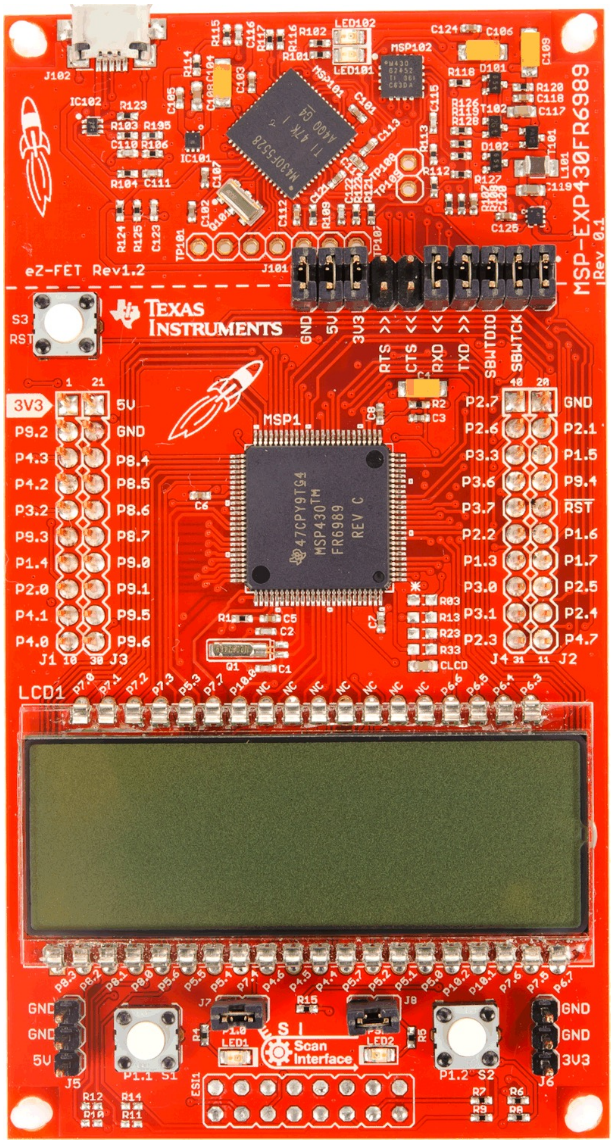
Red LED



Green LED

Lecture 19

GPIO

General Purpose Input Output

But First – Joke of the Day

You know the song:
99 bottles of ~~beer~~ on the wall

barley juice

The never-ending song of programming

99 little bugs in the code,

99 little bugs

Take one down, patch it around

117 little bugs in the code

More on the Project

Task: Sort the elements in a given array

```
;-------------------------------------------------------------------------------
; Subroutine: sort
; Inputs: R8 pointer to word array -- returned unchanged
;           R8 contains the 16-bit starting address of an array
;               The array consists of n 16-bit signed integers
;
;         R10 = n, the number of elements in the array -- returned unchanged
;
; Output: The subroutine sorts the elements in a given array from smallest to largest
;           You will implement selection sort
;
; All core registers in R4-R15 unchanged
; Subroutine does not access any global variables or defined constants
;-------------------------------------------------------------------------------
```

You will write three subroutines

- `sort` calls `select` and `swap`
- If `select` or `swap` does not work correctly, `sort` will not correctly either!
- **Test and test and test !!!**
- Develop and test your code with smaller arrays to make sure they work correctly

More on the Project

Task: Sort the elements in a given array

```
;-------------------------------------------------------------------------------
; Subroutine: sort
; Inputs: R8 pointer to word array -- returned unchanged
;              R8 contains the 16-bit starting address of an array
;                  The array consists of n 16-bit signed integers
;
;            R10 = n, the number of elements in the array -- returned unchanged
;
; Output: The subroutine sorts the elements in a given array from smallest to largest
;              You will implement selection sort
;
; All core registers in R4-R15 unchanged
; Subroutine does not access any global variables or defined constants
;-------------------------------------------------------------------------------
```

What exactly are you given?

- Starting address of the array in register `R8`
- Number of elements in this array in register `R10`

How are you going to access the elements in this array?

More on Addressing Modes

So far, we have always used **indexed mode** for addressing arrays:

```
array1: .word  0x0100,  0x0200,  0x0300

        mov.w  array1(R4), R5        ; array is the starting address
```

But when you have the starting address of an array in a register indexed mode does not work

```
;-----------------------------------------------------------------------
; Subroutine: sort
; Inputs: R8 pointer to word array -- returned unchanged
;             R8 contains the 16-bit starting address of an array
;                 The array consists of n 16-bit signed integers
;
```

no ~~R8(R4)~~

What to do?

