Lecture 11

Control Flow II: `if-else`

How to Avoid Spaghetti Code?

Announcements

Midterm 1 posted – due Wednesday February 22 before class

Task: Compute the mean absolute deviation (MAD) of a given set of numbers

Submission: PDF file with screenshots of code & results

Source code as txt – if your source code does not produce the results you claim, not good!

Office hours: Tuesdays 1 pm – 3 pm Dreese Lab 259

Quiz #3 – The Numbers

The array `twice` contains the numbers 6, 28, 496, 8128, 33550336

One possibility for expressing the nth number as $a_n = (2^{p_n} - 1)2^{p_n - 1}$

where p_n is the nth prime number

When $2^{p_n} - 1$ is a prime number then a_n is a **perfect number**

All proper divisors add up to the original number:

6 = 1 + 2 + 3

28 = 1 + 2 + 4 + 7 + 14

$2^{p_n} - 1$ is prime for $p_n = 2, 3, 5, 7, 13, 17, 19, 31, 61, 89, \dots, 82589933, \dots ?$

Any application?

None for perfect numbers.

Lots for prime numbers.

Just for fun.

Last time: Action

Task in many parts:

1. Create an array in RAM with values {1, 1, 2, 3, 5, 8, 13, 21}
2. Write a loop to add all numbers together
3. Modify the loop so that it does not add if value == 13

Bonus: Can you loop through the array from last element to first element?

These make the best loops!

4. Can you find the average of the given numbers?

- Add all values
- Divide by the number of values Here divide by 8 How?

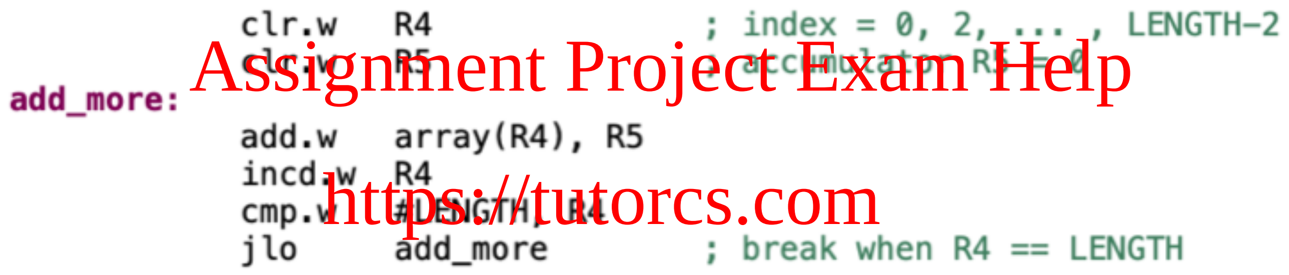
`rra.w`

Always keep an eye for signed/unsigned range and overflow

Last time: Action

Solving Part 2: Write a loop to add all numbers together

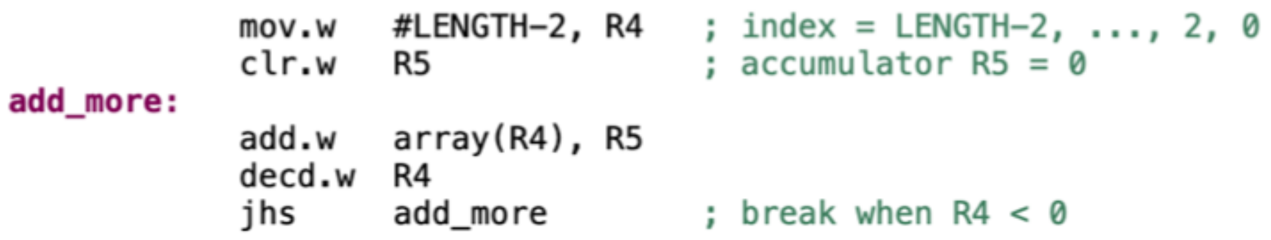
Counting Up



```
          clr.w   R4              ; index = 0, 2, ..., LENGTH-2
          clr.w   R5              ; accumulator R5 = 0
add_more:
          add.w   array(R4), R5
          incd.w  R4
          cmp.w   #LENGTH, R4
          jlo     add_more        ; break when R4 == LENGTH
```

