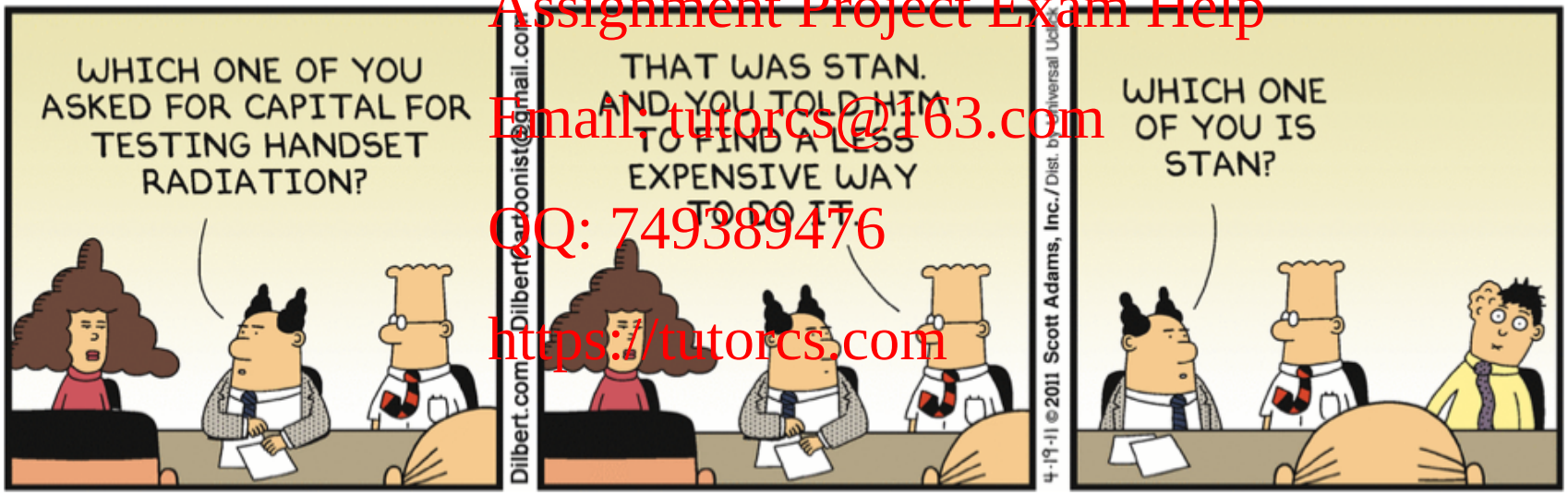
Lecture 23

Testing & Problem Solving



First the Daily Joke

There are 10 types of people in this world.

Those that understand binary, and those who don't.

Midterm #2 Posted

Due Wednesday April 12 – No extension unless there is a documented excuse for the entire we

- Task th
- Builds
- Will dis

king LEDs

The only a

- A codin
- More th

ECE 2560 Intro to Micr ystems Spring 2023

dterm #2

Due: Wednesday, April 12 – by 4:10 PM (Before Class)

No Late Submissions

Submission: Media Recording and File Uploads to Carmen

This quiz is individual work. You are not allowed to collaborate with others including AI.

Part 1: Coding task: Smart Blinky – teaching the LEDs how to count

Your program should start with both LEDs off (i.e. not emitting light) and wait for a push button to be pressed.

Your program will count the times S1 is pressed before pressing S2 and store this number in a variable in RAM. When S2 is pressed, the LaunchPad responds by blinking both LEDs the number of times S1 was pressed, and enters a new cycle. If S2 is pressed consecutively, with no presses of S1 in between, the LEDs will not blink. Use a call to following subroutine between blinks

```
delay:  clr.w   R5
countdown
        dec.w   R5
        jnz     countdown

        ret
```

Please watch the short demo video posted to Carmen exhibiting the desired behavior.

How to Test and Debug Code?

