Lecture 10

Control Flow I: Loops and IFs

How to Avoid Spaghetti Code?

Announcements

Midterm 1 will be posted next Wednesday February 15

dnesday February 22 before class

What you need to know ng until the end of Lecture 11

- Instructions and addressing modes, esp. array addressing
- Conditional jumps and how to use them to control program flow
- Implementing loops and if statements in assembly

Practice opportunity: ~~Quiz 4 will be posted later today – due 2/15~~

- Posted Midterm 1 from last semester to Carmen, ungraded practice
- **Task:** Given an array of signed integers find the minimum positive number

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

Quiz #3 – A Coding Recipe

Task: Write a short program in assembly that reads the numbers stored in array `once`, doubles the values, and writes them to array `twice`.

- Initialize both arrays in memory using assembler directives. The array `once` should contain the values {0x3, 0xE, 0xF8, 0xFE0}. You can initialize the array `twice` to all zeros.

Let's start with the definition of the arrays/variables:

First question you should ask yourself is: **What *type* of variables?**

- Byte array or word array?
- Signed or unsigned integers (or other?)

16-bit unsigned integers
for given task – 0xFE0 !!

```
            .data                       ; Assembly into RAM
            .retain
            .retainrefs

once:       .word   0x3, 0xC, 0xF8, 0xFE0   ; 16-bit unsigned integer
twice:      .space  8                       ; 16-bit unsigned integer

            .text                           ; Assemble into FRAM
            .retain
            .retainrefs
```

Quiz #3

Task: Write a short program in assembly that reads the numbers stored in array `once`, doubles the v[illegible] writes them to array `twice`.

First thing to do: Create [illegible]art/pseudocode of your program

Given task is simple and with no tools to do loops (at the time of Quiz 3)

⇒ Simple, linear flowchart – even then there is quite a bit of choice

Read element from `once`, double by bit-shifting left, write to `twice` (x4)

or

Read element from `once`, write to `twice`, double by bit-shifting left (x4)

or

Add element from `once`, to element in `twice`, repeat addition (x4)

or

…

Quiz #3

Next thing: Decide on variables, addressing, registers etc. to use

Array addressing: Cons ed mode or indirect register modes

Works for source and destination

Works **only** for source

Since we will read from one array, write to another one we need an addressing mode that works for source and destination ⇒ **indexed mode**

Choose a core register for index: R4 is good, R10 is good

Never ever use R0 – R3

Working with **word** arrays with 4 elements each: **index will cover 0, 2, 4, 6**

- Start with `R4 = 0` and use `incd.w R4` to proceed to next element
- Start with `R4 = 6` and use `decd.w R4`

Quiz #3

We have everything we need to put our code together

```
;-------------------------------------------------------------------------
; Main loop here
;-------------------------------------------------------------------------

            clr.w   R4                  ; R4 is the index, start with R4 = 0

            mov.w   once(R4), twice(R4) ; copy once(R4) to twice(R4)
            rla.w   twice(R4)           ; double twice(R4)
            incd.w  R4                  ; proceed to next element
                                        ; addresses in word array grow by 2

            mov.w   once(R4), twice(R4) ; copy once(R4) to twice(R4)
            rla.w   twice(R4)           ; double twice(R4)
            incd.w  R4                  ; proceed to next element
                                        ; addresses in word array grow by 2

            mov.w   once(R4), twice(R4) ; copy once(R4) to twice(R4)
            rla.w   twice(R4)           ; double twice(R4)
            incd.w  R4                  ; proceed to next element
                                        ; addresses in word array grow by 2

            mov.w   once(R4), twice(R4) ; copy once(R4) to twice(R4)
            rla.w   twice(R4)           ; double twice(R4)
;           incd.w  R4                  ; here R4 = 8 stopping condition met!


main:       jmp     main
            nop
```