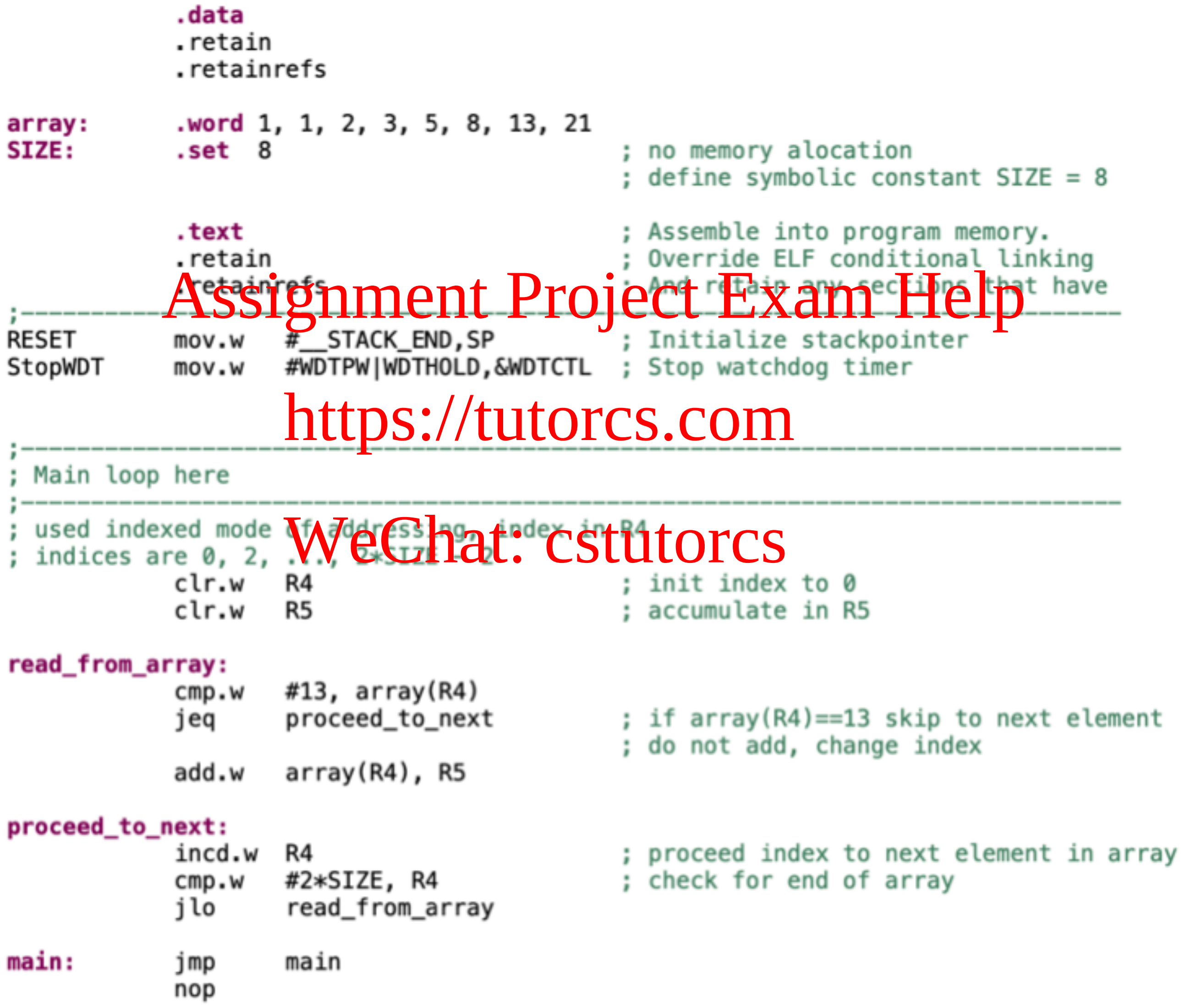
Both are great !

Counting Down



```
          mov.w   #LENGTH-2, R4   ; index = LENGTH-2, ..., 2, 0
          clr.w   R5              ; accumulator R5 = 0
add_more:
          add.w   array(R4), R5
          decd.w  R4
          jhs     add_more        ; break when R4 < 0
```

