Embedded System (ES)

Lecturer: Dr. Cheng-Kai Lu
Phone: (02)7749-3554
Office: TD302/BAIR Lab
Email: cklu@ntnu.edu.tw

國立臺灣師範大學
NATIONAL TAIWAN NORMAL UNIVERSITY

Outline

In this lecture, we will cover:

- Review on the **important points** which were covered in the previous lecture
- ARM ARCHITECTURE OVERVIEW(Continuation)
- BITWISE OPERATIONS & GPIOs
- Serial Communication and USART
- Contents to be covered(Presentation and Report)
- Lucky Draw

Recap: RISC vs CISC

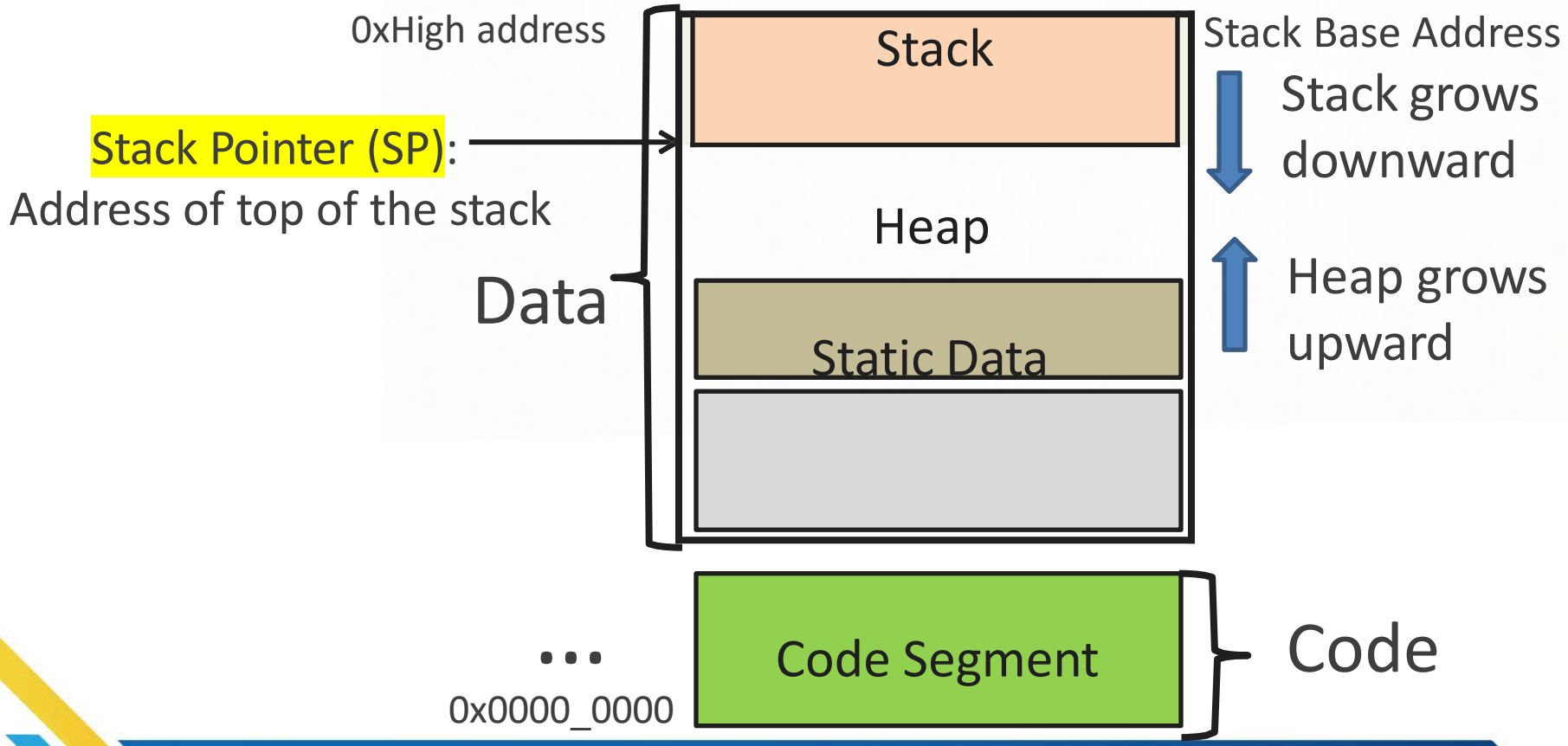
Size / time	**CISC**
1 byte, 1clk	**mov R1, 10**
1 byte, 1clk	**mov R2, 5**
4 byte, 30 clk	**mul R2, R1**

