Lecture 19

GPIO

General Purpose Input/Output

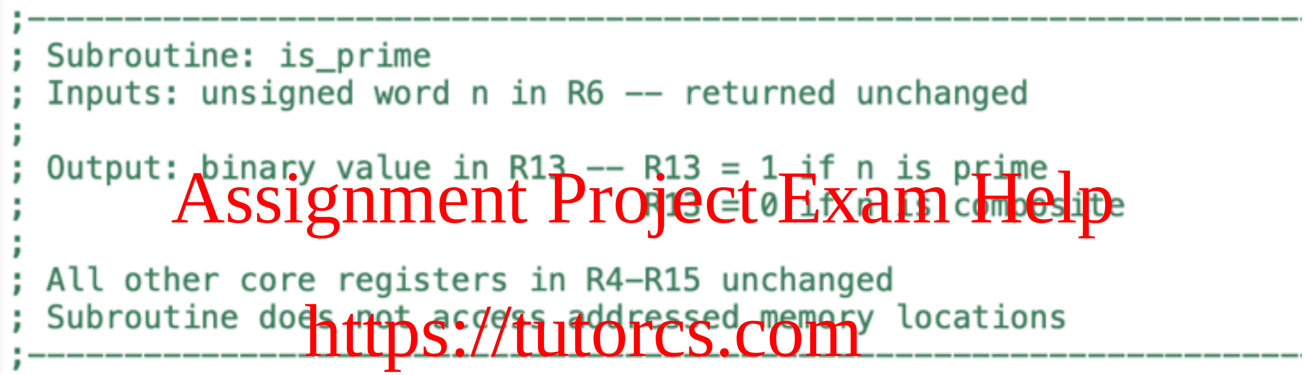
Bad joke of the day:

Why are assembly programmers always wet?

Because they work below C-level

Solution to Quiz 5

Task: Write a subroutine that checks whether a given integer n is prime

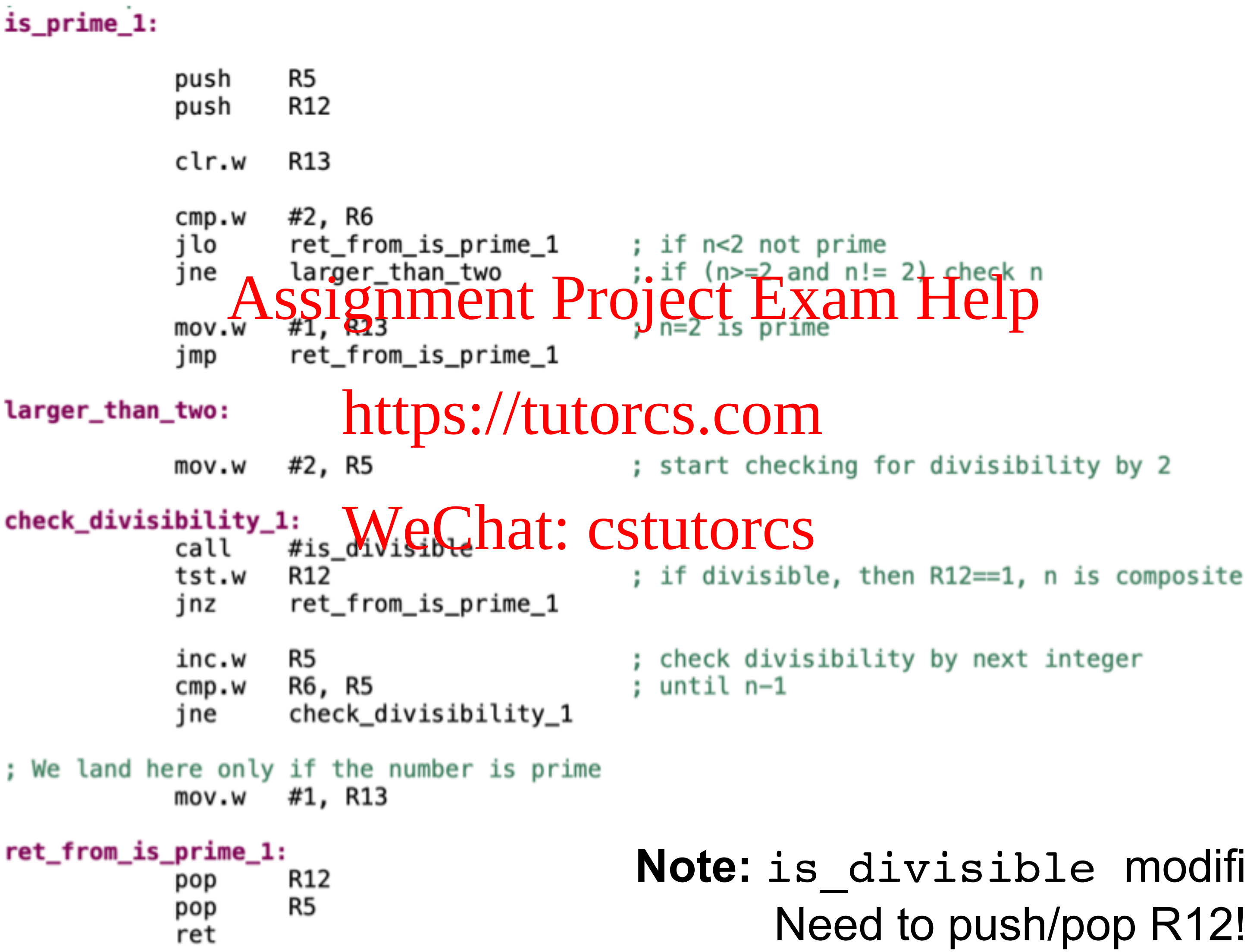


```
;-----------------------------------------------------------------
; Subroutine: is_prime
; Inputs: unsigned word n in R6 -- returned unchanged
;
; Output: binary value in R13 -- R13 = 1 if n is prime
;                                R13 = 0 if n is composite
;
; All other core registers in R4-R15 unchanged
; Subroutine does not access addressed memory locations
;-----------------------------------------------------------------
```

Approach:

- Check if n is divisible by 2, 3, 4, …, n-1
- If it is divisible by any of these numbers, the number is composite
- If not divisible, the number is prime
- Works only if n>3, the cases n = 0, 1, 2 need separate logic

Solution to Quiz 5 v.1



```
is_prime_1:

            push    R5
            push    R12

            clr.w   R13

            cmp.w   #2, R6
            jlo     ret_from_is_prime_1        ; if n<2 not prime
            jne     larger_than_two            ; if (n>=2 and n!= 2) check n

            mov.w   #1, R13                    ; n=2 is prime
            jmp     ret_from_is_prime_1

larger_than_two:

            mov.w   #2, R5                     ; start checking for divisibility by 2

check_divisibility_1:
            call    #is_divisible
            tst.w   R12                        ; if divisible, then R12==1, n is composite
            jnz     ret_from_is_prime_1

            inc.w   R5                         ; check divisibility by next integer
            cmp.w   R6, R5                     ; until n-1
            jne     check_divisibility_1

; We land here only if the number is prime
            mov.w   #1, R13

ret_from_is_prime_1:
            pop     R12
            pop     R5
            ret
```

Note: `is_divisible` modifies R12
Need to push/pop R12!!

Solution to Quiz 5

What about algorithmic complexity? How many calls to `is_divisible`?