Solution

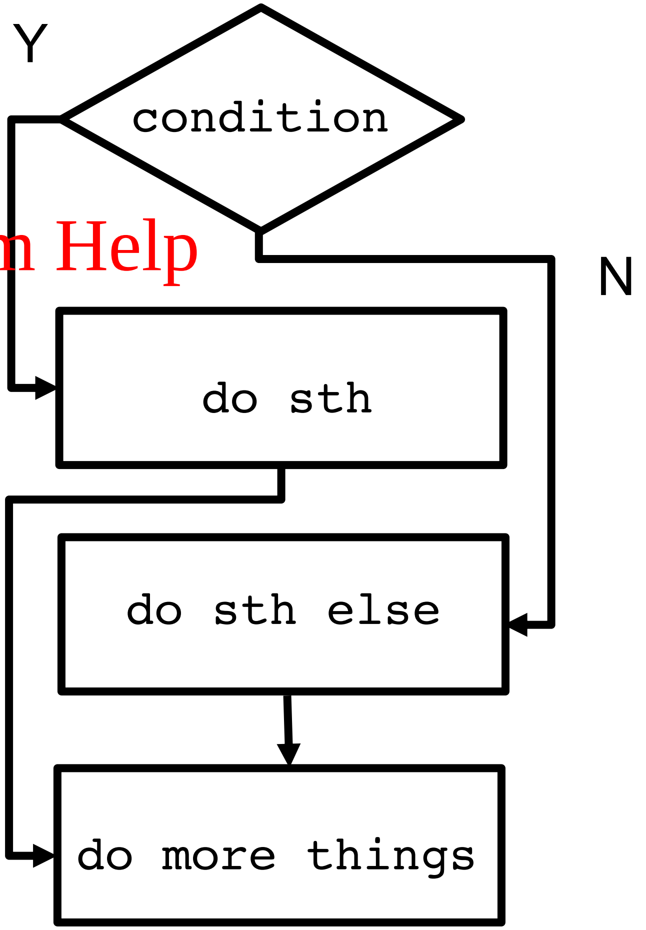


```
            .data
            .retain
            .retainrefs

array:      .word 1, 1, 2, 3, 5, 8, 13, 21
SIZE:       .set  8                         ; no memory alocation
                                            ; define symbolic constant SIZE = 8

            .text                           ; Assemble into program memory.
            .retain                         ; Override ELF conditional linking
            .retainrefs                     ; And retain any sections that have
;-------------------------------------------------------------------------------
RESET       mov.w   #__STACK_END,SP         ; Initialize stackpointer
StopWDT     mov.w   #WDTPW|WDTHOLD,&WDTCTL  ; Stop watchdog timer

;-------------------------------------------------------------------------------
; Main loop here
;-------------------------------------------------------------------------------
; used indexed mode of addressing, index in R4
; indices are 0, 2, ..., 2*SIZE-2
            clr.w   R4                      ; init index to 0
            clr.w   R5                      ; accumulate in R5

read_from_array:
            cmp.w   #13, array(R4)
            jeq     proceed_to_next         ; if array(R4)==13 skip to next element
                                            ; do not add, change index
            add.w   array(R4), R5

proceed_to_next:
            incd.w  R4                      ; proceed index to next element in array
            cmp.w   #2*SIZE, R4             ; check for end of array
            jlo     read_from_array

main:       jmp     main
            nop
```

if-else

if-else provides more control flow ...

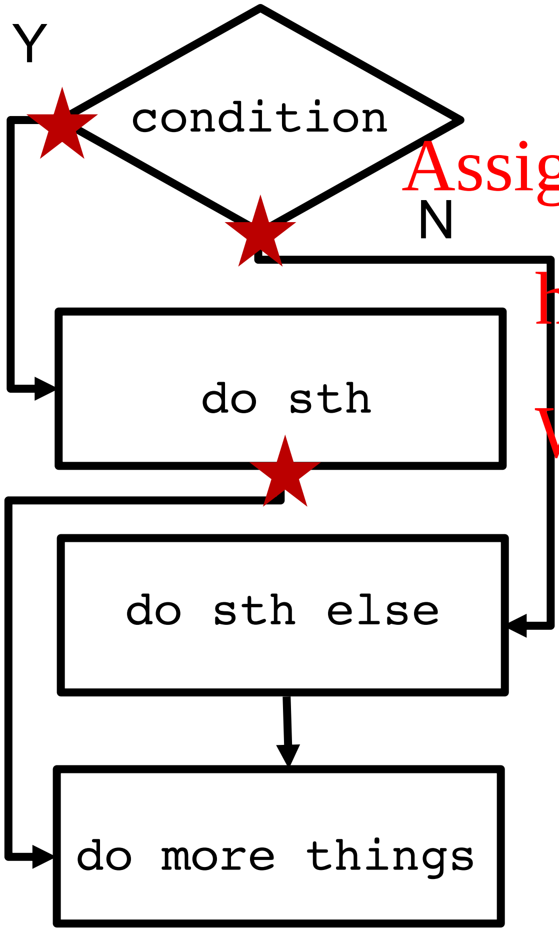
```
if (condition)
{
    do something
}
else
{
    do something else
}
do more things
```



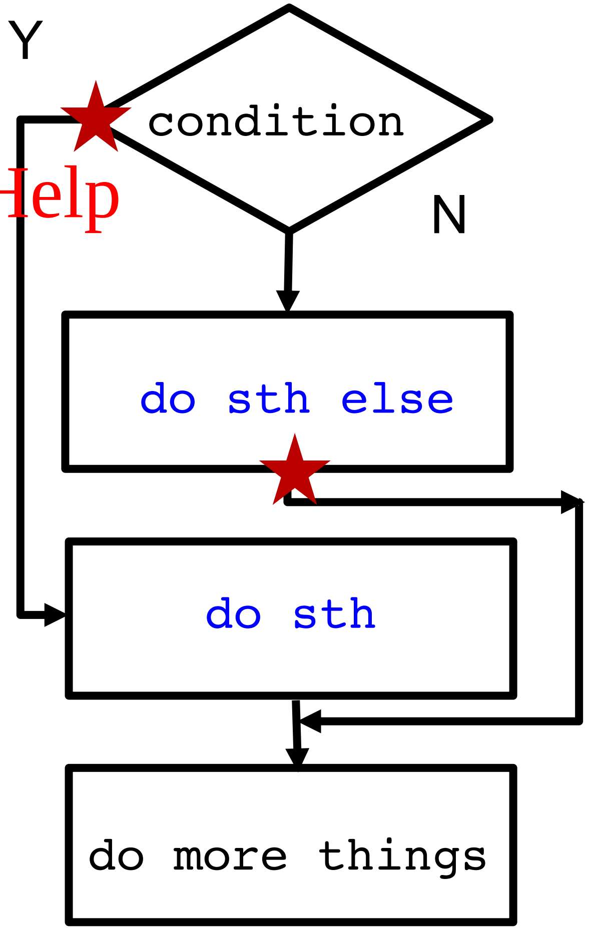
if-else results in more tangled spaghetti code