Size / time	**RISC**
1 byte, 1clk	**mov R1, 0**
1 byte, 1clk	**mov R2, 10**
1 byte, 1 clk	**mov R3, 5**
1 byte, 1clk	**Begin: add R1, R2**
1 byte, 1 clk	**loop Begin**

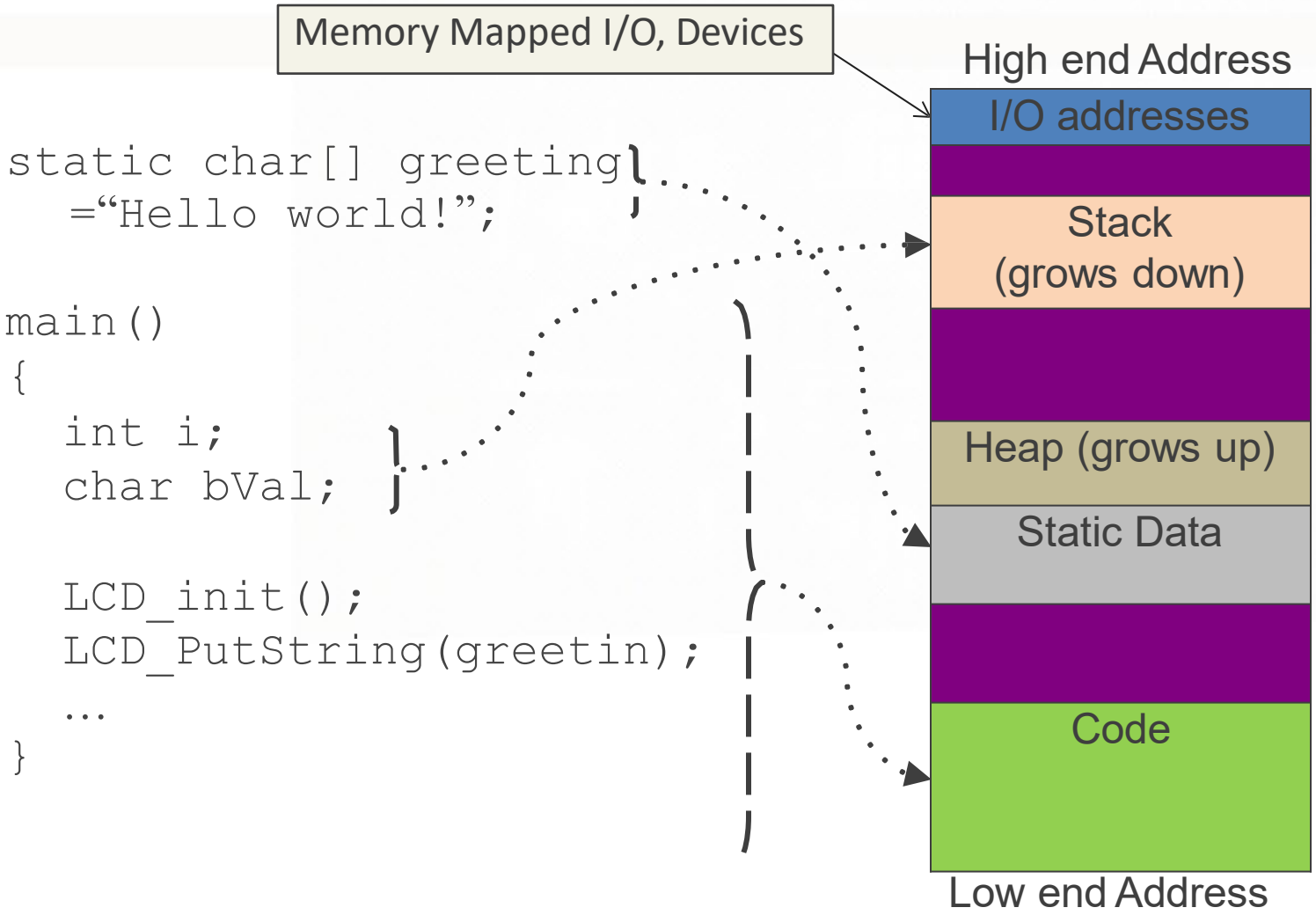
- **CISC**: Instructions often **variable** length and **variable** time
- **RISC**: Instruction typically **constant** length and time
  - **Simpler hardware logic** for decoding instructions (thus typically **faster**)