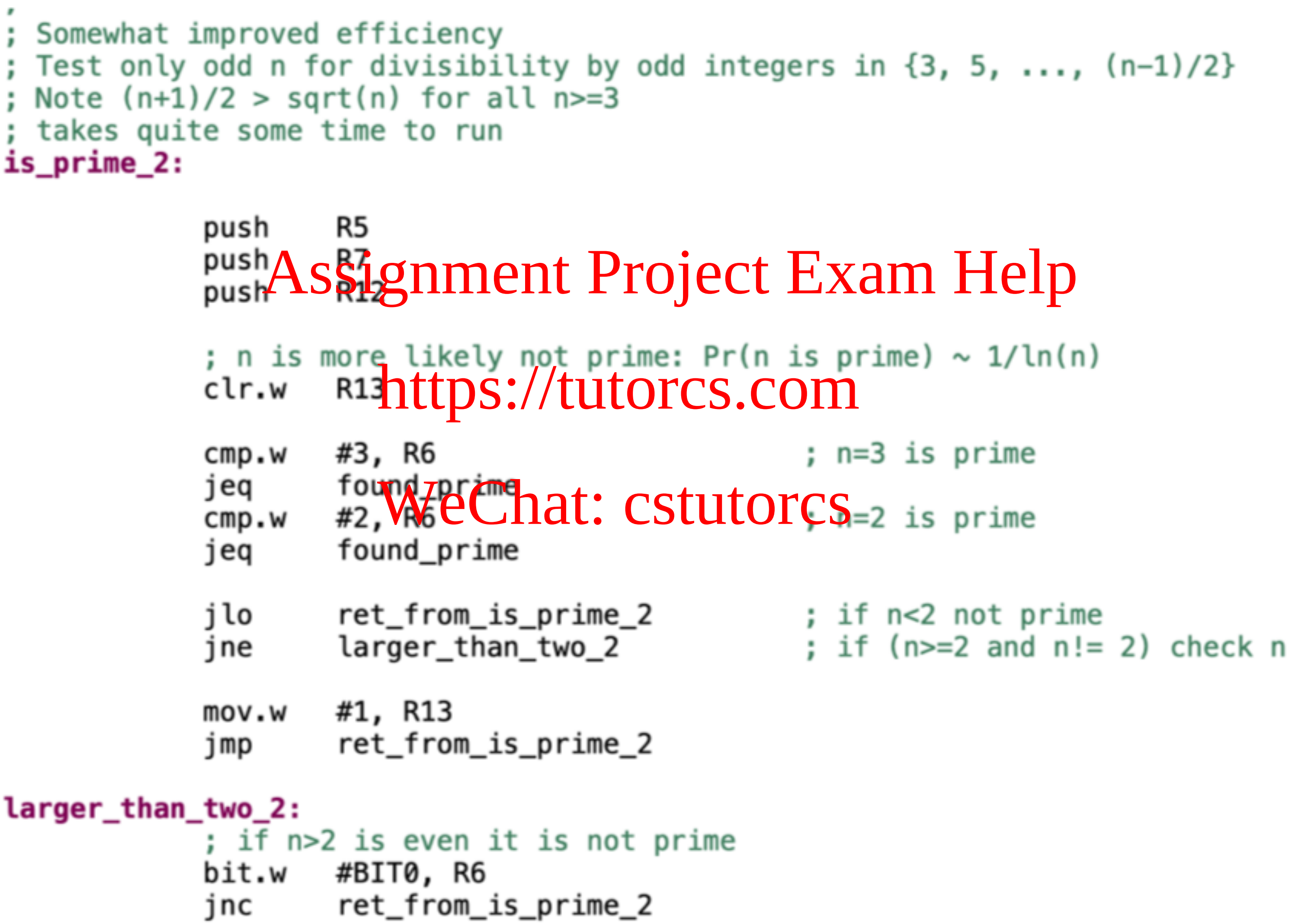
Depends on the nature of the number:

- If n is even, then only one call: n is divisible by 2
- If n is an odd multiple of 3, then two calls: check for divisibility by 2 and 3
- …
- If n is prime, then **n-2** calls to `is_divisible` with 2, 3, …, n-1

How can we speed things up?

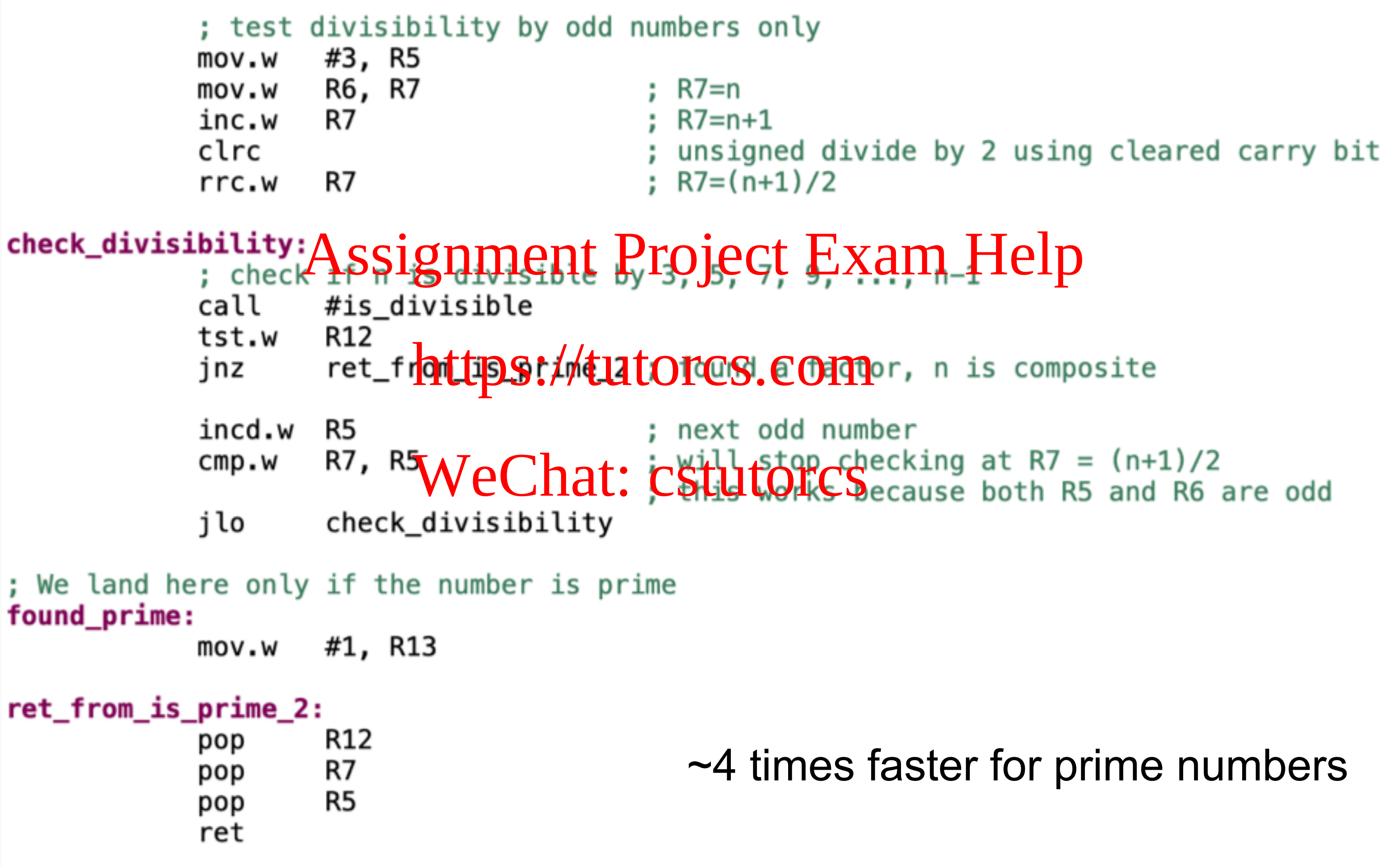
- No need to check n>2 if it is an even number `bit.w #BIT0, R6`
- For odd n, it suffices to check divisibility by odd numbers {3, 5, …, n-2}
- For n>4, no need to check for factors > n/2 check only {3, 5, …, floor(n/2)}
- Actually no need to check for factors > sqrt(n)

Solution to Quiz 5 v.2



```
;
; Somewhat improved efficiency
; Test only odd n for divisibility by odd integers in {3, 5, ..., (n-1)/2}
; Note (n+1)/2 > sqrt(n) for all n>=3
; takes quite some time to run
is_prime_2:

            push    R5
            push    R7
            push    R12

            ; n is more likely not prime: Pr(n is prime) ~ 1/ln(n)
            clr.w   R13

            cmp.w   #3, R6                      ; n=3 is prime
            jeq     found_prime
            cmp.w   #2, R6                      ; n=2 is prime
            jeq     found_prime

            jlo     ret_from_is_prime_2         ; if n<2 not prime
            jne     larger_than_two_2           ; if (n>=2 and n!= 2) check n

            mov.w   #1, R13
            jmp     ret_from_is_prime_2

larger_than_two_2:
            ; if n>2 is even it is not prime
            bit.w   #BIT0, R6
            jnc     ret_from_is_prime_2
```