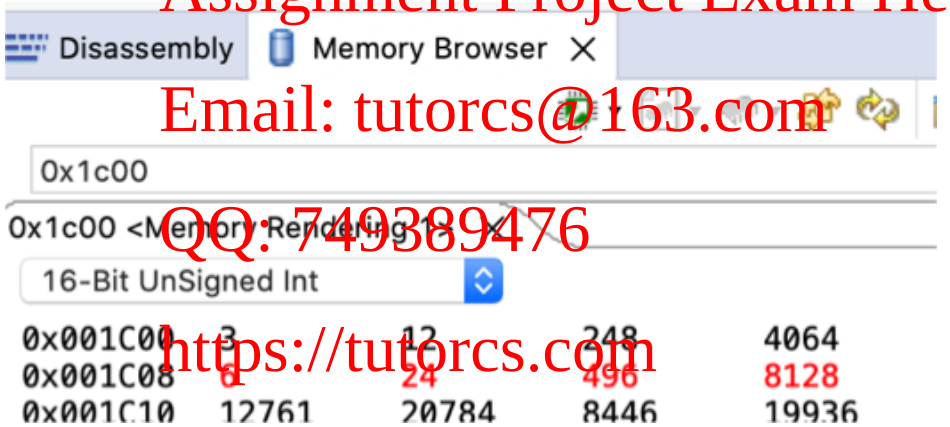
Quiz #3

But we are not done yet!!! We need to test the code!

Testing code is serious business – we will revisit it later

Testing *linear* code is easy – there is only one execution path, achieving complete path coverage is trivial

Run the code, look at the results:



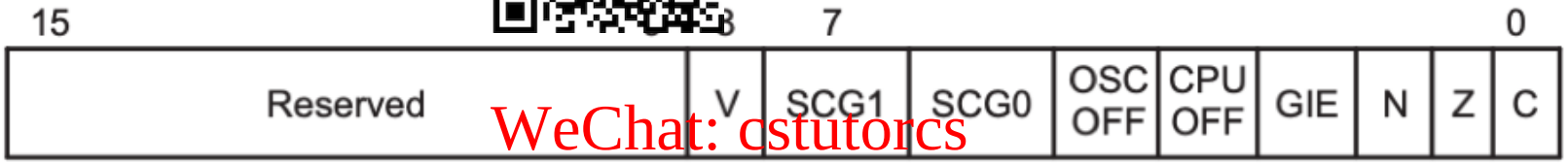
Good idea to check that you are not exceeding array boundaries

Recap: Conditional Jump Instructions

We use **conditional jump instructions** to control the flow of the program

Two step procedure:

1. An instruction (arithmetic instruction or explicit compare or test) is performed and sets the status bits C, N, V and Z



2. The conditional jump instruction controls the flow of the program based on the status bits

```
cmp.w   #101, R4
jlo     Repeat          ; if R4 < 101 jump to Repeat

tst.w   array(R4)
jn      if_neg          ; if array(R4) < 0 jump to if_neg
```

Recap: Conditional Jump Instructions

All you care is whether two values are **equal or not** `cmp.w src, dst`

`jeq`

You want to check for **orc**e., **>= or <** `cmp.w src, dst`

- with **signed values**

`jge` `jl`

- with **unsigned values**

`jhs` `jlo`

You care whether one value (e.g. result of operation or `tst.w dst`) is zero, nonzero, negative

`jz` `jnz` `jn`

You are working explicitly with the carry bit (e.g., `bit.w`) `jc` `jnc`

Today: Say no to Spaghetti Code

Loops and Conditionals (IF statements) in Assembly Language

- The goal is **not** to write an assembly program that accomplishes a given task
- There are many ways to write a program to accomplish a task and most of these end up in a tangle of labels and jumps – **Spaghetti Code**
- The goal is to come up with **easy to follow and minimal structures to implement loops and conditionals**

Writing C code and translating your C code to assembly is not the best recipe

- Instead, start with a good understanding of the algorithm, i.e., steps to do
- Develop the **right flowchart**
- Translate the flow chart into assembly code

In ECE 2560 you have to follow these steps whenever you write code

Simple Loops

Task: Add all integers from 1 to 100

in C:

```
sum = 0;
for (ii = 1; ii <= 100; ii++)
    sum += ii;
```

A `for` loop is not the best place to start when writing assembly code


```
for (expr1; expr2, expr3)
        action;
```