You write and run some code, see the desired results ⇒ A promising start

Testing **IS NOT** running the code you wrote and checking the final results

Sometimes you do not even know what results to expect

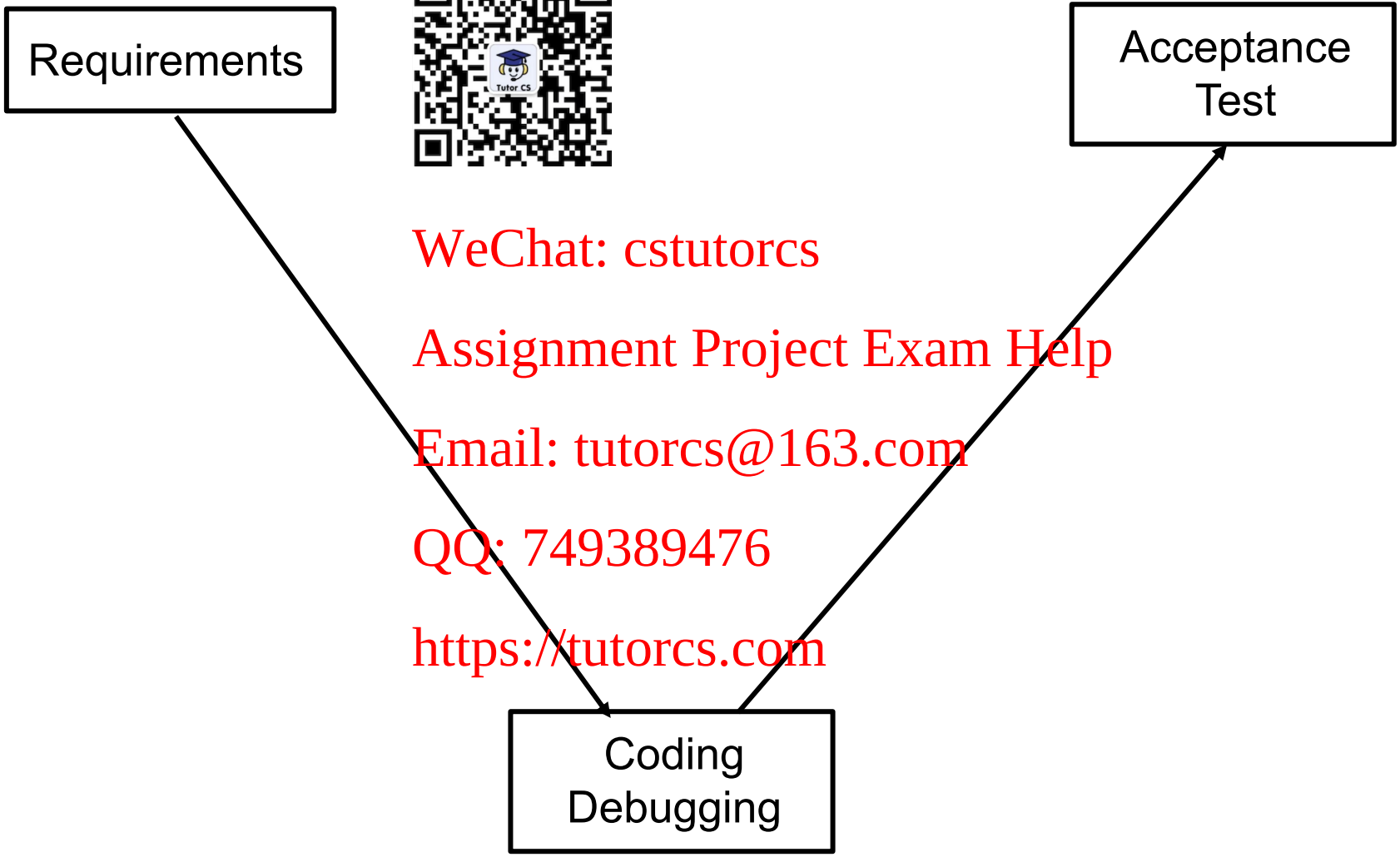
Example: In the project you were asked to add two subroutines `signed_x_times_y` and `inner_product` to make pre-existing code work without knowing the expected outcome of the program

What to do?