Solution to Quiz 5 v.2



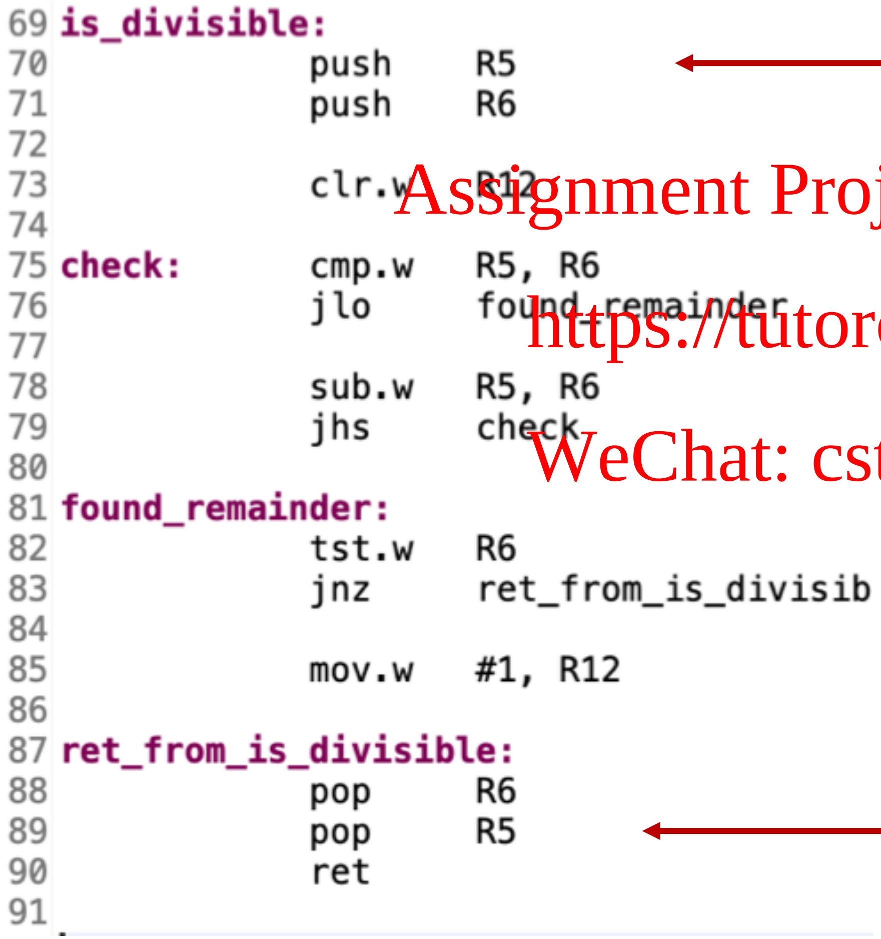
```
            ; test divisibility by odd numbers only
            mov.w   #3, R5
            mov.w   R6, R7                      ; R7=n
            inc.w   R7                          ; R7=n+1
            clrc                                ; unsigned divide by 2 using cleared carry bit
            rrc.w   R7                          ; R7=(n+1)/2

check_divisibility:
            ; check if n is divisible by 3, 5, 7, 9, ..., n-1
            call    #is_divisible
            tst.w   R12
            jnz     ret_from_is_prime_2         ; found a factor, n is composite

            incd.w  R5                          ; next odd number
            cmp.w   R7, R5                      ; will stop checking at R7 = (n+1)/2
                                                ; this works because both R5 and R6 are odd
            jlo     check_divisibility

; We land here only if the number is prime
found_prime:
            mov.w   #1, R13

ret_from_is_prime_2:
            pop     R12
            pop     R7
            pop     R5
            ret
```

~4 times faster for prime numbers

Learning From Quiz 4

How can we *improve* following code?

Logic is correct



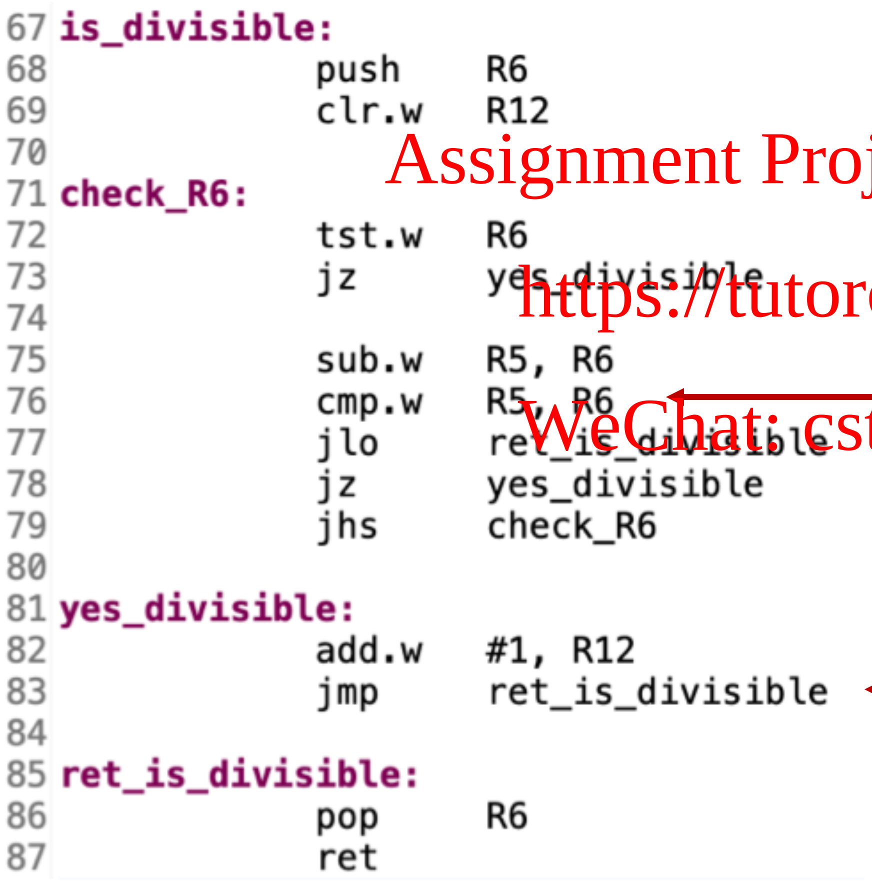
```
is_divisible:
            push    R5
            push    R6

            clr.w   R12

check:      cmp.w   R5, R6
            jlo     found_remainder

            sub.w   R5, R6
            jhs     check

found_remainder:
            tst.w   R6
            jnz     ret_from_is_divisib

            mov.w   #1, R12

ret_from_is_divisible:
            pop     R6
            pop     R5
            ret
```

No need to push/pop R5 since it is not modified

Learning From Quiz 4

How can we *improve* following code?

Logic is correct



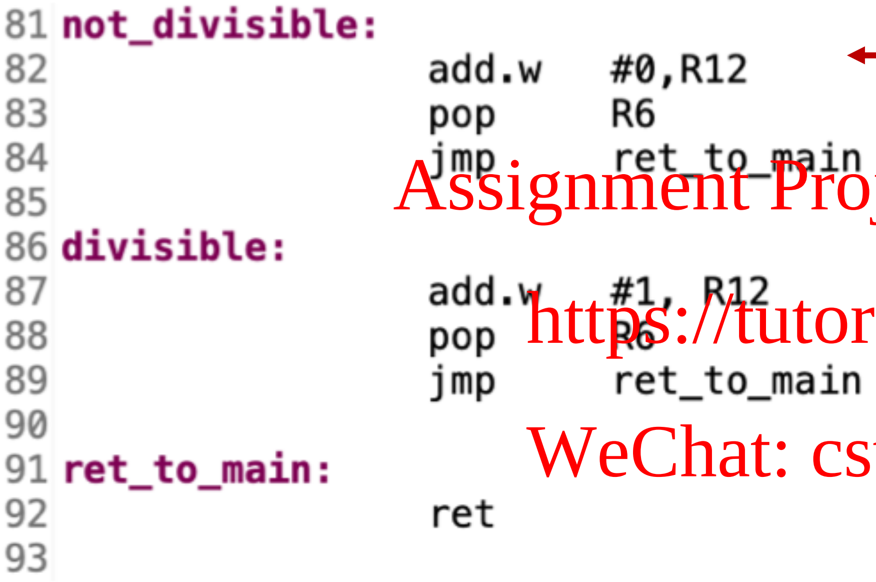
```
is_divisible:
            push    R6
            clr.w   R12

check_R6:
            tst.w   R6
            jz      yes_divisible

            sub.w   R5, R6
            cmp.w   R5, R6
            jlo     ret_is_divisible
            jz      yes_divisible
            jhs     check_R6

yes_divisible:
            add.w   #1, R12
            jmp     ret_is_divisible

ret_is_divisible:
            pop     R6
            ret
```

No need: `sub.w` already sets the status bits!!

No need to jump
ret_is_divisible is the next line already!!!

Learning From Quiz 4

How can we *improve* following code?

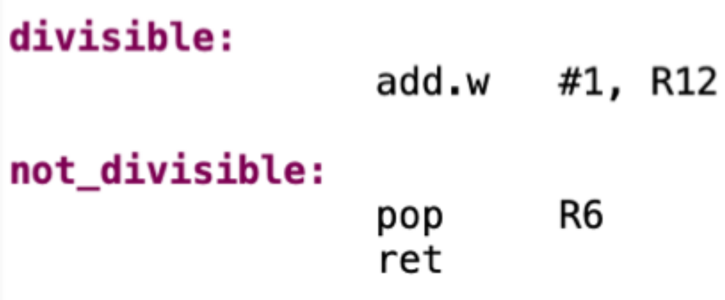


```
81 not_divisible:
82                  add.w   #0,R12
83                  pop     R6
84                  jmp     ret_to_main
85
86 divisible:
87                  add.w   #1, R12
88                  pop     R6
89                  jmp     ret_to_main
90
91 ret_to_main:
92                  ret
93
```