How to implement `if-else` – Opt. 1

Simply change the order of doing things: do sth else block first

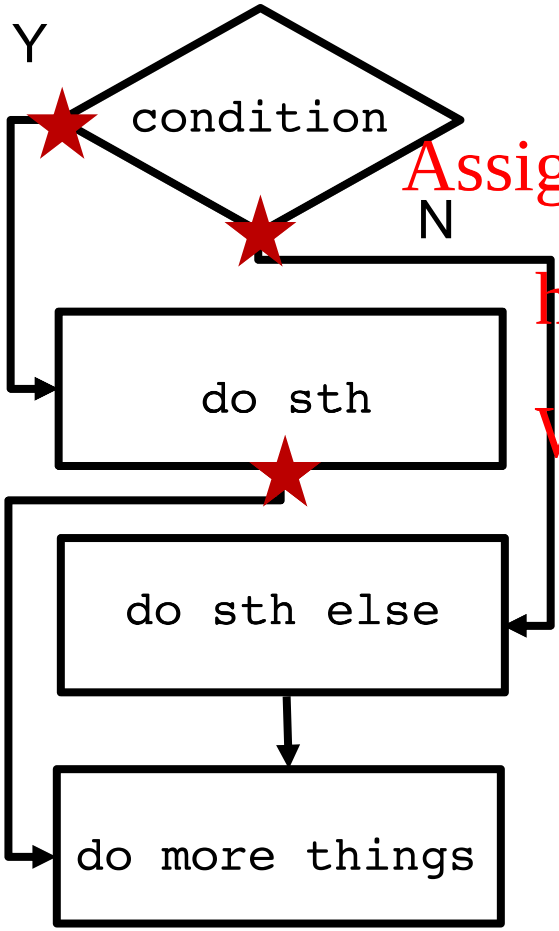


```
if (condition)
{
    do sth
}
else
{
    do sth else
}
do more things
```

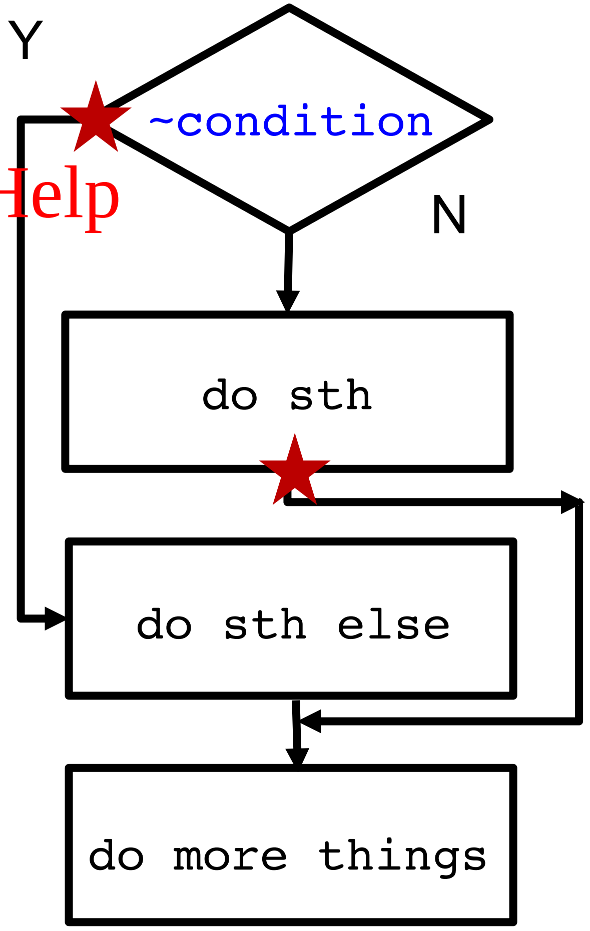


How to implement `if-else` – Opt. 2

We can negate the condition:



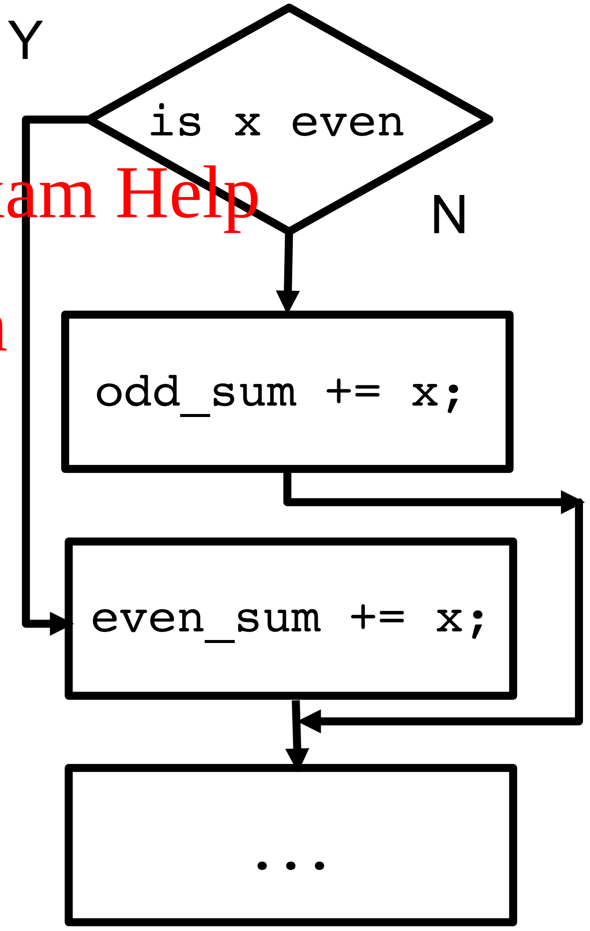
```
if (condition)
{
    do sth
}
else
{
    do sth else
}
do more things
```



Example `if-else`

Task: Given an array of integers find the sum of even and odd numbers

```
if (x is even)
{
    even_sum += x;
}
else
{
    odd_sum += x;
}
...
```



At this point, the flowchart is easy

How do we check if a number is even?

Excuse to learn more instructions

`bit.w`

decimal

When is a number even?

When the last digit is even – i.e., 0, 2, 4, 6, 8

When is a binary number even?

When the last bit is even – i.e., 0

How do we check this condition?

There is an instruction to check individual bits – called **bitwise test**

`bit.w` `src, dst`

`bit.w` is similar to compare, it does not change the value of `src` or `dst`

It only sets status bits (the `C` bit) according to (`src & dst`)

where `&` is bitwise and

bit.w

e.g.

$$C = \begin{cases} 1 & \text{if last bit of x is 1} \\ 0 & \text{if last bit of x is 0} \end{cases}$$

```
bit.b   #00000001b, x
```

We have two conditional jump instructions that check the carry bit explicitly

jc **jnc**

Typing binary numbers is cumbersome and prone to error – esp. with 16 bits

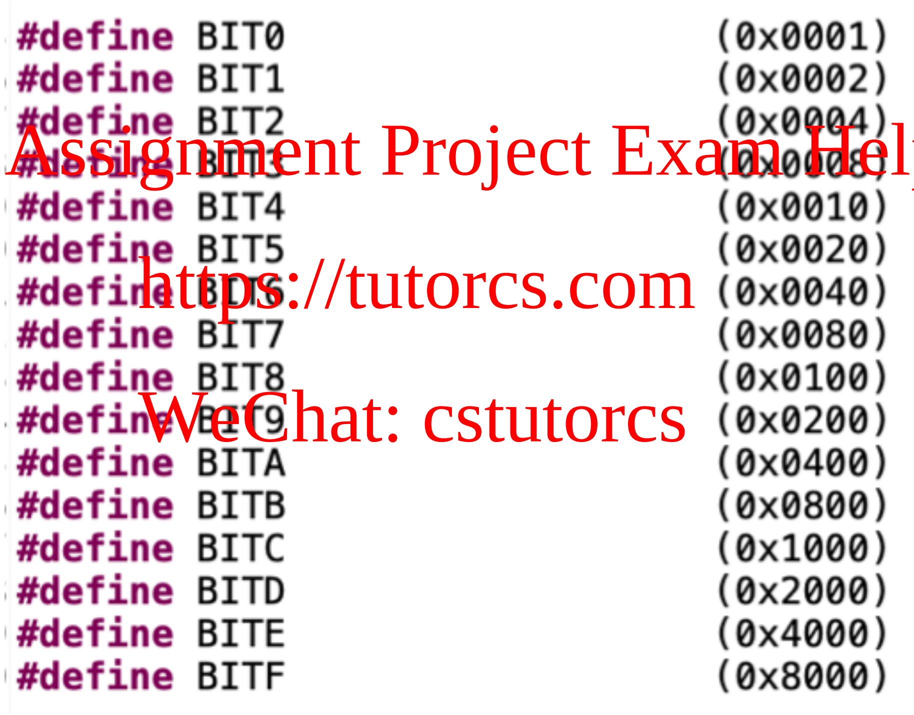
```
bit.w   #0000000000000001b, x        ; how many zeros??
```