- Unit tests
- Integration tests

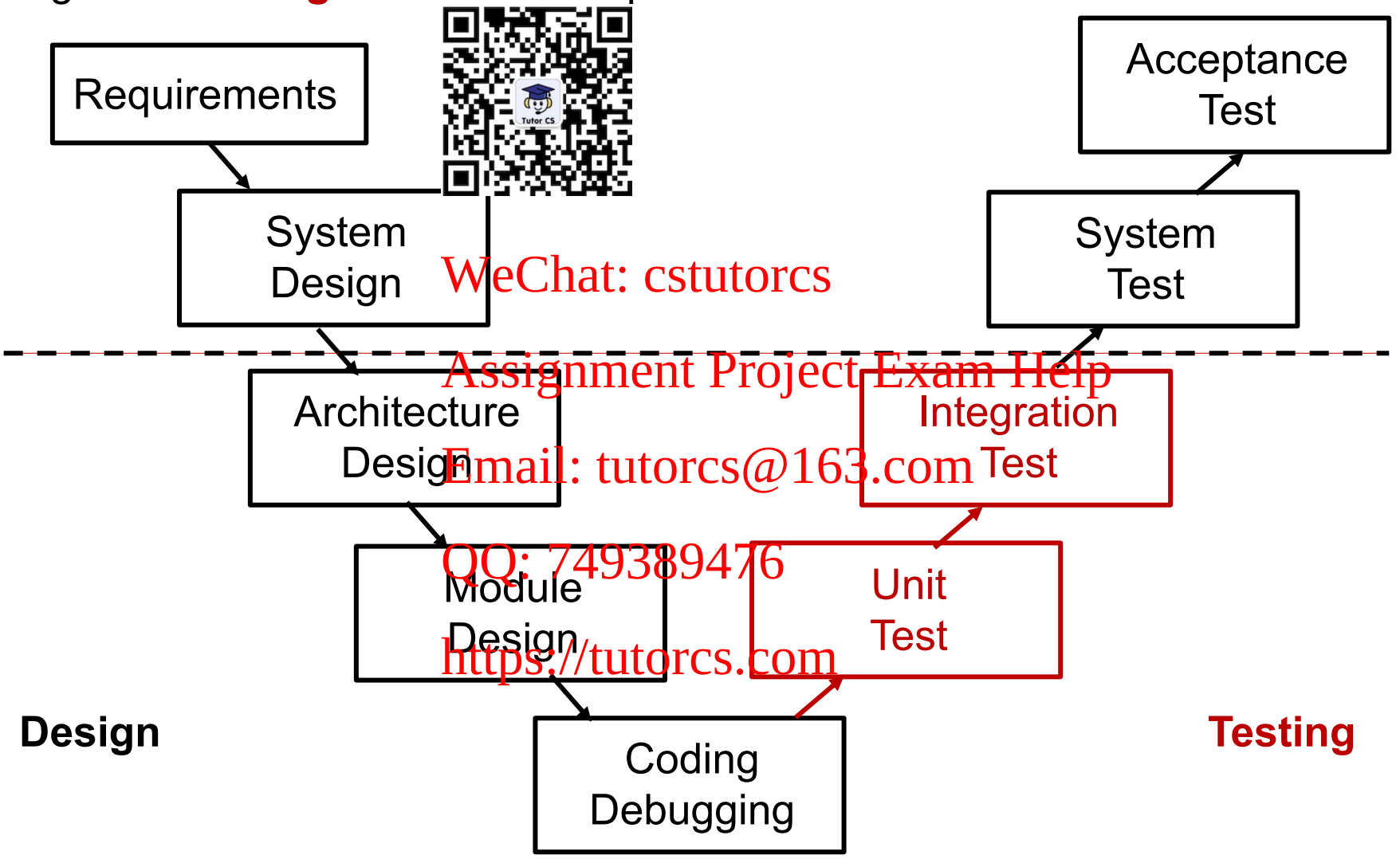
Test V or V – Model

Just writing & running the code and checking the results misses many steps



Test V or V-Model

Design and **testing** is done in multiple levels



What does this mean for us?

When you write a module, e.g., a subroutine or an interrupt service routine, you test it independently (**Unit test**) on task

e.g., project asked you to write subroutines

- `signed_x_times_y` to find the product of two signed numbers
- `inner_product_Qm` to find the inner product of two vectors/arrays

Goal is to make sure that the subroutines work

- for ANY set of numbers – as long as there is no **overflow**
- for vectors of ANY starting address ANY length and ANY Q-value

You create **unit tests** to confirm the input output behavior of the subroutine

i.e., run the subroutine using carefully selected input: **test cases**

aiming for good path coverage and range of tested values

What does this mean for us?

Unit tests confirm individual modules (subroutine or ISR) work correctly

Only part of the task

Also need to run **integrat** to confirm that modules continue to work correctly together