Indirect Register Mode

Indirect Register Mode of addressing works with *pointers*

The address of the source is contained in a core register

Syntax

```
mov.w  @R8, R5
```

Copy word from address in `R8` to the destination

e.g.:

```
        .data
array:  .space 512
                                        array = 0x1C00

        mov.w   #array, R8    ; R8 has the address of the array
        mov.w   @R8, R5       ; R5 = first element in array
        incd.w  R8            ; R8 points to next element
```

Indirect Register Mode of addressing works **only for the source!**

Indirect Register Mode

How to write to a destination whose address is given in a core register?

You want to write to memory location whose address is in `R9`

This will fail

```
mov.w   R5, @R9
```

Indirect register mode works only for the source!

What will work?

```
mov.w   R5, 0(R9)
```

You *dereference* using indexed mode

How do you check for the end condition? i.e., When does the array end?

With indexed mode we were checking indices 0, 2, … , LENGTH-2 etc.

Here it is easiest to check the number of elements

- You have N elements at the beginning
- After processing an element, decrement the element count
- Repeat until the element count hits zero

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:

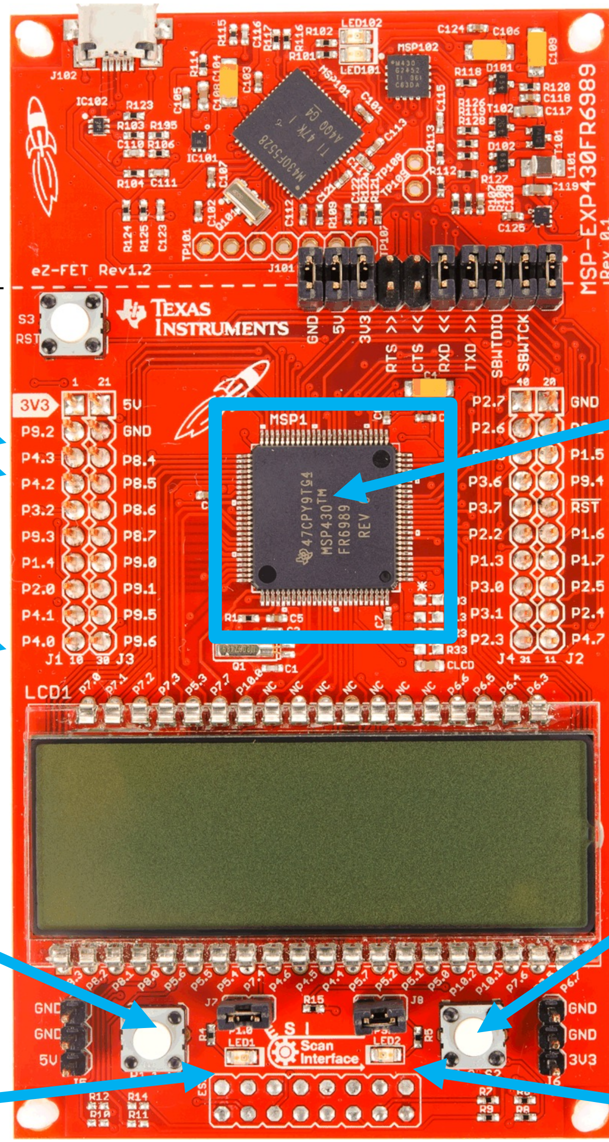
- No real input or interaction with the user/environment
- We defined (hardcoded) data in RAM/FRAM as input to some logic
- Output is limited too: we peek into registers using CCS to view the input