Use the MACROs that are already defined in header file "msp430.h"

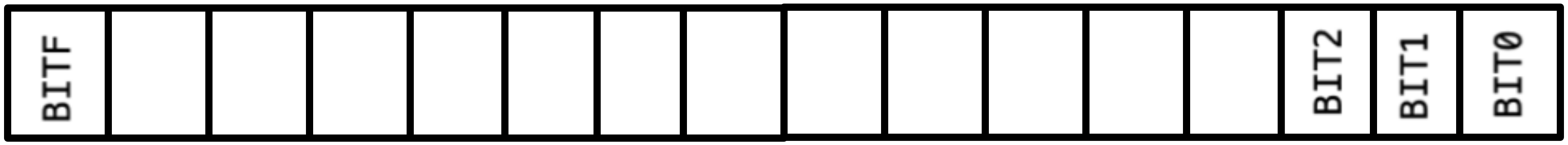
actually "msp430fr69891.h"

Bitmasks

We will use **bitmasks** for setting, clearing or testing bits defined in `“msp430fr69891.h”`

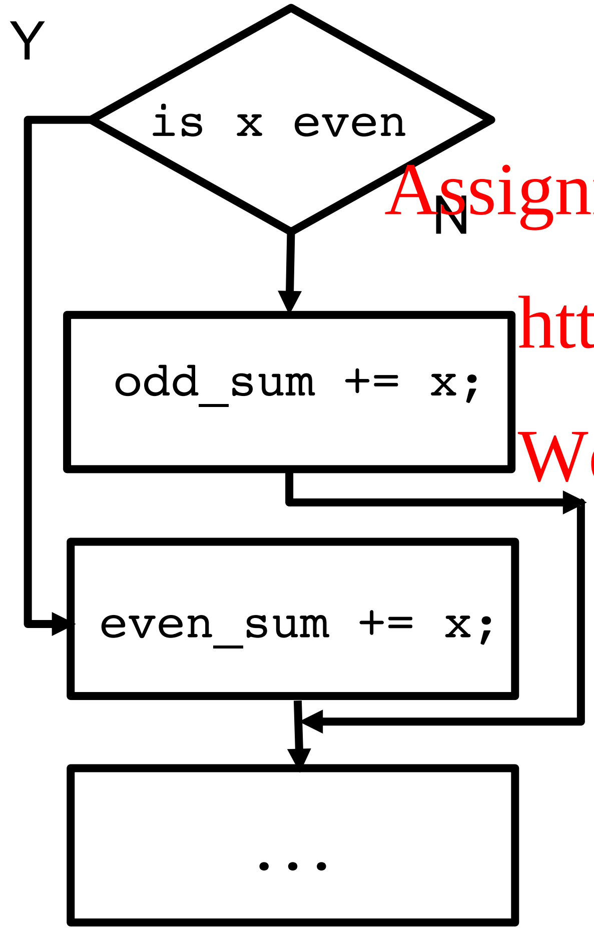


```
#define BIT0                (0x0001)
#define BIT1                (0x0002)
#define BIT2                (0x0004)
#define BIT3                (0x0008)
#define BIT4                (0x0010)
#define BIT5                (0x0020)
#define BIT6                (0x0040)
#define BIT7                (0x0080)
#define BIT8                (0x0100)
#define BIT9                (0x0200)
#define BITA                (0x0400)
#define BITB                (0x0800)
#define BITC                (0x1000)
#define BITD                (0x2000)
#define BITE                (0x4000)
#define BITF                (0x8000)
```



Back to our Example

Consider following pseudocode – how do we implement it in assembly?