No need: x + 0 = x

No need to jump to the next line

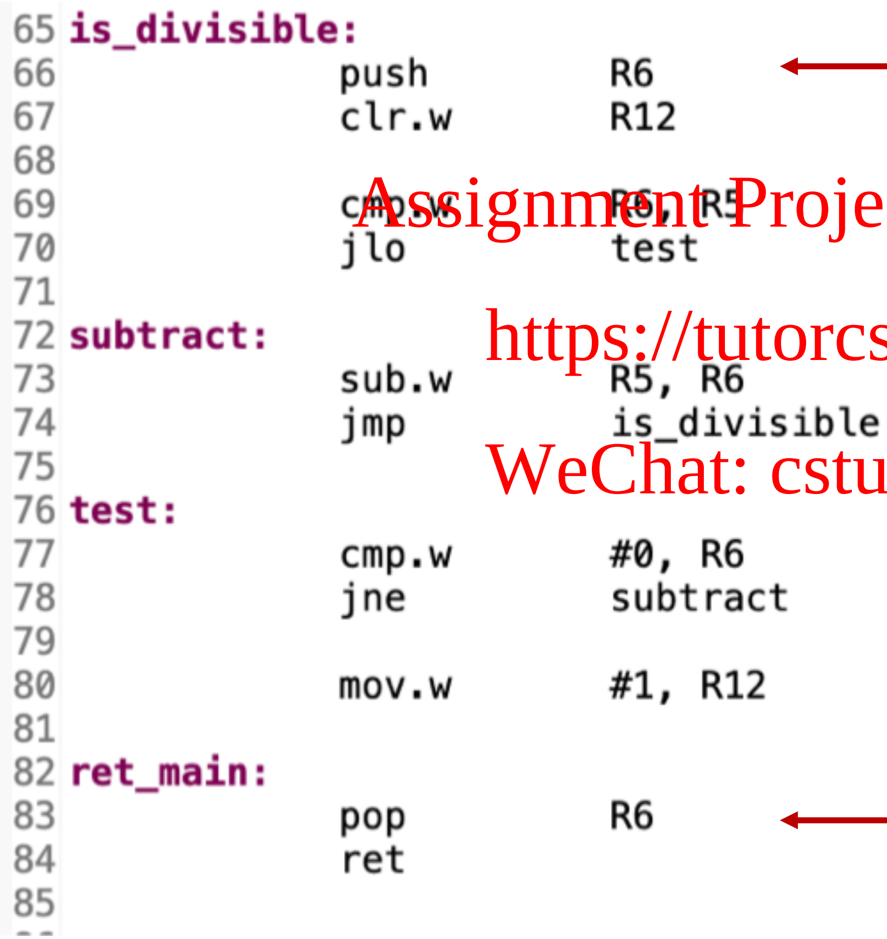
No need to repeat lines 83-84



```
divisible:
                add.w   #1, R12

not_divisible:
                pop     R6
                ret
```

Learning From Quiz 4

What is the issue with this code?



```
is_divisible:
                push        R6
                clr.w       R12

                cmp.w       R6, R5
                jlo         test

subtract:
                sub.w       R5, R6
                jmp         is_divisible

test:
                cmp.w       #0, R6
                jne         subtract

                mov.w       #1, R12

ret_main:
                pop         R6
                ret

```

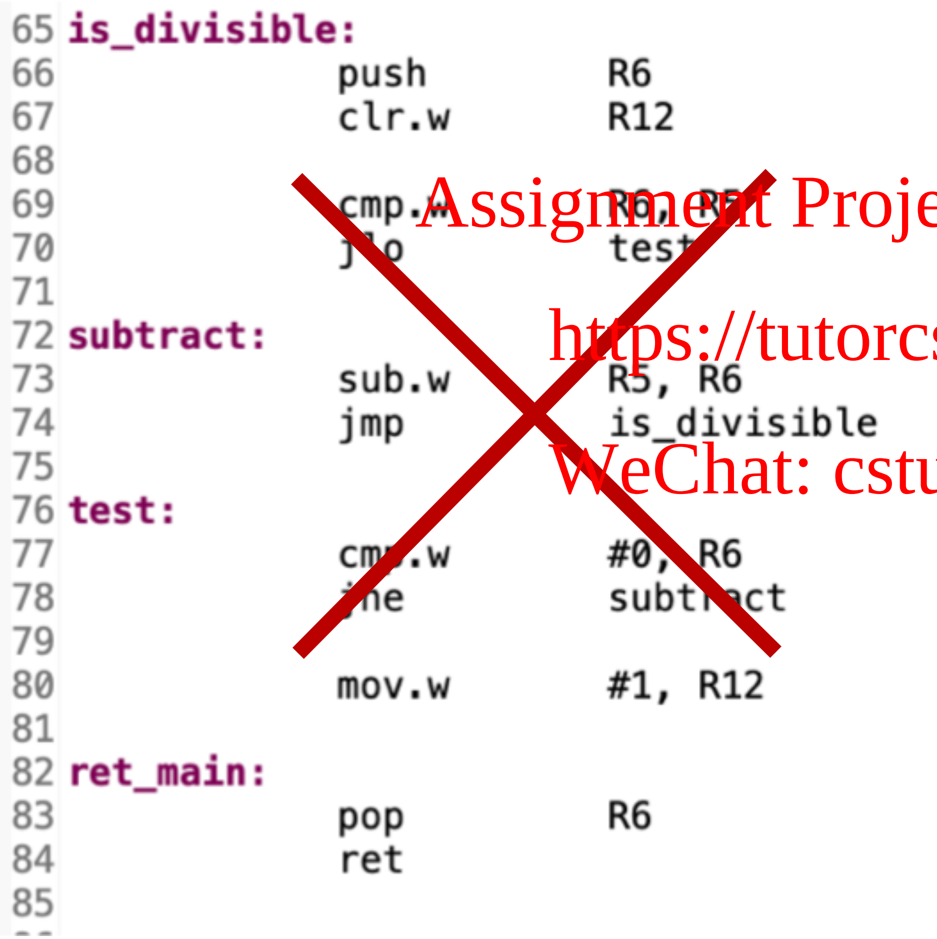
Pushes to stack with each iteration!!!

Pops only once

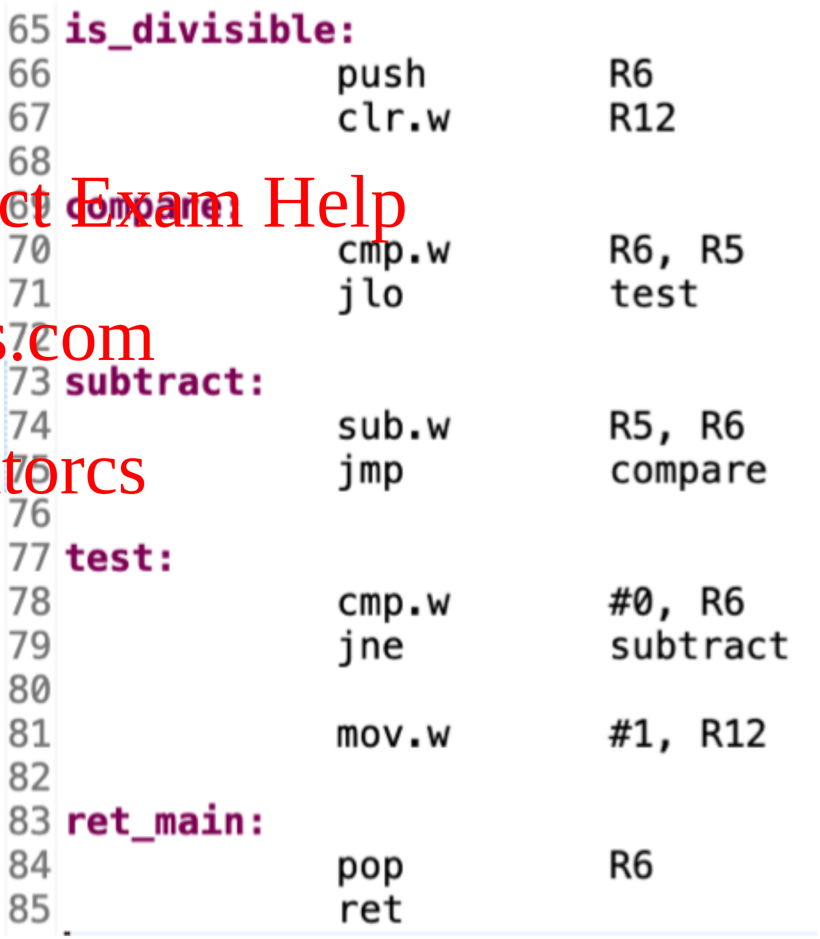
`(R6) -> PC -> crash`

This is a Mistake

How do we fix it?



```
is_divisible:
                    push        R6
                    clr.w       R12

                    cmp.w       R6, R5
                    jlo         test

subtract:
                    sub.w       R5, R6
                    jmp         is_divisible

test:
                    cmp.w       #0, R6
                    jne         subtract

                    mov.w       #1, R12

ret_main:
                    pop         R6
                    ret

```



```
is_divisible:
                    push        R6
                    clr.w       R12

compare:
                    cmp.w       R6, R5
                    jlo         test

subtract:
                    sub.w       R5, R6
                    jmp         compare

test:
                    cmp.w       #0, R6
                    jne         subtract

                    mov.w       #1, R12

ret_main:
                    pop         R6
                    ret
```