≡

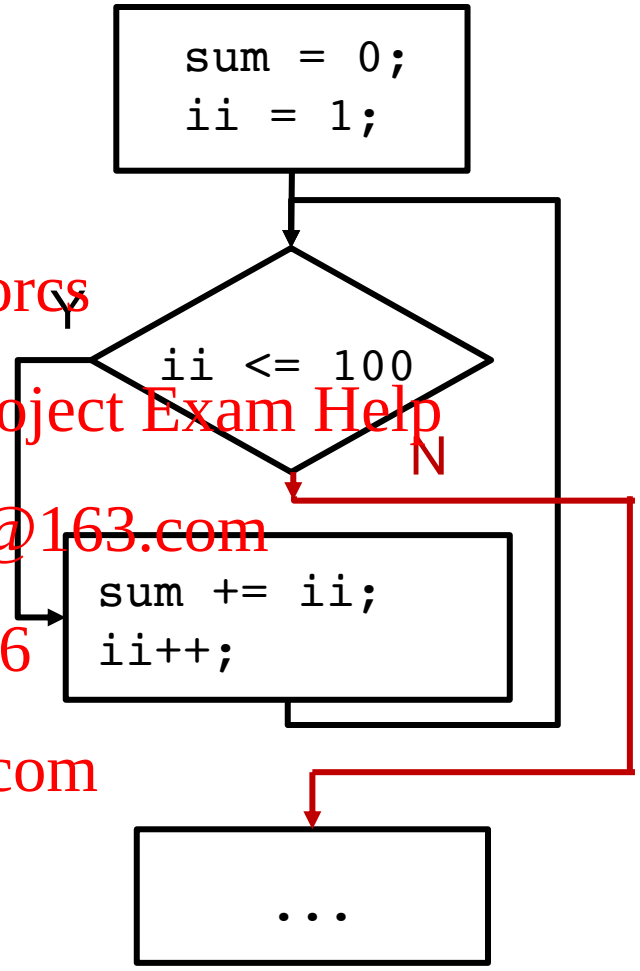
```
expr1;
while (expr2) {
    action
    expr3;
}
```

A `for` loop is anything but simple!

Simple Loops ??

Let's translate a **`while`** loop to a flowchart:

```
sum = 0;
ii = 1;
while (ii <= 100) {
     sum += ii;
     ii++;
}
```

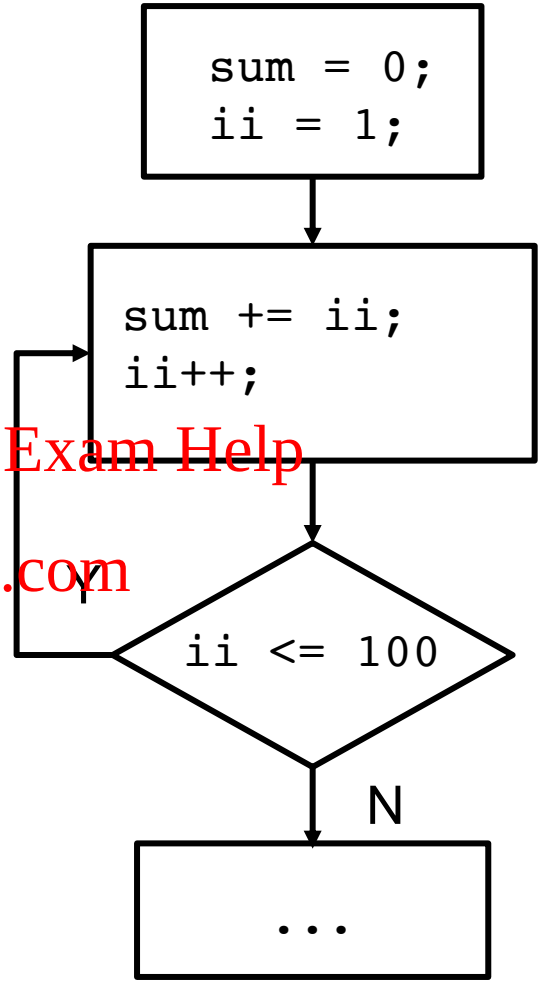


The definition of spaghetti code!