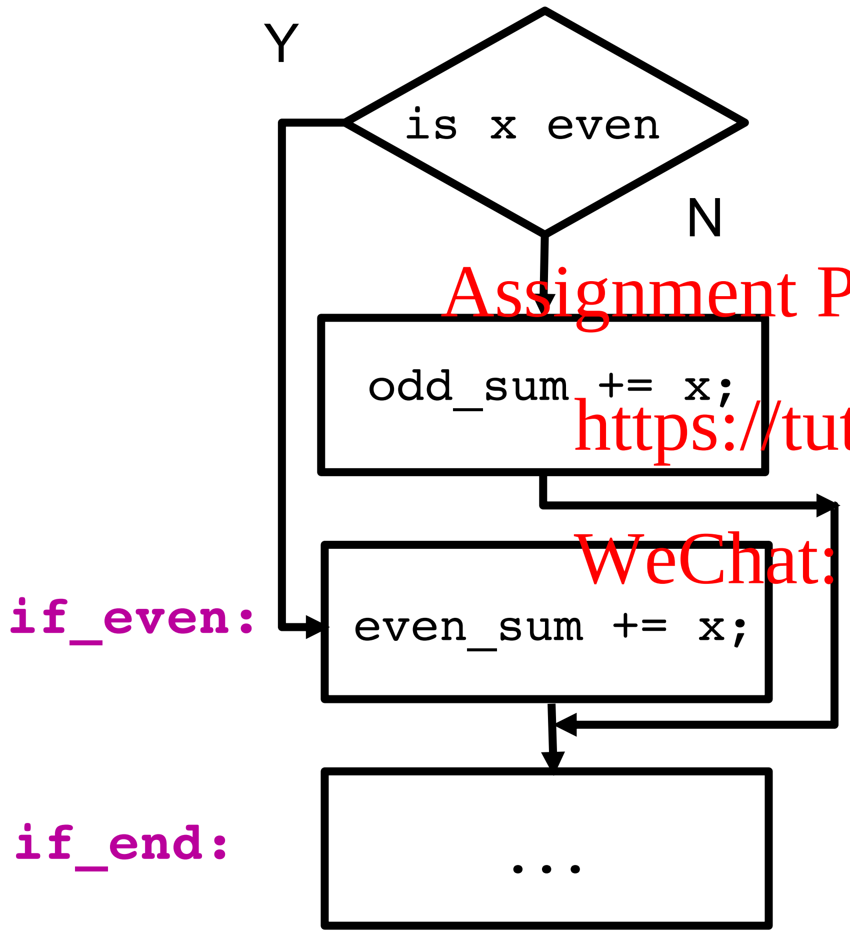


How do we check if a number is even?

```
bit.w   #BIT0, x
jc / jnc
```

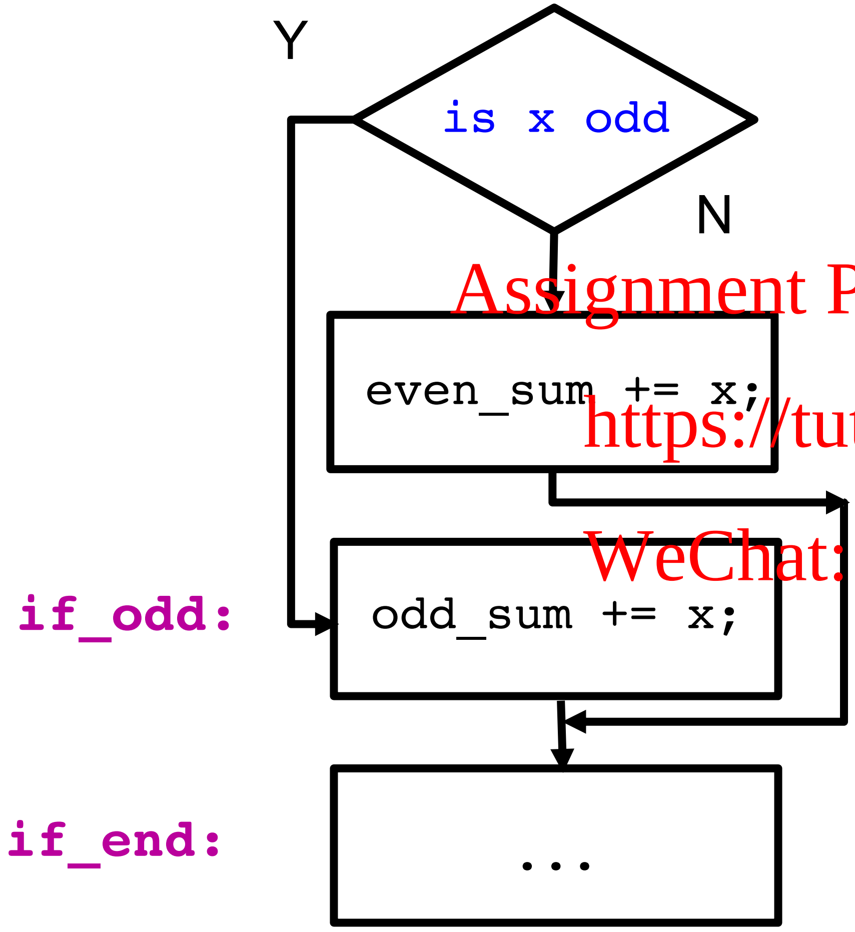
The rest we already know how to do

Back to our Example



```
          bit.w   #BIT0, x
          jnc     if_even

          add.w   x, odd_sum
          jmp     if_end

if_even:  add.w   x, even_sum

if_end:   ...
```

Alternate Implementation



```
         bit.w   #BIT0, x
         jc      if_odd

         add.w   x, even_sum
         jmp     if_end

if_odd:  add.w   x, odd_sum

if_end:  ...
```

More: Action

Task in many parts:

1. Create an array in RAM with values {1, 1, 2, 3, 5, 8, 13, 21}
2. Write a loop to add all numbers together
3. Modify the loop so that it does not add if value == 13
4. Can you find the average of the given numbers?