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

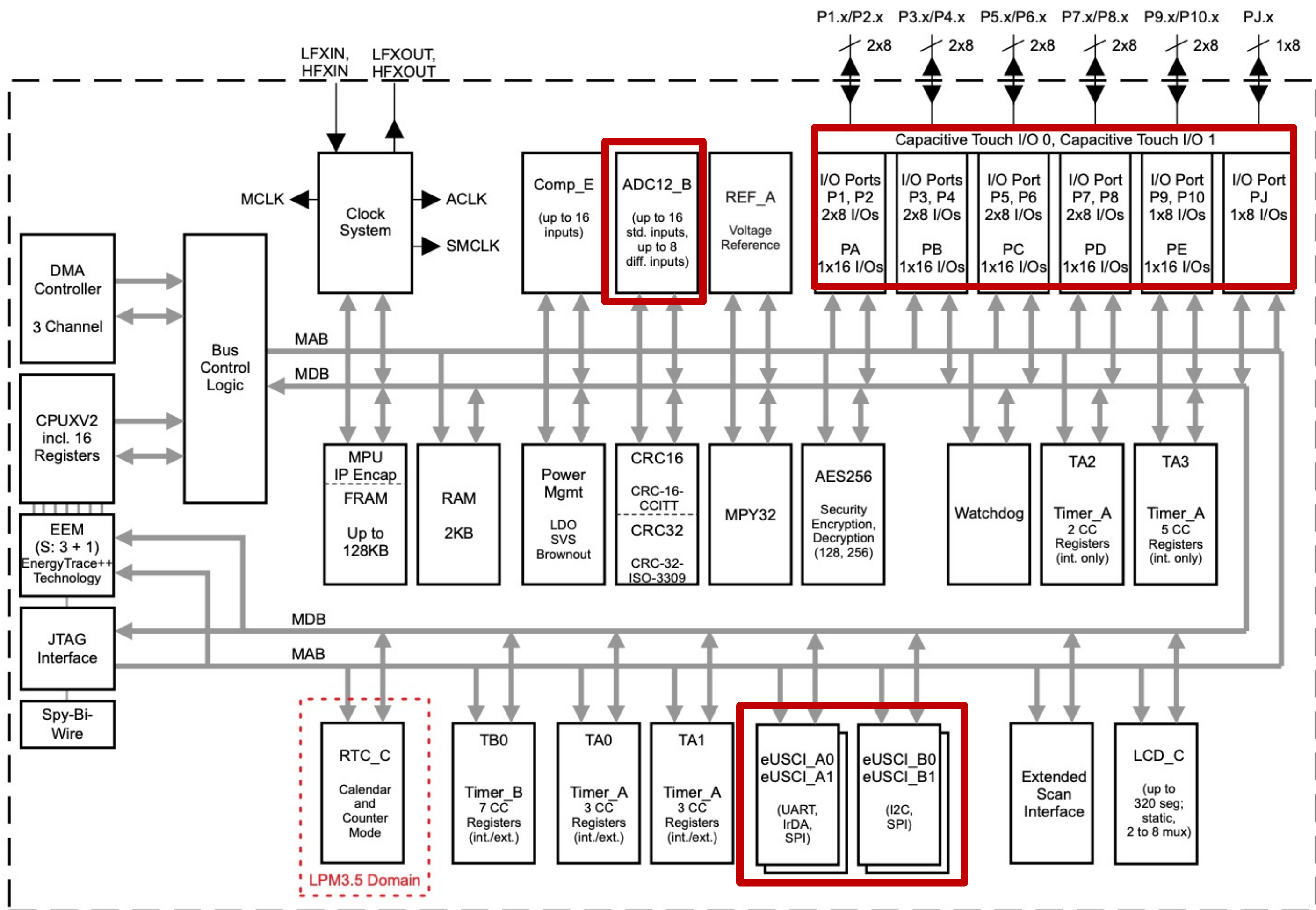
Push Button S2

Red LED

Green LED

MSP430FR6989 I/O Options

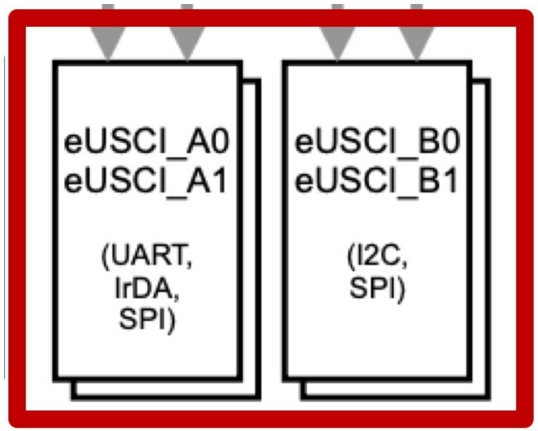
Ports for GPIO



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

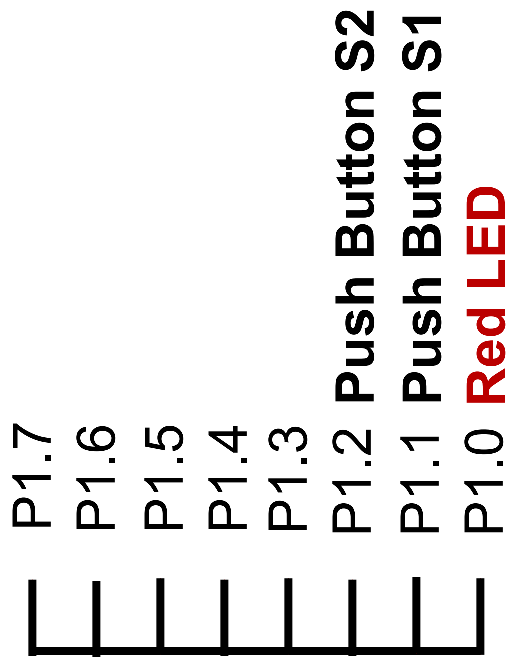
All the details in **slau367p.pdf**
Posted to Carmen under Resources

GPIO Ports P1 – P10

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

TI refers to these as **Digital I/O ports** (or PA – PJ)

- Each port has **8 pins**, each pin can connect to an I/O device
- Pins are labeled as **Px.y** – **x** is the port number, **y** is the pin number