A Simple Loop

A **do-while** loop is simpler

```
sum = 0;
ii = 1;
do {
    sum += ii;
    ii++;
} while (ii <= 100);
```

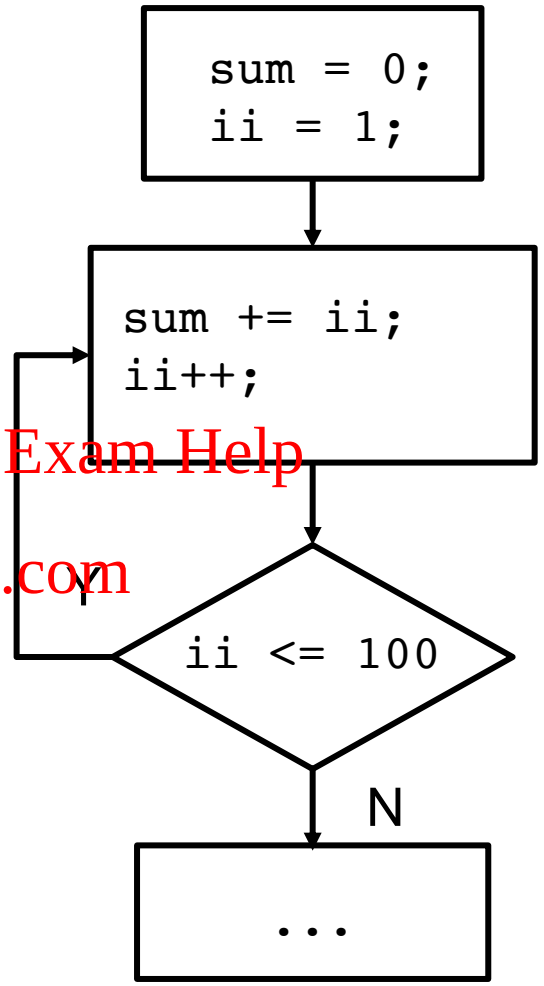


Note: In this flowchart you will add the first time without checking a condition
A loop like this does not always work – sometimes you do need to check the condition before performing the first iteration.
But when reading from a non-empty array this is the way to go!

A Simple Loop

The simplest loop by far is a **do-while** loop

```
sum = 0;
ii = 1;
do {
     sum += ii;
     ii++;
} while (ii <= 100);
```

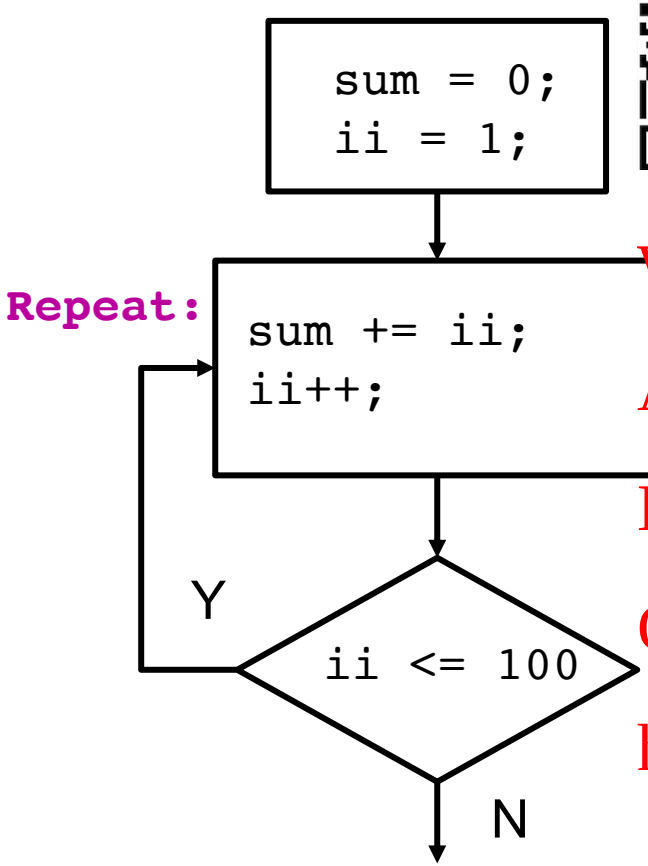


Step-by-step

- Execute/repeat the block of action at least once
- Check condition after each execution
- Repeat as long as condition is true