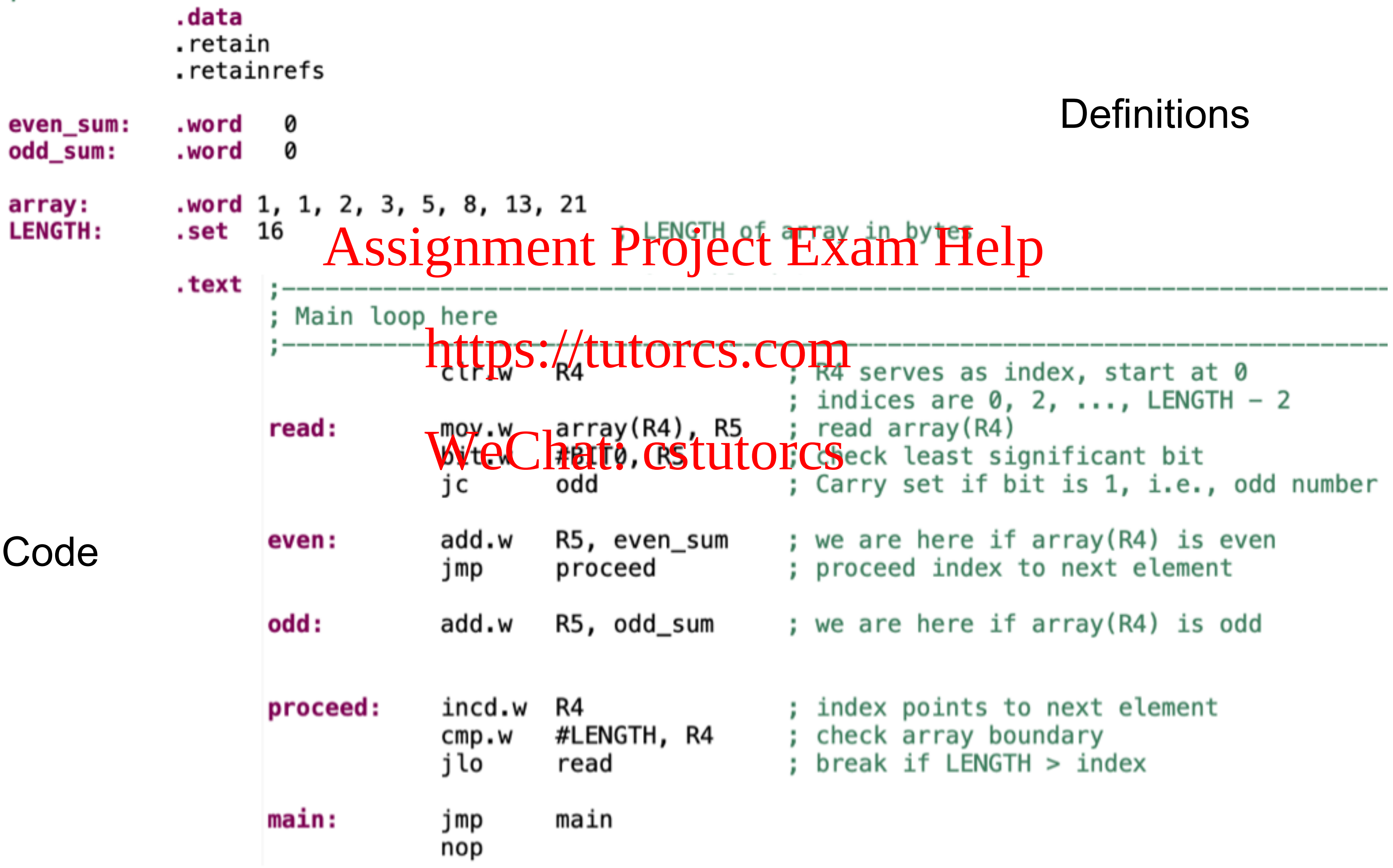
Today

- Define two variables even_sum and odd_sum in RAM
- Loop trough the array and find the sum of even and odd numbers

Solution

Definitions

Code



```
               .data
               .retain
               .retainrefs

even_sum:      .word   0
odd_sum:       .word   0

array:         .word 1, 1, 2, 3, 5, 8, 13, 21
LENGTH:        .set  16                        ; LENGTH of array in bytes

               .text
                     ;-------------------------------------------------------------------------------
                     ; Main loop here
                     ;-------------------------------------------------------------------------------
                                  clr.w     R4              ; R4 serves as index, start at 0
                                                            ; indices are 0, 2, ..., LENGTH - 2
                     read:        mov.w     array(R4), R5   ; read array(R4)
                                  bit.w     #BIT0, R5       ; check least significant bit
                                  jc        odd             ; Carry set if bit is 1, i.e., odd number

                     even:        add.w     R5, even_sum    ; we are here if array(R4) is even
                                  jmp       proceed         ; proceed index to next element

                     odd:         add.w     R5, odd_sum     ; we are here if array(R4) is odd


                     proceed:     incd.w    R4              ; index points to next element
                                  cmp.w     #LENGTH, R4     ; check array boundary
                                  jlo       read            ; break if LENGTH > index

                     main:        jmp       main
                                  nop
```