Pins of Port 1

Push buttons and LEDs are connected to GPIO pins

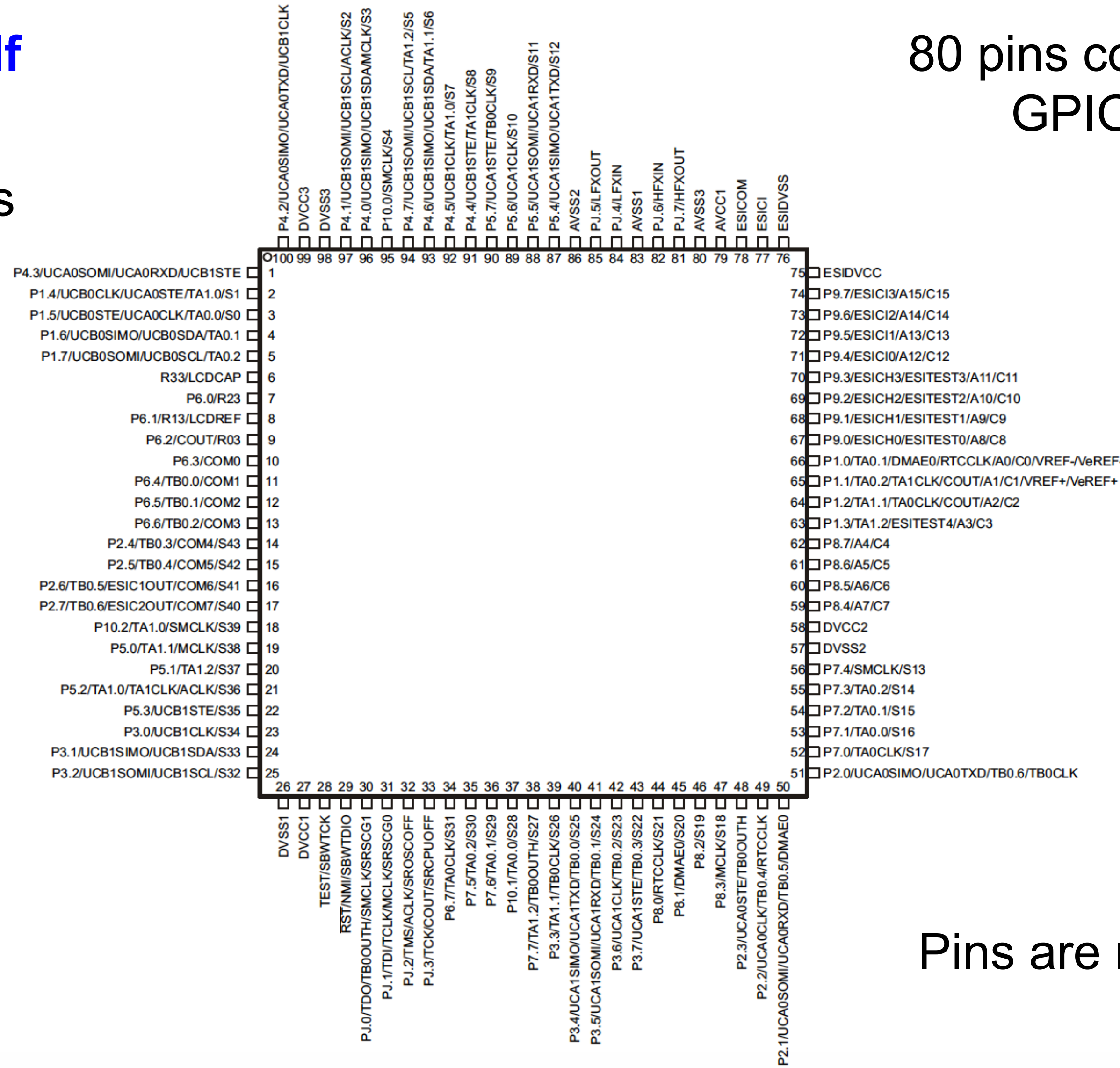
Pins of Port 9

MSP430FR6989IPZ Pinout

slas789d.pdf

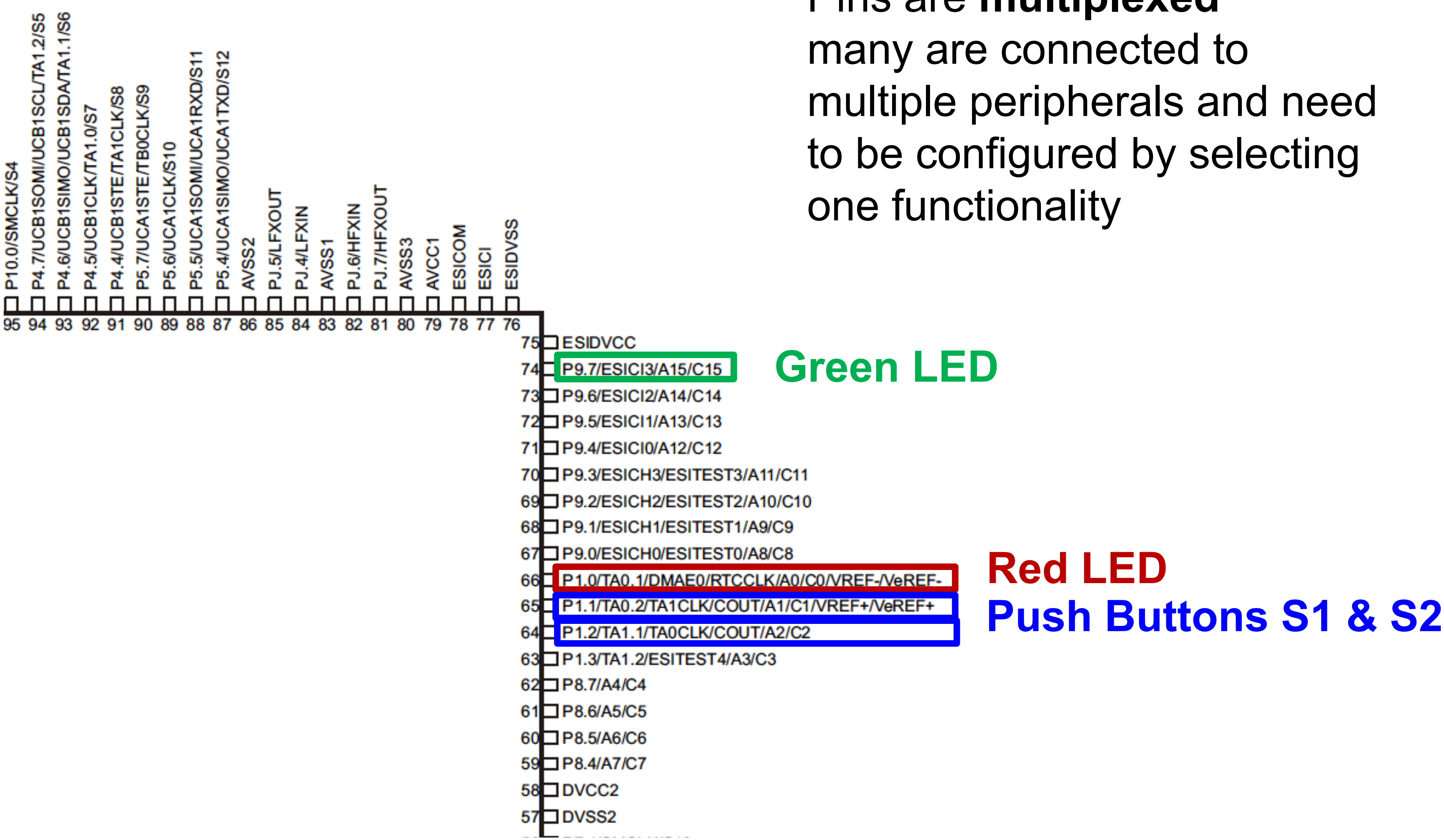
80 pins connected to GPIO Ports

100 total pins



Pins are **multiplexed**

MSP430FR6989IPZ Pinout



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

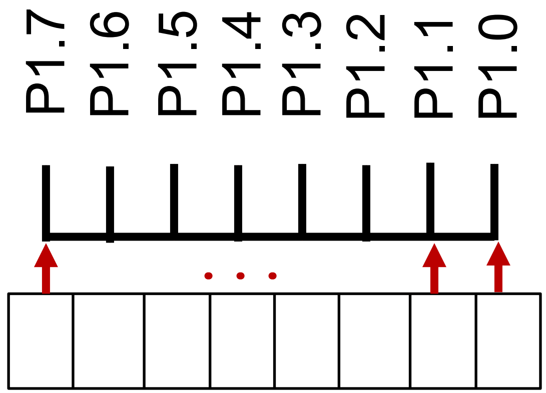
Registers Controlling GPIO Ports

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 to do it?

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

ALWAYS use bit operations
NEVER use move

Addressing GPIO Port Registers

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

byte

immediate mode

absolute mode

(rather than symbolic mode)

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,…

Notation and Instructions

Shorthand notation:

PxDIR.y refers to bit **y** ∈ {0,1, …, 7} of register controlling port **x** ∈ {1, …, 10}

Px.y refers to pin **y** ∈ {0,1, …, 7} of port **x** ∈ {1, …, 10}

Instructions:

e.g.,