How MCUs are used in the Real-World

Not the way we have used them so far: we have only used the CPU and memory (RAM/FRAM) of our MCU to do basic data manipulations

When treated like this, the MCU is a very limited computer:

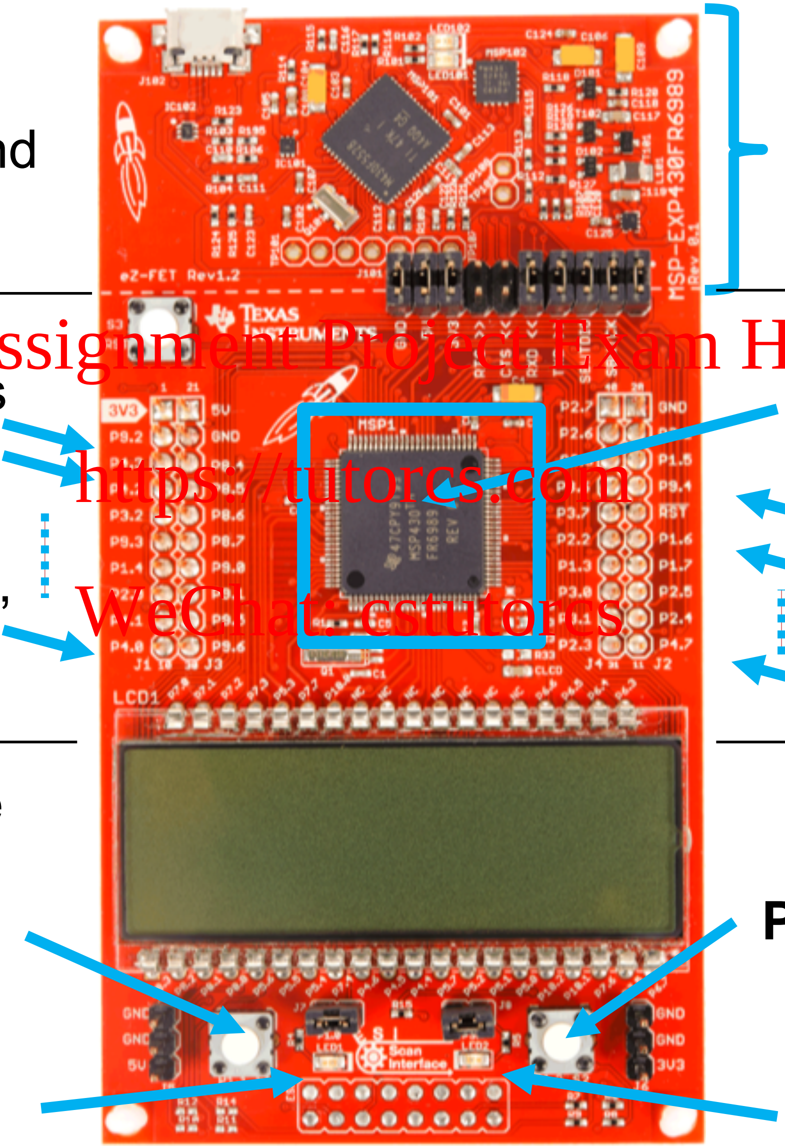
- No real input or interaction with the user/environment
- We defined (hardcoded) data in RAM/FRAM as input to some logic
- Output is limited too: we peek into registers using CCS to view the input

An MCU is intended to do much more

- Interact with the user / environment / other MCUs
- **Input** from buttons, sensors, other MCUs
- **Output** via displays, motors (motor drivers), actuators, control circuitry …

All input / output to the MCU is through **Input/Output Pins**

The MSP430FR6989 Launchpad



This part enables interface to a PC and enables debugging

eZ-FET emulator

Headers with access to selected pins connect I/O devices e.g., sensors, motors, logic analyzer…

MSP430FR6989IPZ

100 Pins

More headers

Only I/O we will use

Push Button S1

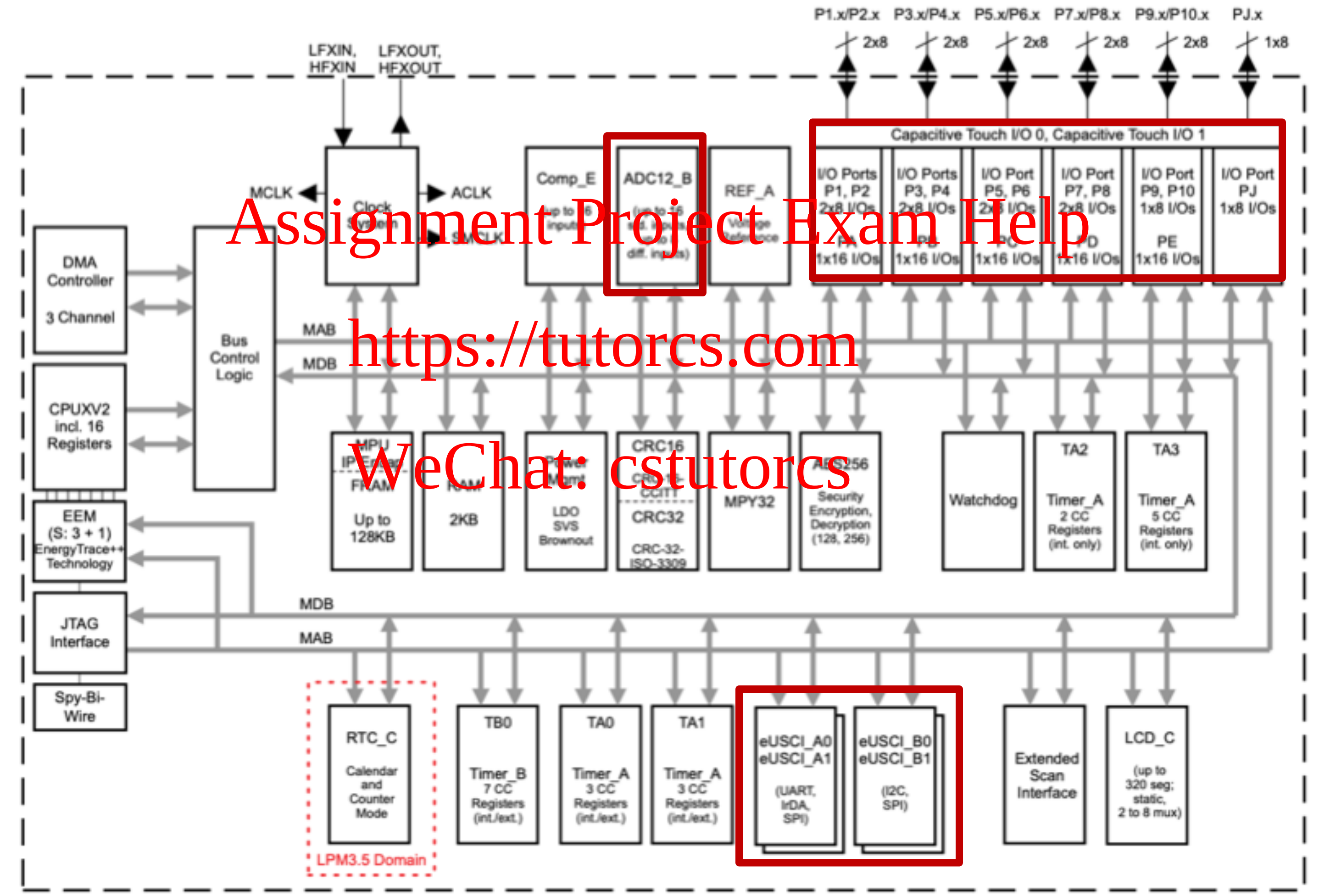
Push Button S2

Red LED

Green LED

MSP430FR6989 I/O Options

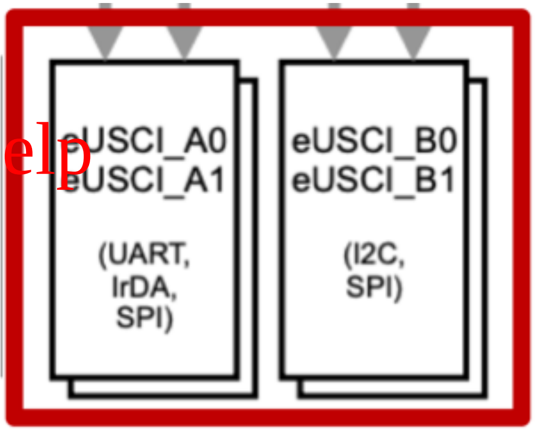
Ports for GPIO



I/O Through Standard Protocols

The **enhanced Universal Serial Communication Interfaces (eUSCI_A** and **eUSCI_B)** support several standard protocols for I/O

- **Serial Peripheral Interface (SPI)**
- **Inter Integrated Circuit (I^2C, I2C, IIC)**
- **Universal Asynchronous Receiver Transmitter (UART)**



How to use?

