the return address is **restored** into the PC
- Execution continues from this point in the calling function

```
;-------------------------------------------------
; Main loop here
;-------------------------------------------------
            mov.w   #LENGTH-2, R4

read_nxt:   mov.w   array_1(R4), R5
            call    #div_by_16
ret_addr:   mov.w   R5, array_2(R4)

            decd.w  R4
            jhs     read_nxt

main:       jmp     main
            nop

;-------------------------------------------------
; Subroutine: div_by_16
; Input:    16-bit signed number in R5 -- mod
; Output:   16-bit signed number in R5 -- R5 =
;-------------------------------------------------
div_by_16:  rra.w   R5          ; R5 <-- R5/2
            rra.w   R5          ; R5 <-- R5/2
            rra.w   R5          ; R5 <-- R5/2
            rra.w   R5          ; R5 <-- R5/2
            ret
```


Where is the return address saved?

The Stack

Shift and Rotate Instructions

Processors often offer three types of shifts and rotations

- **Logical Shift:** Inserts zeros for both right and left shifts
  Divide/Multiply by 2 for unsigned numbers

No Instruction in MSP430

- **Arithmetic Shift:** Insert zeros for left shifts
  Repeat the most significant bit for right shifts
  Divide/Multiply by 2 for signed numbers

- **Bit Rotation:** No bits inserted or lost – bits are moved out of one end of the register and passed around to the other end

Shift and Rotate Instructions

Arithmetic Shift/Roll Left

```
rla.w   dst          ; shift all bits left, insert 0
```

- You can use **`rla.w`** to multiply a signed/unsigned number by 2

Arithmetic Shift/Roll Right

```
rra.w   dst          ; shift all bits right, insert msb
```


- You can use **`rra.w`** to divide **a signed number** by 2
- Does not work with unsigned numbers!!

Shift and Rotate Instructions

Rotate Left Through Carry

```
rlc.w   dst
```

Rotate Right Through Carry

```
rrc.w   dst
```

Shift and Rotate Instructions

```
rla.w   dst       ; arithmetic shift left
rra.w   dst       ; arithmetic shift right
rlc.w   dst       ; rotate left through carry
rrc.w   dst       ; rotate right through carry
```

Syntax: These instructions have one operand

Even More Instructions

Operations on Bits in Status Register

```
clrc        ; clear carry bit                 C = 0
clrn        ; clear negative bit              N = 0
clrz        ; clear zero bit                  Z = 0
setc        ; set carry bit                   C = 1
setn        ; set negative bit                N = 1
setz        ; set zero bit                    Z = 1
dint        ; disable general interrupts      GIE = 0
eint        ; enable general interrupts       GIE = 1
```


Syntax: These instructions do not have operands. They act on the specific status bits in SR = R2

Coding Task

Task: Write a subroutine that performs unsigned division by 16 with following contract

```
;-------------------------------------------------------------------------
; Subroutine: div_by_16
; Input:    16-bit unsigned number in R5 -- modified
; Output:   16-bit unsigned number in R5 -- R5 = floor(R5/16)
;-------------------------------------------------------------------------
```


You can download Lecture_13.asm from Carmen and add your code