```
bic.b   #BITy, &PxOUT
bis.b   #BITy, &PxOUT
```

To check if a bit is 0 or 1

```
bit.b   #BITy, &PxOUT
```

will set the carry bit if bit is 1

clear the carry bit if bit is 0

use **`jc/jnc`**

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**

$\Rightarrow$ 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 **y** ∈ {0,1, …, 7} of register controlling port **x** ∈ {1, …, 10}

Px.y refers to pin **y** ∈ {0,1, …, 7} of port **x** ∈ {1, …, 10}

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
```

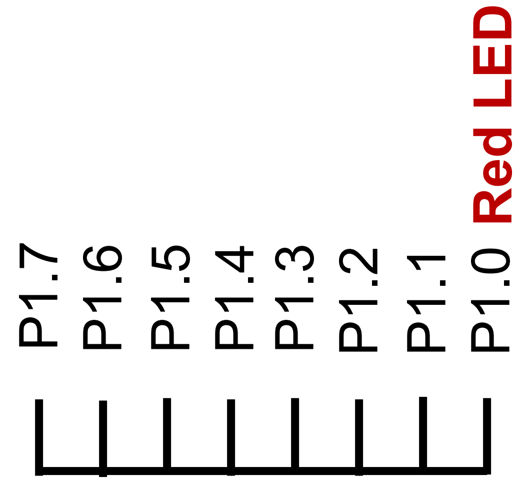
Option 2

GPIO in Action: Blinky v. 1

Task: Make the red LED blink

Go through documentation or slides:

- Red LED is connected to P1.0



- GPIO is default function for Px.y

⇒ No need to change P1SEL0 or P1SEL1

- For GPIO default is PxDIR.y = 0
- i.e., all pins Px.y are configures as input

⇒ Change P1DIR.0 = 1

- What about the output value?

⇒ Toggle it between HIGH and LOW

P1OUT.0 = 1 and P1OUT.0 = 0

GPIO in Action: Blinky v. 1

Task: Make the red LED blink **Red LED is on P1.1**

How do we toggle between **P1OUT.0 = 1** and **P1OUT.0 = 0**?

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

How about a timer?

⇒ Easiest way is to do a countdown timer

Start with a large unsigned value in a register

Decrease until the value hits zero

How do we get the LEDs to light up?

Need to enable GPIO output by clearing the LPM5 lock

```
bic.w #LOCKLPM5, &PM5CTL0
```

GPIO in Action: Blinky v. 1

```
            bis.b   #BIT0, &P1OUT      ; First set output value
            bis.b   #BIT0, &P1DIR      ; Then change direction to output

            bic.b   #LOCKLPM5, &PM5CTL0

toggle:     xor.b   #BIT0, &P1OUT

            mov.w 0xFFFF, R5

countdown:  dec.w   R5
            jnz     countdown

            jmp     toggle
            nop

    ; The whole program is an extended infinite loop,
    ; no need to add another one!
```

Override Power Lock

Can omit this line – only first cycle will be of random length

Exercise: Make the red and green LEDs blink in an alternating pattern.