- Dedicated pins for I/O
- Registers for configuration

All the details in **slau367p.pdf**
Posted to Carmen under Resources

- Devices that use these protocols
- e.g., SPI sensors, I^2C sensors, I^2C motor drivers etc.

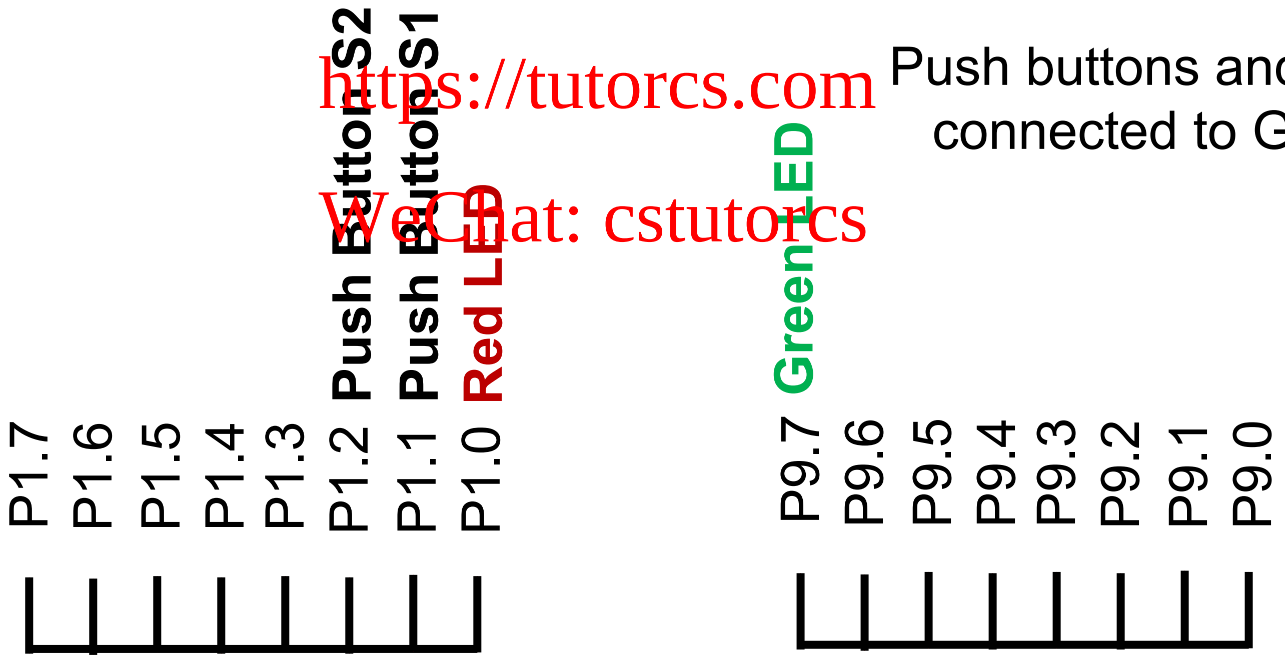
GPIO Ports P1 – P10

Our MCU has 10 **General Purpose Input Output (GPIO) Ports P1 – P10**

(or PA – PJ)

TI refers to these as **Digital I/O ports**

- Each port has **8 pins**
- Pins are labeled as **Px.y** – **x** is the port number, **y** is the pin number