From Flowchart to Assembly Code

Task: Add all numbers from 1 to 100 …



rt with the **right flowchart**

ntify variables:

- `sum` ↔ `R5`
- `ii` ↔ `R4`

- **Identify jumps** in the flowchart, **add labels**
- **Decide on which conditional jump to use**
  - Order comparison Index is unsigned!!
  - Two possibilities

```
jhs (>=)      cmp.w  R4, #100
jlo (<)
```

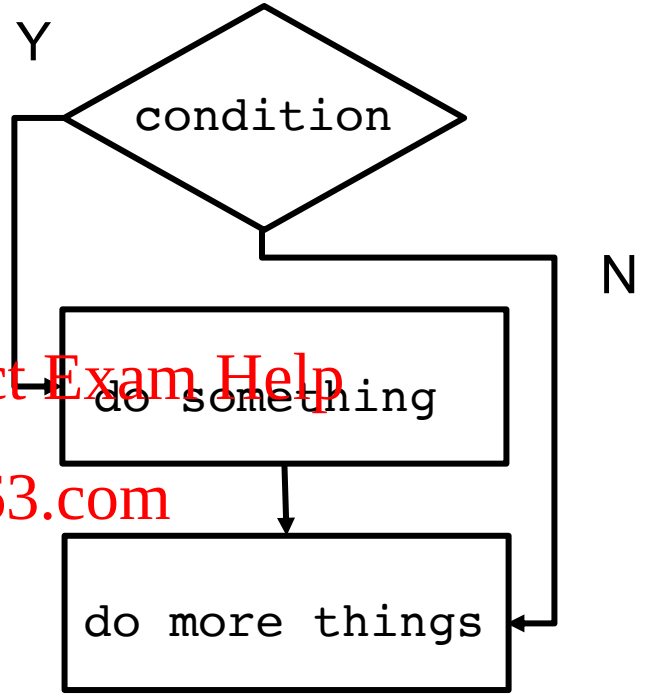
  - Modify condition: `ii < 101`

```
cmp.w  #101, R4
jlo    Repeat
```

The Simple `if` Statement

`if` statements are the simplest form of control flow

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



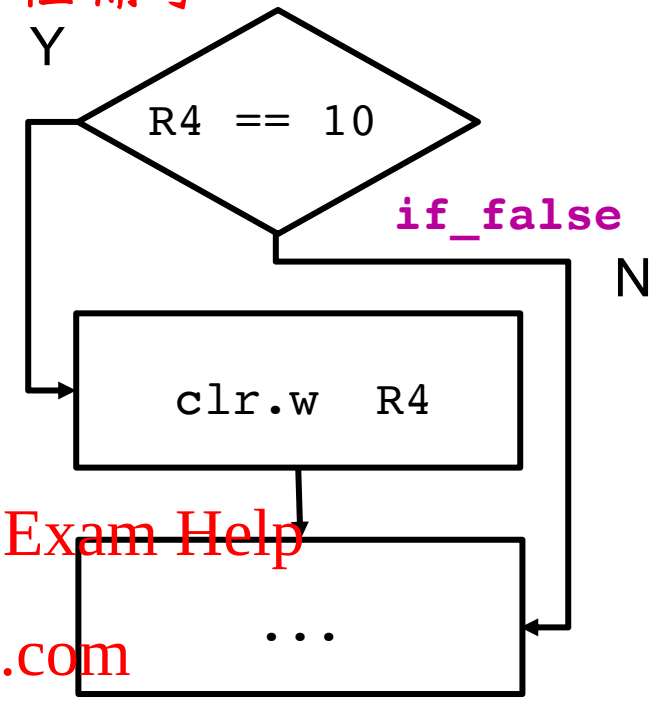
Such an implementation gives correct results

BUT

Creates a tangle of jumps – spaghetti code

How **not** to implement an `if` Statement


```
if (R4 == 10)
{
      clr.w   R4
}
inc.w   R4
```