Recap: Typical ARM Logical Memory Layout

- **Static** Data (e.g. **Global** vars)
- Stack (e.g. local vars, function args & return value, return address)
- Heap (e.g. dynamically allocated memory: malloc, new)
- Code (Code may be in the same or a separate memory device)



Recap: Example Memory Layout



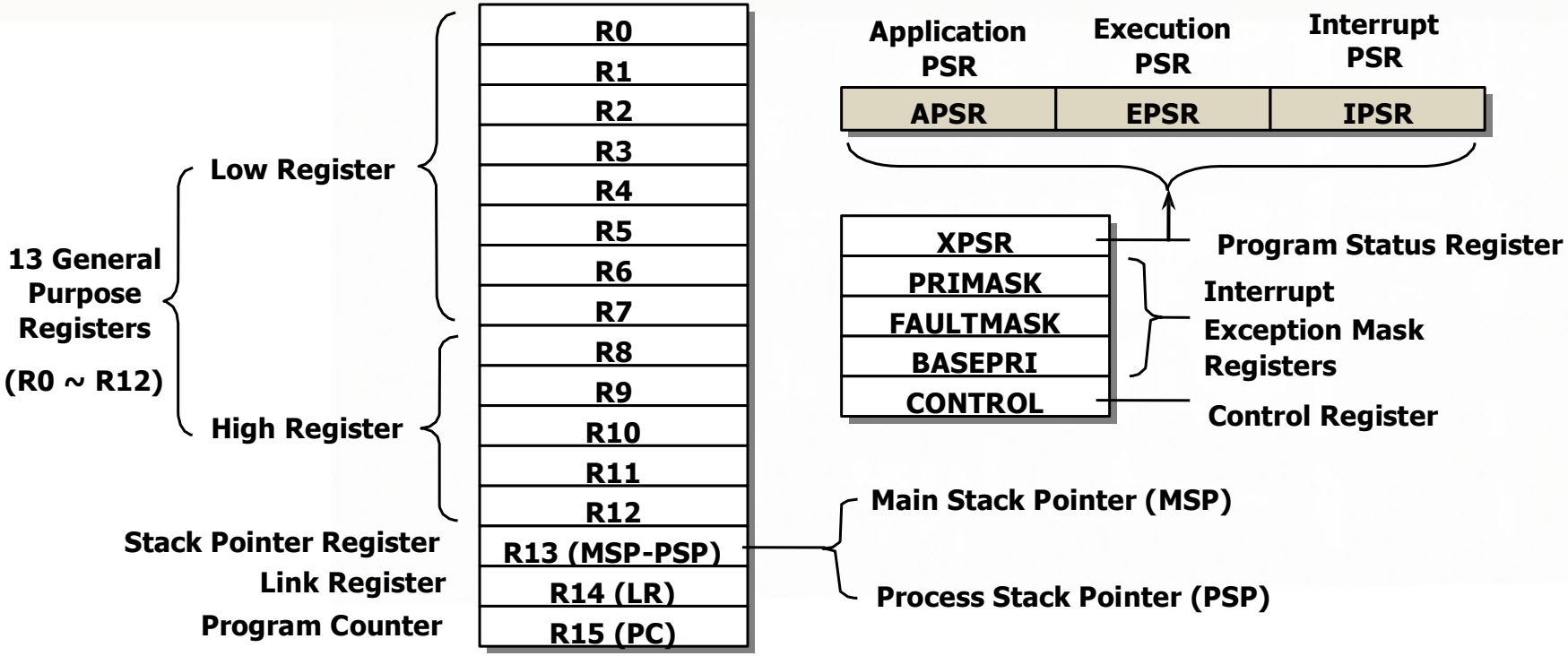
Recap: Example ARM Assembly

```
int x, a, b;
x = (a + b);

MOVW r4,address-of-a; get address for a
LDR r0,[r4]; get value of a
MOVW r4,address-of-b; get address for b
LDR r1,[r4]; get value of b
ADD r3,r0,r1; compute a+b
MOVW r4,address-of-x; get address for x
STR r3,[r4]; store value of x
```