Push buttons and LEDs are connected to GPIO pins

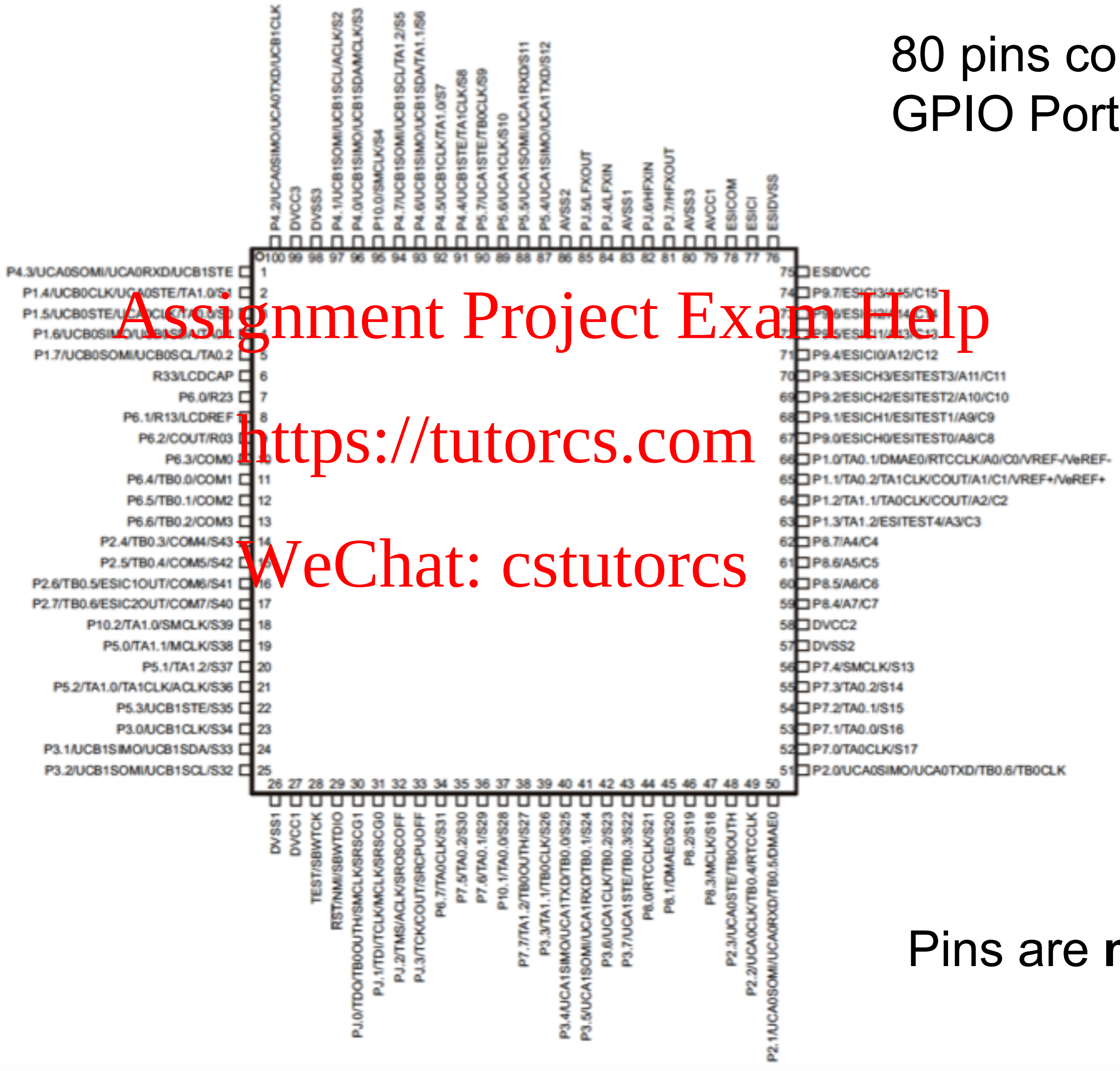
Pins of Port 1

Pins of Port 9

MSP430FR6989IPZ Pinout

slas789d.pdf

80 pins connected to GPIO Ports

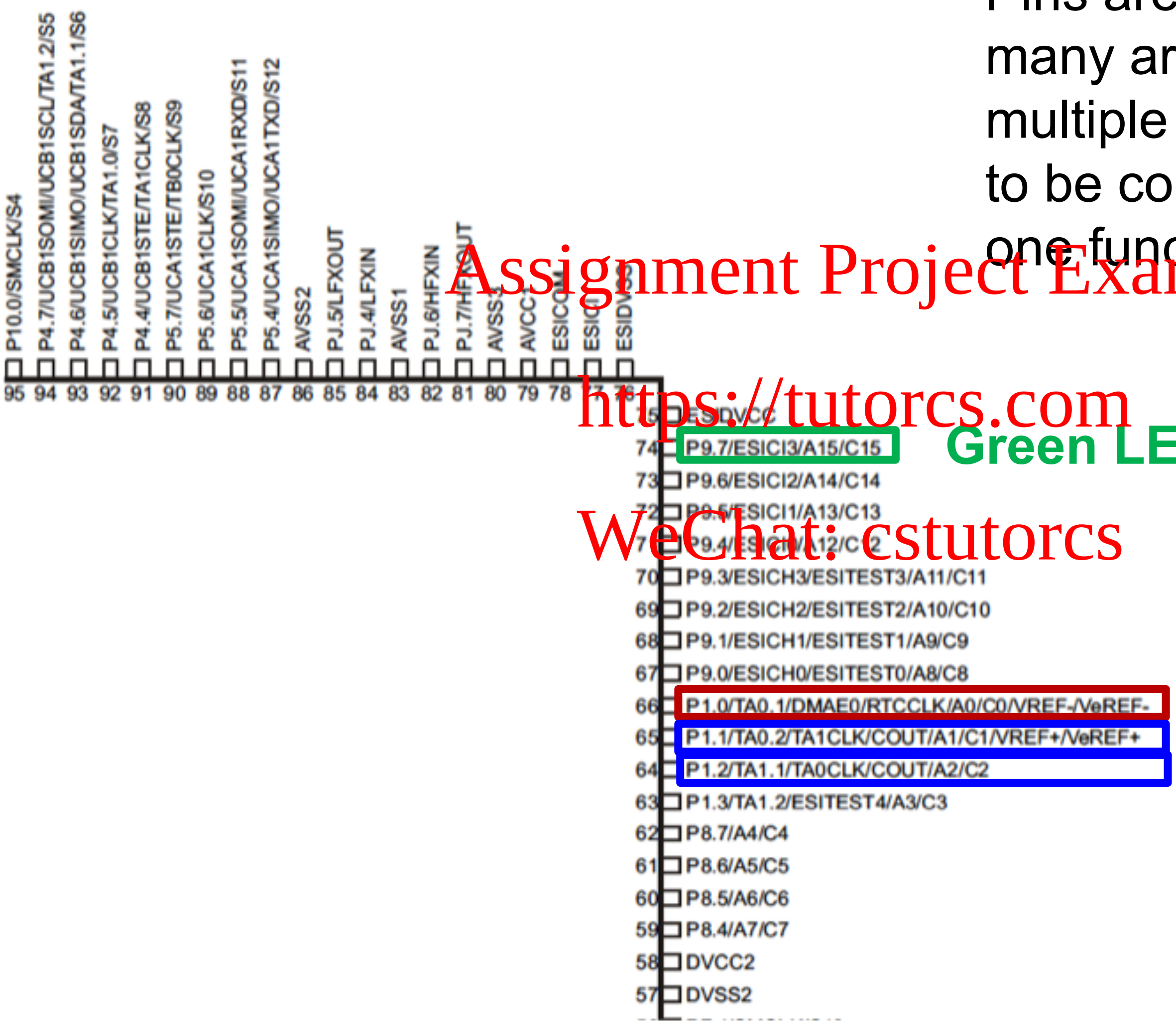


1 P4.3/UCA0SOMI/UCA0RXD/UCB1STE
2 P1.4/UCB0CLK/UCA0STE/TA1.0/S1
3 P1.5/UCB0STE/UCA0CLK/TA0.0/S0
4 P1.6/UCB0SIMO/UCB0SDA/TA0.1
5 P1.7/UCB0SOMI/UCB0SCL/TA0.2
6 R33/LCDCAP
7 P6.0/R23
8 P6.1/R13/LCDREF
9 P6.2/COUT/R03
10 P6.3/COM0
11 P6.4/TB0.0/COM1
12 P6.5/TB0.1/COM2
13 P6.6/TB0.2/COM3
14 P2.4/TB0.3/COM4/S43
15 P2.5/TB0.4/COM5/S42
16 P2.6/TB0.5/ESIC1OUT/COM6/S41
17 P2.7/TB0.6/ESIC2OUT/COM7/S40
18 P10.2/TA1.0/SMCLK/S39
19 P5.0/TA1.1/MCLK/S38
20 P5.1/TA1.2/S37
21 P5.2/TA1.0/TA1CLK/ACLK/S36
22 P5.3/UCB1STE/S35
23 P3.0/UCB1CLK/S34
24 P3.1/UCB1SIMO/UCB1SDA/S33
25 P3.2/UCB1SOMI/UCB1SCL/S32
26 DVSS1
27 DVCC1
28 TEST/SBWTCK
29 RST/NMI/SBWTDIO
30 PJ.0/TDO/TB0OUTH/SMCLK/SRSCG1
31 PJ.1/TDI/TCLK/MCLK/SRSCG0
32 PJ.2/TMS/ACLK/SROSCOFF
33 PJ.3/TCK/COUT/SRCPUOFF
34 P6.7/TA0CLK/S31
35 P7.5/TA0.2/S30
36 P7.6/TA0.1/S29
37 P10.1/TA0.0/S28
38 P7.7/TA1.2/TB0OUTH/S27
39 P3.3/TA1.1/TB0CLK/S26
40 P3.4/UCA1SIMO/UCA1TXD/TB0.0/S25
41 P3.5/UCA1SOMI/UCA1RXD/TB0.1/S24
42 P3.6/UCA1CLK/TB0.2/S23
43 P3.7/UCA1STE/TB0.3/S22
44 P8.0/RTCCLK/S21
45 P8.1/DMAE0/S20
46 P8.2/S19
47 P8.3/MCLK/S18
48 P2.3/UCA0STE/TB0OUTH
49 P2.2/UCA0CLK/TB0.4/RTCCLK
50 P2.1/UCA0SOMI/UCA0RXD/TB0.5/DMAE0
51 P2.0/UCA0SIMO/UCA0TXD/TB0.6/TB0CLK
52 P7.0/TA0CLK/S17
53 P7.1/TA0.0/S16
54 P7.2/TA0.1/S15
55 P7.3/TA0.2/S14
56 P7.4/SMCLK/S13
57 DVSS2
58 DVCC2
59 P8.4/A7/C7
60 P8.5/A6/C6
61 P8.6/A5/C5
62 P8.7/A4/C4
63 P1.3/TA1.2/ESITEST4/A3/C3
64 P1.2/TA1.1/TA0CLK/COUT/A2/C2
65 P1.1/TA0.2/TA1CLK/COUT/A1/C1/VREF+/VeREF+
66 P1.0/TA0.1/DMAE0/RTCCLK/A0/C0/VREF-/VeREF-
67 P9.0/ESICH0/ESITEST0/A8/C8
68 P9.1/ESICH1/ESITEST1/A9/C9
69 P9.2/ESICH2/ESITEST2/A10/C10
70 P9.3/ESICH3/ESITEST3/A11/C11
71 P9.4/ESICI0/A12/C12
72 P9.5/ESICI1/A13/C13
73 P9.6/ESICI2/A14/C14
74 P9.7/ESICI3/A15/C15
75 ESIDVCC
76 ESIDVSS
77 ESICI
78 ESICOM
79 AVCC1
80 AVSS3
81 PJ.7/HFXOUT
82 PJ.6/HFXIN
83 AVSS1
84 PJ.4/LFXIN
85 PJ.5/LFXOUT
86 AVSS2
87 P5.4/UCA1SIMO/UCA1TXD/S12
88 P5.5/UCA1SOMI/UCA1RXD/S11
89 P5.6/UCA1CLK/S10
90 P5.7/UCA1STE/TB0CLK/S9
91 P4.4/UCB1STE/TA1CLK/S8
92 P4.5/UCB1CLK/TA1.0/S7
93 P4.6/UCB1SIMO/UCB1SDA/TA1.1/S6
94 P4.7/UCB1SOMI/UCB1SCL/TA1.2/S5
95 P10.0/SMCLK/S4
96 P4.0/UCB1SIMO/UCB1SDA/MCLK/S3
97 P4.1/UCB1SOMI/UCB1SCL/ACLK/S2
98 DVSS3
99 DVCC3
100 P4.2/UCA0SIMO/UCA0TXD/UCB1CLK

Pins are **multiplexed**

MSP430FR6989IPZ Pinout

Pins are **multiplexed** many are connected to multiple peripherals and need to be configured by selecting one functionality



Green LED

Red LED

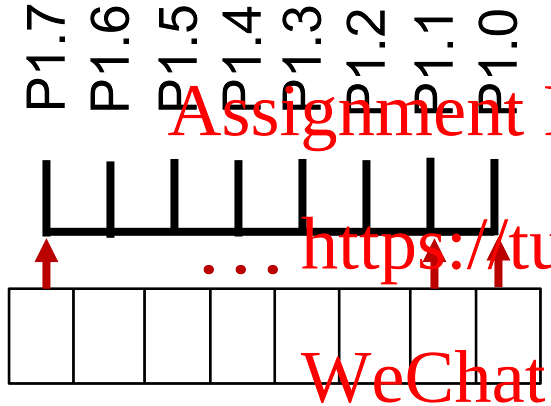
Push Buttons S1 & S2

Registers Controlling GPIO Ports

Each port is configured and controlled by a set of **8-bit registers**

Px.y is controlled by **bit y** in the register corresponding to **port x**

e.g.



Pins

Bits

8-bit Registers

P1SEL0 / P1SEL1
P1SELC
P1DIR
P1OUT P1IN
P1REN
P1IFG
P1IE
P1IES

e.g., output HIGH on p1.0

⇒ set BIT0 in P1OUT

How?