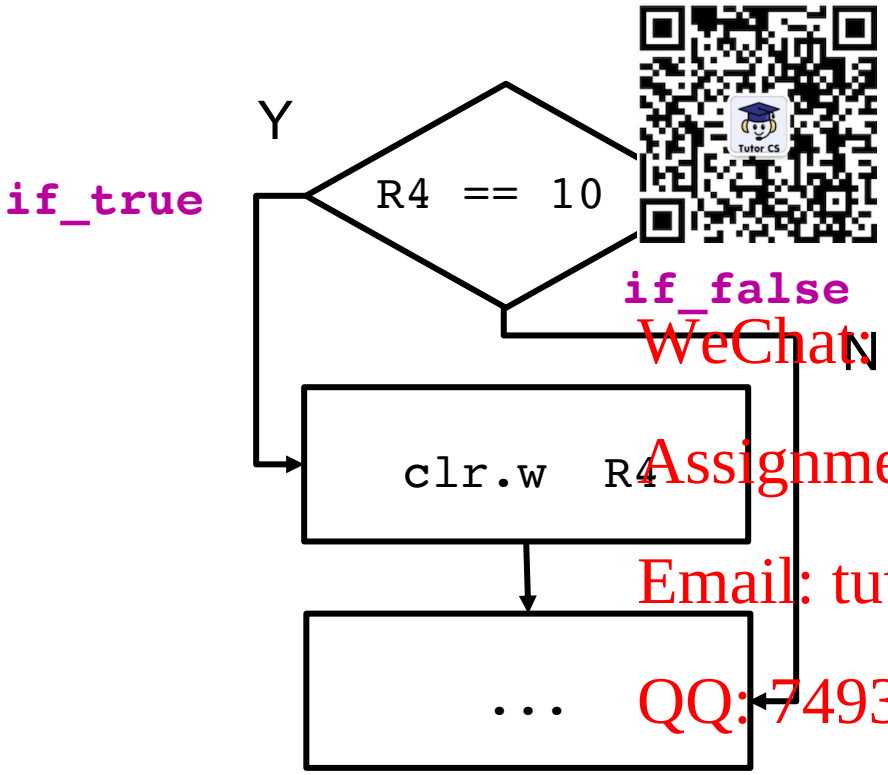


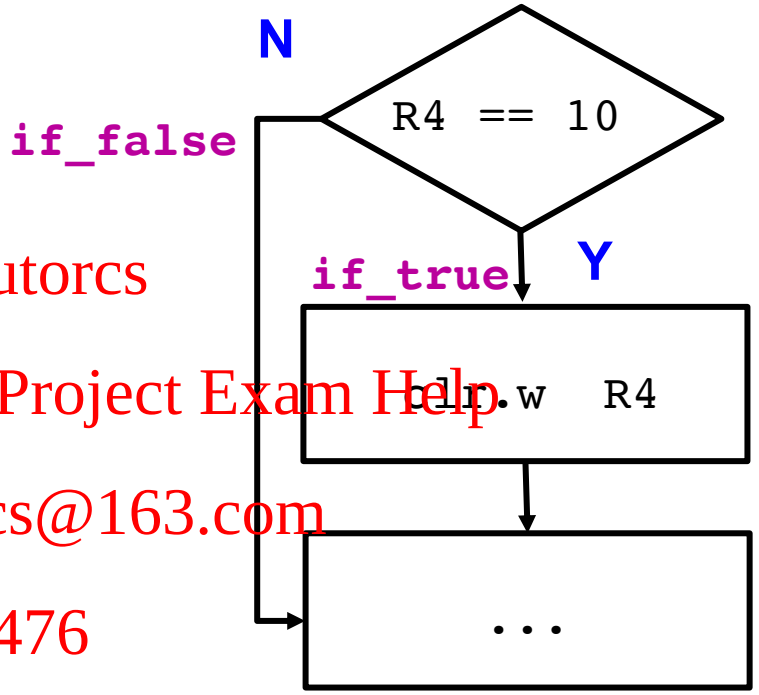
Assembly Implementation

```
        cmp.w   #10, R4
        jeq     if_true
        jne     if_false

if_true:    clr.w   R4

if_false:   inc.w   R4
```