ARM ARCHITECTURE OVERVIEW(Continuation)

General Purpose Register File



A structure block diagram of 21 registers in the Cortex®-M4 Core.

- ARM's register-based design allows fast execution compared to memory-based architectures.
- Efficient use of registers can significantly improve performance.

ARM: GP Registers (Cont.)

- 16(13) 32-bit general purpose registers
  - Used for accessing SRAM
  - Used for storing function parameters (Temporary)
  - Used for instructions to execute operations on
- What is a 32-bit register.
  - Basically just 32 D-Flips connected together

Bit 31	Bit 30	Bit 5	…	…	Bit 2	Bit 1	Bit 0

STATUS REGISTER (SREG)

Status Register (SREG)

Bits	31 30 29 28 27	26:25	24	23:20	19:16	15:10	9	8 7 6 5 4 3 2 1 0
APSR	N Z C V Q				GE*	Reserved		
IPSR	Reserved							Exception Number
EPSR	Reserved	ICI/IT	T	Reserved		ICI/IT	Reserved	

(a) Three individual register – APSR, IPSR and EPSR.

Bits	31 30 29 28 27	26:25	24	23:20	19:16	15:10	9	8 7 6 5 4 3 2 1 0
PSR	N Z C V Q	ICI/IT	T		GE*	ICI/IT		Exception Number

(b) The combined register PSR.

Structure and bit functions in special registers.

*It helps the CPU make decisions and handle interrupts or exceptions efficiently.

Status Register (SREG): Common flags

- Z: **Zero flag**
  - Set to 1 when the result of an instruction is 0
- C: **Carry flag**
  - Set to 1 when the result of an instruction causes a carry to occur
- N: **Negative**
  - Set to 1 to indicate the result of an instruction is negative
- V: **Overflow**
  - Set to 1 to indicate the result of an instruction caused an overflow

*Another type: Condition flags help in **decision-making and branching**.

Usage Example: Checking the Current Interrupt and Flags in ARM Cortex-M

Detect HardFault exceptions (Number = 3): HardFault exceptions occur due to invalid memory access, bus errors, or divide-by-zero errors.

```
uint32_t psr;
__asm volatile ("MRS %0, PSR" : "=r" (psr));

if (psr & (1 << 30)) {
    printf("Zero flag is set!\n");
}

if ((psr & 0x1FF) == 3) {
    printf("Currently handling HardFault Exception!\n");
}
```

Usage Example_How Does It Work?

```
uint32_t psr;
__asm volatile ("MRS %0, PSR" : "=r" (psr));

if (psr & (1 << 30)) {
   printf("Zero flag is set!\n");
}

if ((psr & 0x1FF) == 3) {
   printf("Currently handling HardFault Exception!\n");
}
```

(A) Fetching the PSR Register

- **MRS** (Move to Register from Special Register) is an ARM instruction that reads the **Program Status Register (PSR)** into psr.

(B) Checking the Zero Flag (Z): Bit 30 in APSR checks if the **last operation resulted in zero**.

- If **Z** is set, it means the last arithmetic operation resulted in zero.

(C) Detecting a HardFault Exception Using IPSR

- psr & 0x1FF extracts bits [8:0] of the IPSR.
- IPSR stores the exception number of the currently running interrupt or fault handler.
- If IPSR == 3, it means the CPU is currently handling a HardFault.