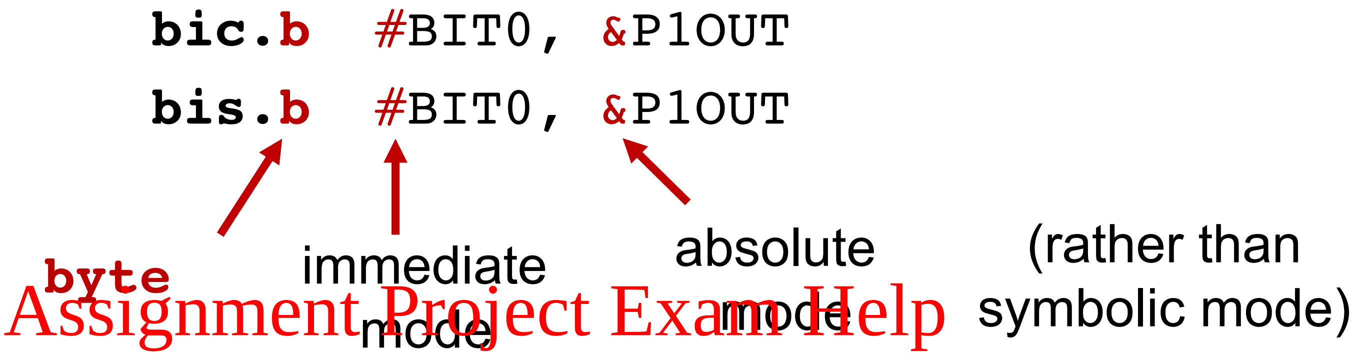
```
bis.b   #BIT0, &P1OUT
```

How not?

~~mov.b #BIT0, &P1OUT~~

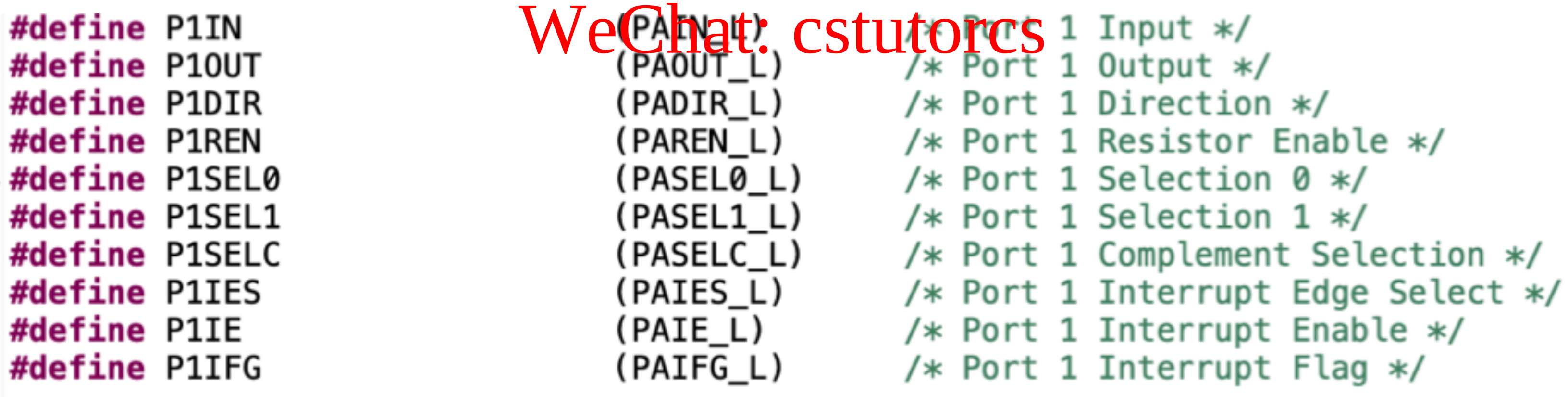
ALWAYS use bit operations
NEVER use move

Addressing GPIO Port Registers



```
bic.b   #BIT0,  &P1OUT
bis.b   #BIT0,  &P1OUT
```

All addresses are defined in the header file msp430fr69891.h

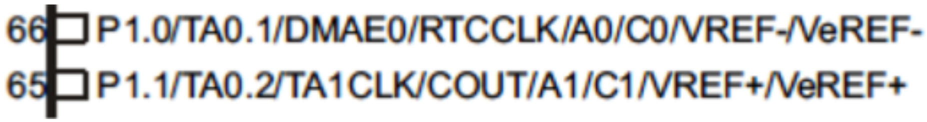


```
#define P1IN                     (PAIN_L)      /* Port 1 Input */
#define P1OUT                    (PAOUT_L)     /* Port 1 Output */
#define P1DIR                    (PADIR_L)     /* Port 1 Direction */
#define P1REN                    (PAREN_L)     /* Port 1 Resistor Enable */
#define P1SEL0                   (PASEL0_L)    /* Port 1 Selection 0 */
#define P1SEL1                   (PASEL1_L)    /* Port 1 Selection 1 */
#define P1SELC                   (PASELC_L)    /* Port 1 Complement Selection */
#define P1IES                    (PAIES_L)     /* Port 1 Interrupt Edge Select */
#define P1IE                     (PAIE_L)      /* Port 1 Interrupt Enable */
#define P1IFG                    (PAIFG_L)     /* Port 1 Interrupt Flag */
```

Registers are replicated for all 10 ports: P2IN, …, P3IN, …, P10IN,…

Configuring Px.y: PxSEL0/ PxSEL1

Function Select Registers: PxSEL0, PxSEL1


66 P1.0/TA0.1/DMAE0/RTCCLK/A0/C0/VREF-/VeREF-

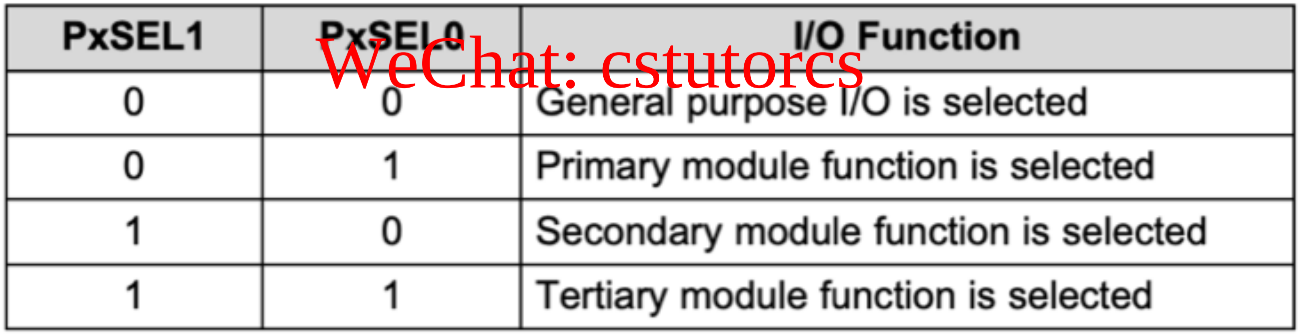
65 P1.1/TA0.2/TA1CLK/COUT/A1/C1/VREF+/VeREF+

Pins are multiplexed

PxSEL0 and PxSEL1 determine the pin function

(PxSELC is a helper register to complement between 00 and 11)

Table 12-2. I/O Function Selection



PxSEL1	PxSEL0	I/O Function
0	0	General purpose I/O is selected
0	1	Primary module function is selected
1	0	Secondary module function is selected
1	1	Tertiary module function is selected

Default values are **PxSEL0.y = 0** and **PxSEL1.y = 0**

⇒ the default function for each pin **Px.y** is **GPIO / Digital I/O**

Configuring Px.y: PxDIR

Direction Register: PxDIR

Selects the **direction** of the corresponding I/O pin: i.e., input or output

PxDIR.y = 0: Pin **Px.y** is switched to **input** direction (Default)

PxDIR.y = 1: Pin **Px.y** is switched to **output** direction

Shorthand notation:

PxDIR.y refers to bit $\mathbf{y} \in \{0,1, \ldots, 7\}$ of register controlling port $\mathbf{x} \in \{1, \ldots, 10\}$

Px.y refers to pin $\mathbf{y} \in \{0,1, \ldots, 7\}$ of port $\mathbf{x} \in \{1, \ldots, 10\}$

Configuring Px.y: PxIN

Input Register: PxIN

Bit **PxIN.y** reflects the value of the intput signal at pin **Px.y**

PxIN.y = 0: Input at pin **Px.y** is LOW

PxIN.y = 1: Input at pin **Px.y** is HIGH

Note: PxIN is a read-only register

You cannot write to it, attempting to write only results in increased current consumption while the write attempt is active

Configuring Px.y: PxOUT – Role 1

Output Register: PxOUT

Bit **PxOUT.y** is the value of the output signal at pin **Px.y**

when the pin is configured as I/O function, **output** direction

PxOUT.y = 0: Output at pin **Px.y** is LOW

PxOUT.y = 1: Output at pin **Px.y** is HIGH

How to write to output?

The red LED is connected to **P1.0**

P1DIR.0 =1 selects the pin as **output**

First set the desired output value, then change the direction Otherwise, the initial output may be random

bis.b ~~#BIT0,~~ &P1DIR
bis.b ~~#BIT0,~~ &P1OUT

Option 1

```
bis.b    #BIT0, &P1OUT
bis.b    #BIT0, &P1DIR
```

Option 2