This requires that the subroutines closely follow the contract

- Input / output
- Use patterns of registers and other memory elements

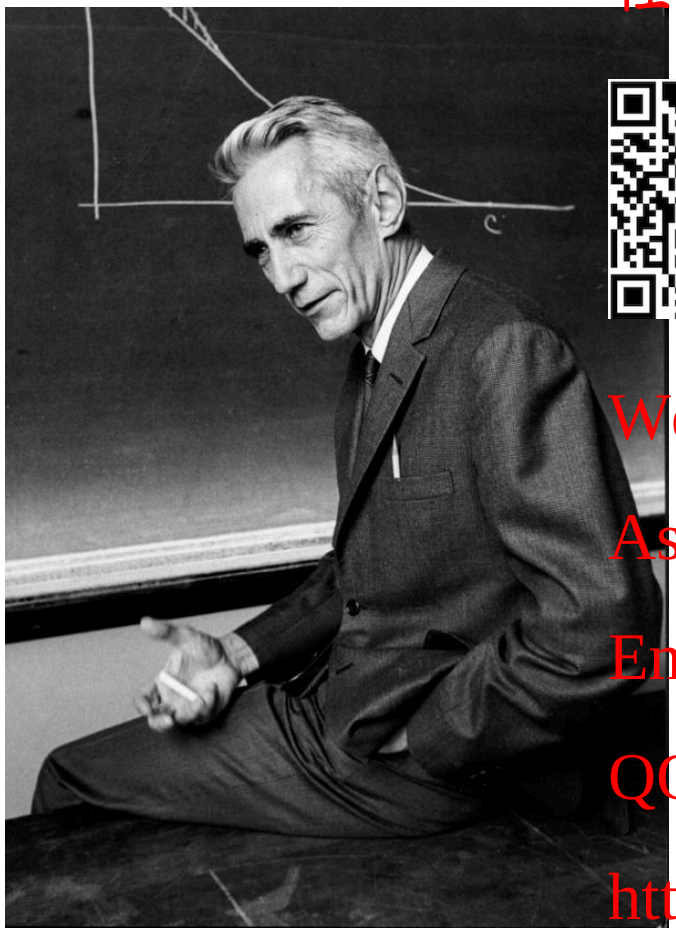
Ideally, devise tests to confirm

If we can show that the subroutines deliver correct results independently and in tandem, then they will most likely work correctly as part of the given program – at least for a program that is of the complexity of the project

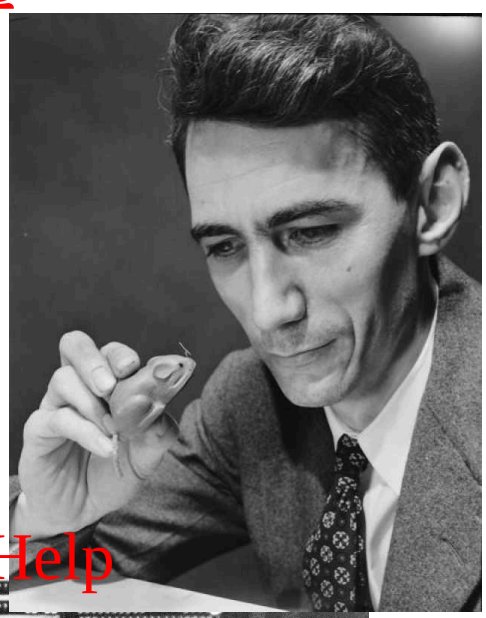
For more complex systems, we need to run **acceptance tests** as well

Actually, I have designed an acceptance test for you as part of the project: the bits and projections you were asked to provide in the screenshot

Claude E. Shannon



Do you know this gentleman?



Claude Shannon

Father of Information Theory

"A Mathematical Theory of Communication" (1948) Bell Labs

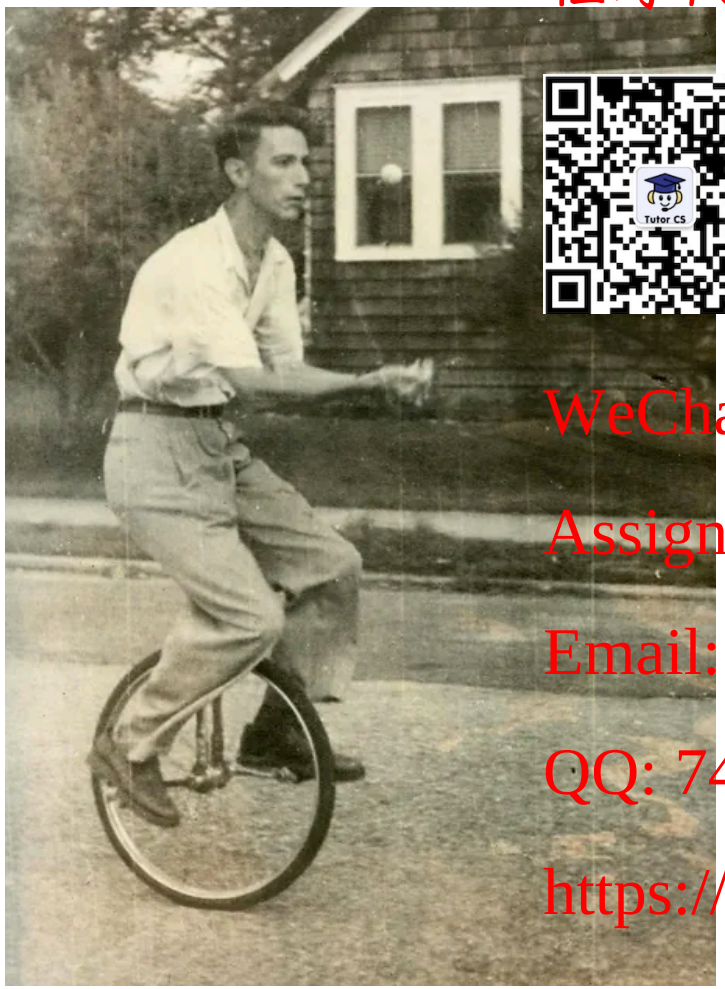
"A Symbolic Analysis of Relay and Switching Circuits" MS Thesis (1937) MIT