Example ARM Assembly

```
int x, a, b;
x = (a + b);

MOVW r4,address-of-a; get address for a
LDR r0,[r4]; get value of a
MOVW r4,address-of-b; get address for b
LDR r1,[r4]; get value of b
ADD r3,r0,r1; compute a+b
MOVW r4,address-of-x; get address for x
STR r3,[r4]; store value of x
```

How to Study Assembly

1. Get to know CPU registers Memorize rules for usage

2. Know basic types of Instruction Memory load and store Arithmetic/Logic Compare and branch

How to Study Assembly

3. Translate C statements
   - Memory accesses
   - Simple arithmetic statements
   - If statement
   - Loop statements
4. Translate C functions
   - Function Linkage
   - Making a function call

How to Study Assembly

5. Interrupt System

   Principle of interrupt and exception

   Interrupt vector table

   Saving and restoring context

Challenges

Challenges in learning Assembly

- Must understand how program works cycle by cycle
  (at hardware level)
- Have to memorize some notations before fully understanding them
- Debugging requires analyzing disassembled code.
  (CPU registers & memory contents)

BITWISE OPERATIONS

Why Bitwise Operation

Why use bitwise operations in embedded systems programming?

Each single bit may have its own meaning

- Push button array: Bit *n* is 0 if push button *n* is pushed
- LED array: Set bit *n* to 0 to light LED *n*

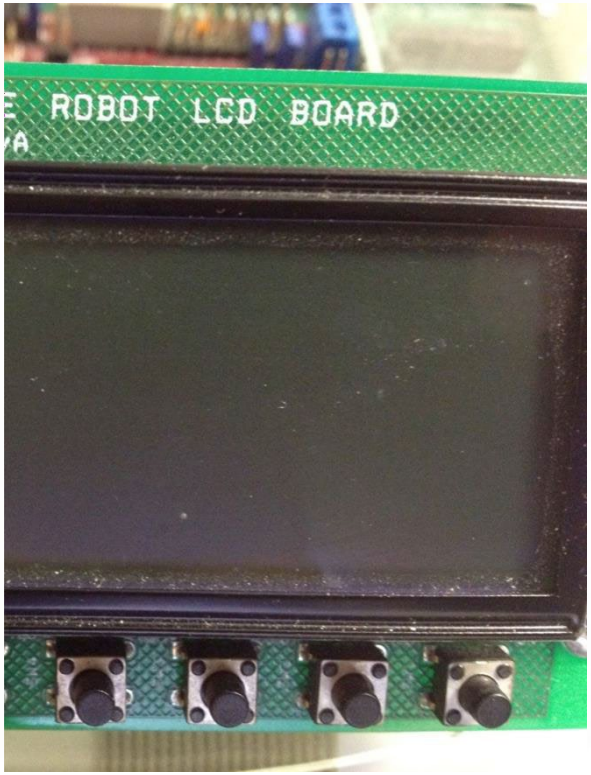
Data from/to I/O ports may be packed

- Two bits for shaft encoder, six bits for push button packed in PINC
- Keypad input: three bits for row position, three bits for column position

Data in memory may be packed to save space

- Split one byte into two 4-bit integers

Why Bitwise Operation



SW4 SW3 SW2 SW1
Bit 3 Bit 2 Bit 1 Bit 0

Read the input:

GPIO_PORTE_R;

How to have application determine which button is being pushed?

Buttons connected to PORTE, bits 3-0

Bitwise Operations

Common programming tasks:

- Clear/Reset certain bit(s)
- Set certain bit(s)
- Test if certain bit(s) are cleared/reset
- Test if certain bit(s) are set
- Toggle/invert certain bits
- Shift bits around

Bitwise Operators: Clear/Force-to-0 Bits

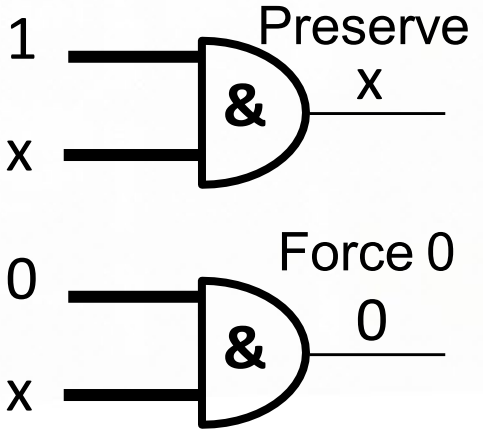
C bitwise AND: **&**

ch = ch & 0x3C; What does it do?