How to implement an `if` Statement

Two ways to fix it: **(1) Be flexible with the placement of Y and N in the flowchart**





Requires two jumps:
jump if (R4 == 10) to if_true
jump if (R4 != 10) to if_false

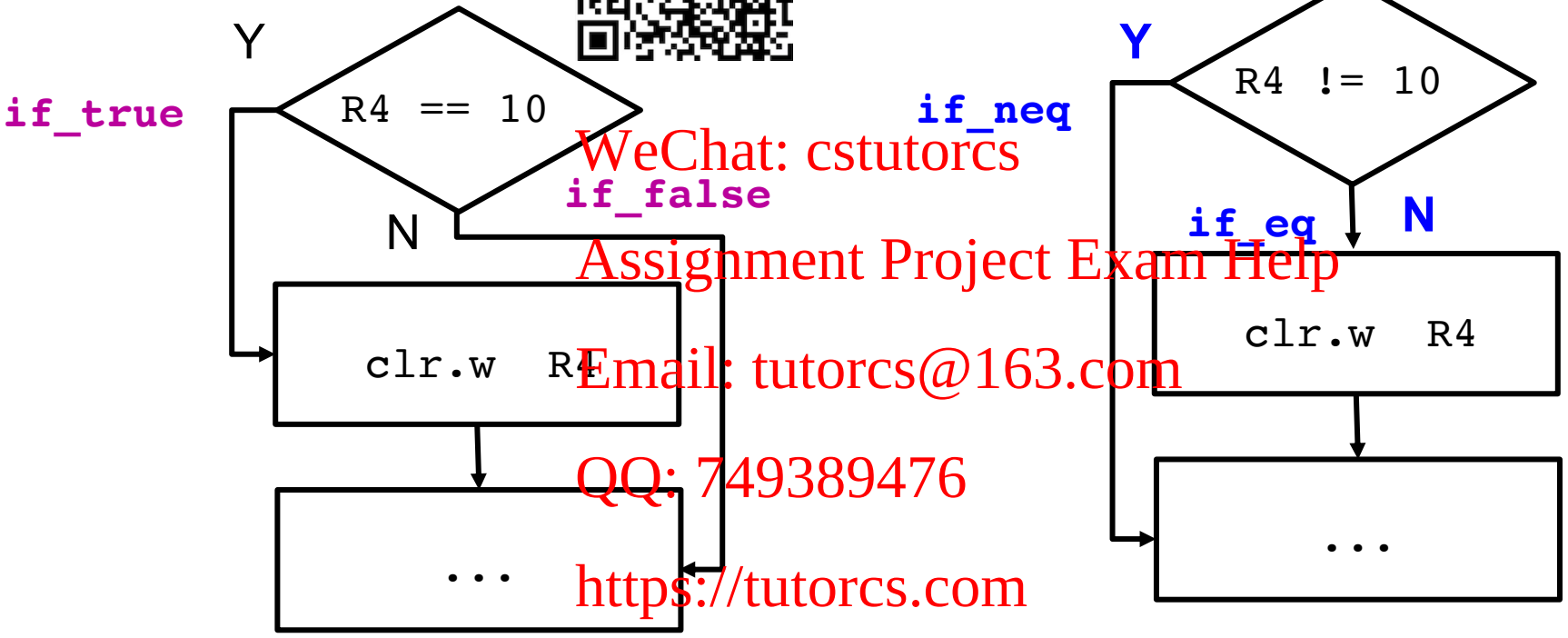
Requires one jump:
jump if (R4 != 10) to if_false

Note: the jump condition changed!

How to implement an `if` Statement

Two ways to fix it: **(2) Use the complementary condition** leave Y, N where they are

Instead of check for equality !!!