"Possibly the most important, and also the most famous, master's thesis of the century"

Lays down the foundations of electronic computing

"Communication Theory of Secrecy Systems" (1949)

Claude Shannon



- He was a unicyclist, juggler, chess player
- Tinkerer
  - Juggling machine
  - Chess Playing Computer
  - Shannon's mouse Theseus – first experiments in AI

- Book: **A Mind at Play: How Claude Shannon Invented the Information Age**

- Documentary: **The Bit Player** (2018)

- Great insights on creativity in problem solving:

- Search for **Shannon + "Creative Thinking"**

And he was a first class problem solver

Revisiting Learning Objectives

Official Course Goals

- Learn the architecture, programming, and interface requirements of a commercially used microprocessor
- Learn to interface a microcontroller to memory, parallel & serial ports, etc.
- Learn to apply microcontroller systems to solve real-time problems

My Version of Course Goals

- Learn to debug and test code
- Practice and develop **clear thinking skills**

- Practice and develop **problem solving skills**

But sometimes teaching seems to undermine creative problem solving skills

Education tries to create one type of thinking where questions already suggest the answers – with not much room for exploration

How to solve a problem?

Sometimes it is easy to fall into the trap of the suggested solution and then there is the dan[illegible]sing a **better** solution

Take the **Barometer Pro**[illegible]

”Show how it is possible to determine the height of a tall building with the aid of a barometer.”

Can you think of a different solution?

Good problem solving requires **thinking** rather than reacting with what we think is the desired answer

- It often requires thinking outside the constraints

- Not listening to the instructor
- Not following the instructions or hints

- Being rebellious

Back to MCU

Remember the joke of the day:

There are 10 types of people in this world.

Those that understand 2's complement signed numbers, and those who don't.

2's Complement Signed Numbers

Given a number x

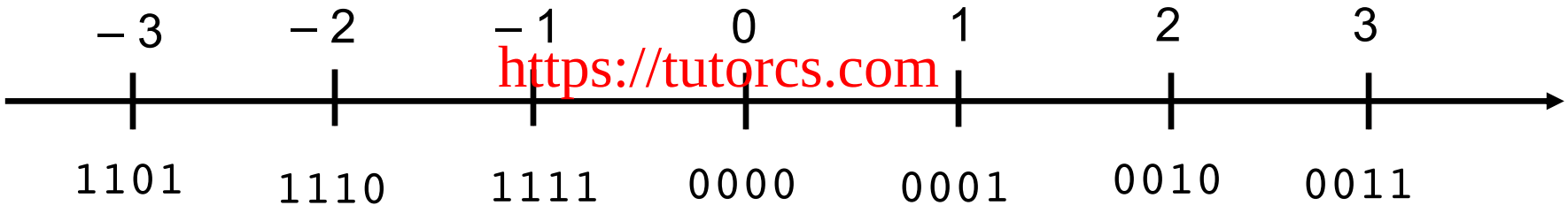
- If the number is positiv[illegible]nt it with the binary numeral for **x**
- If the number is negati[illegible]ent it with the binary numeral for **2^n - |x|**

We have seen that n-bit addition and subtraction works as usual with 2's complement signed numbers

- Successors and predecessor relationships are consistent as long as the numbers are restricted to n-bits

$(1)_{10} = 0001$ $(2)_{10} = 0010$ $(3)_{10} = 0011$

$(-1)_{10} = 1111$ $(-2)_{10} = 1110$ $(-3)_{10} = 1101$



2's Complement Signed Numbers

What about multiplication?

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^n - |x|**

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

Binary representations will be **x** and **2^n - |y|**

$2^n x - x|y|$

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

Binary representations will be **2^n - |x|** and **2^n - |y|**

These are the result in the n-bit register

$2^{2n} - 2^n(|x| + |y|) + |x|\,|y|$

⇒ **Multiplication works the same way for signed & unsigned numbers**