Consider a single bit *x*

x AND 1 = x Preserve

x AND 0 = 0 Clear/Force 0



Truth Table

Bit x	Mask bit	Bit x & Mask bit
x	0	0 (Forced to 0)
x	1	x (Value preserved)

Bitwise Operators: Clear/Force-to-0 Bits

ch = ch & 0x3C;

	x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0
AND	0	0	1	1	1	1	0	0
	0	0	x_5	x_4	x_3	x_2	0	0

Clear bits 7, 6, 1, 0

Preserve bits 5, 4, 3, 2

Clear bit(s): Bitwise-AND with a mask of 0(s)

Bitwise Operators: Clear/Reset Bits

Another example:

char op1 = 1011 1100; We want to clear bit 4 to 0.

char op2 = 1110 1111; We use op2 as a mask

char op3;

op3 = op1 & op2;

```
        1011 1100
AND     1110 1111
-------------------
        1010 1100
```

Bitwise Operators: Set Bits

C bitwise OR: |

ch = ch | 0xC3; What does it do?

Consider a single bit *x*

x OR 1 = 1 Set

x OR 0 = x Preserve

Bitwise Operators: Set Bits

ch = ch | 0xC3;

	x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0
OR	1	1	0	0	0	0	1	1
	1	1	x_5	x_4	x_3	x_2	1	1

Set bits 7, 6, 1, 0

Preserve bits 5, 4, 3, 2

Set bit(s): Bitwise-OR with a mask of 1(s)

Bitwise Operators: Set Bit

Another example:

char op1 = 1000 0101; We want to set bit 4 to 1.

char op2 = 0001 0000; We use op2 as a mask

char op3;

op3 = op1 | op2;

```
  1000 0101
OR 0001 0000
-----------
  1001 0101
```

Bitwise Operators: Toggle Bits

C bitwise XOR: ^

ch = ch ^ 0x3C; What does it do?

Consider a single bit *x*

x XOR 1 = ~~x~~ Toggle

x XOR 0 = x Preserve

Bitwise Operators: Toggle Bits

C bitwise XOR: ^ ch = ch ^ 0x3C;

	x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0
XOR	0	0	1	1	1	1	0	0
	x_7	x_6	$\overline{x_5}$	$\overline{x_4}$	$\overline{x_3}$	$\overline{x_2}$	x_1	x_0

Toggle bits 5, 4, 3, 2

Preserve bits 7, 6, 1, 0

Toggle bit(s): Bitwise-XOR with a mask of 1(s)

Bitwise Operators: Invert Bits

C bitwise invert: **~** ch = ~ch;

INV	x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0
	$\overline{x_7}$	$\overline{x_6}$	$\overline{x_5}$	$\overline{x_4}$	$\overline{x_3}$	$\overline{x_2}$	$\overline{x_1}$	$\overline{x_0}$

Example: ch = 0b00001111;

~ch == 0b11110000

Primitive Types and Sizes

Name	Number of Bytes sizeof()	Range
char	1	0 to 255 or -128 to 127 (Depends on Compiler settings)
signed char	1	-128 to 127
unsigned char	1	0 to 255
short	2	-32,768 to 32,767
unsigned short	2	0 to 65,535
int	Varies by platform	Varies by platform
int (on TM4C123)	4	-2,147,483,648 to 2,147,483,647
unsigned int (on TM4C123)	4	0 to 4,294,967,295
(pointer)	Varies by platform	Varies by platform
(pointer on TM4C123)	4	Address Space

- Primitive types in C: char, short, int, long, float, double default modifier on primitive types is **signed** (not unsigned)
- Note: char does not have a standard default, it depends on Compiler settings