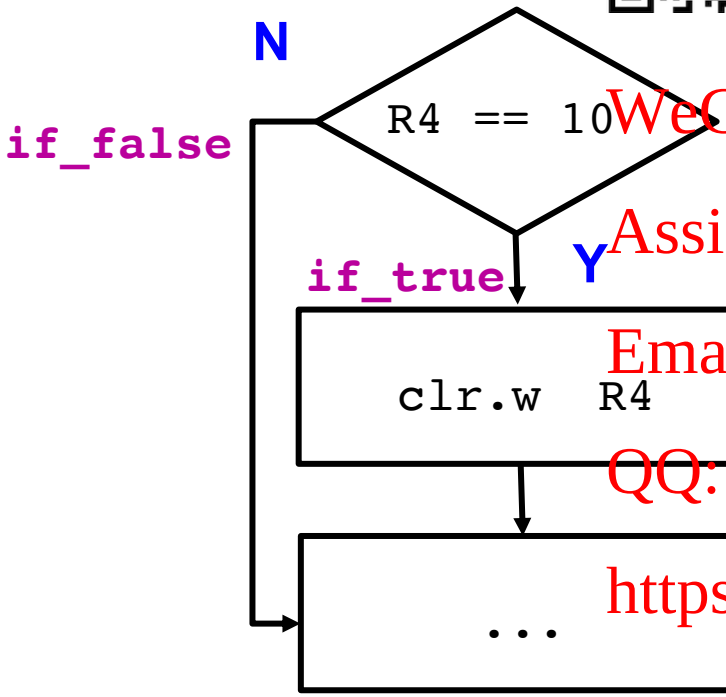


Note: The meaning of the labels change !!!

How to implement an `if` Statement

Two ways to fix it: **(1) Be flexible with the placement of Y and N in the flowchart**

es the conditional jump to the
ntary one – i.e., `jeq` becomes `jne`



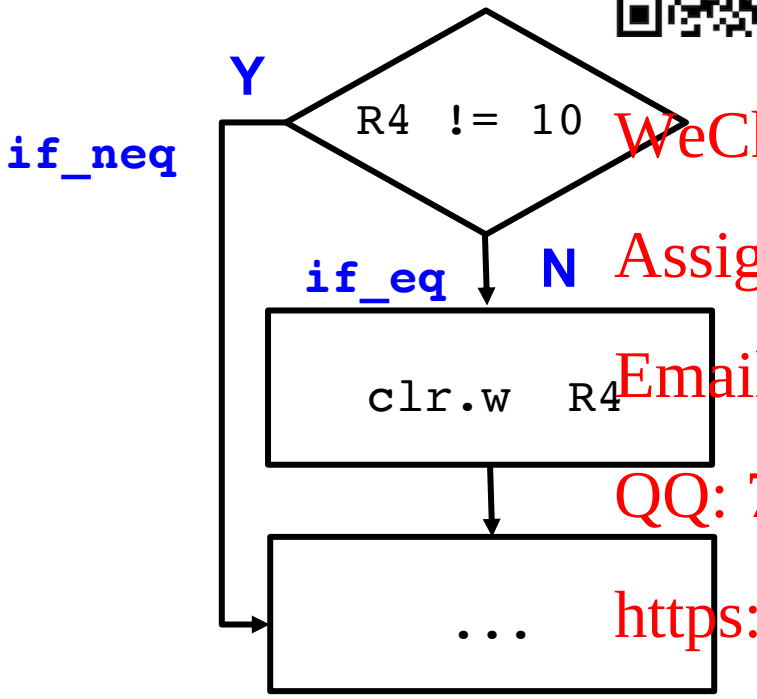
Assembly Implementation

```
        cmp.w   #10, R4
        jne     if_false
if_true:  clr.w   R4
if_false: ...
```

How to implement an `if` Statement

Two ways to fix it: **(2) Use the complementary condition** leave Y, N where they are

Instead of ... check for inequality !!!



Assembly Implementation

```
        cmp.w   #10, R4
        jne     if_neq
if_eq:  clr.w   R4
if_neq: ...
```

Action

Task in many parts:

1. Create an array in RA[illegible]lues {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 13

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

These make the best loops!

Solution


```
            .data
            .retain
            .retainref

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

            .text                     ; Assemble into program memory.
            .retain                   ; Override ELF conditional linking
            .retainref                ; 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
```