Class Exercise

unsiged char ch; short n;

Using bitwise operations with a bitmask, perform the following tasks:

Force the lower half of ch to 0

Starting from bit 0 of ch, force every other bit to 1

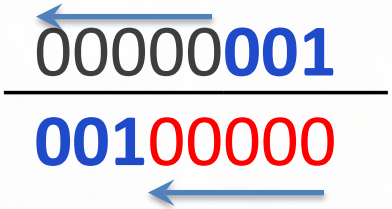
Force bit 15 and bit 0 of n to 0

Toggle bits 7 and 6 of ch

Bitwise Operators: Shift-Left

```
unsigned char my_reg = 0b00000001;
unsigned char shift_amount = 5;
unsigned char my_result;

my_result = my_reg << shift_amount;
```



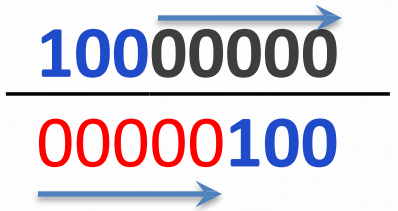
<<, shifts “my_reg”, “shift_amount” places to the left

0s are shifted in from the right

Bitwise Operators: Shift-Right Logical

```
unsigned char my_reg = 0b10000000;
unsigned char shift_amount = 5;
unsigned char my_result;

my_result = my_reg >> shift_amount;
```



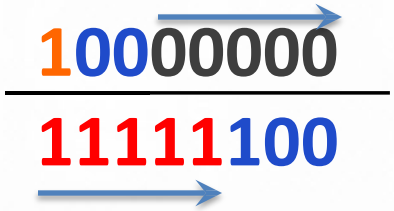
With unsigned type, >> is **shift-to-right logical**

0s are shifted in from the left

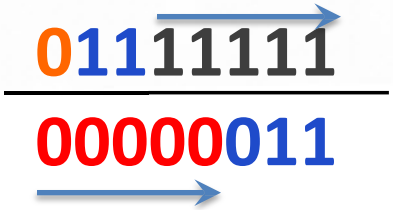
Bitwise Operators: Shift-Right Arithmetic

signed char my_reg = 0b10000000;
unsigned char shift_amount = 5;
unsigned char my_result;

my_result = my_reg >> shfit_amount;



my_reg = 0b01111111;
my_result = my_reg >> shfit_amount;



With signed type, >> is **shift-right arithmetic**
Sign bit value are shifted in from the left

Bitwise Operators: Shift and Multiple/Divide

n << k **is equivalent to** **$n * 2^k$**

Example: 5 << 2 = 5*4 = 20

0b0000 0101 << 2 = 0b0001 0100

n >> k **is equivalent to** **$n / 2^k$**

Example: 20 >> 2 = 5

0b0001 0100 >> 2 = 0b0000 0101

Bitwise Operators: Shift and Multiple/Divide

Shift-Right Arithmetic: Why shift in the sign bit?

Example: (char) 32 >> 2 = 32 / 4 = 8

0b0010 0000 >> 2 = 0b0000 1000

Example: (char) -32 >> 2 = -32/ 4 = -8

0b1110 0000 >> 2 = 0b1111 1000

Bitwise Operators: Shift and Set

What's the effect of the following state?

```
#define  BIT_POS   4
ch = ch | (1 << BIT_POS);
```

What is (1 << 4)?

```
0000 0001          << 4
0001 0000
```

In general case: (1 << n) yields **a mask of a 1 at bit n**

The effect of the statement: Set bit 4

Bitwise Operators: Shift and Set

Another example:

unsigned char my_mask = 0000 0001;

unsigned char shift_amount = 5;

unsigned char my_result = 1101 0101; Want to force bit 5 to a 1

my_result = my_result | (my_mask << shift_amount);

1101 0101 | 00100000

1101 0101

OR 0010 0000

1111 0101

Shift the 1(s) of the MASK to the appropriate position, then OR with my_result to force corresponding bit positions to 1.

Bitwise Operators: Shift and Clear

What's the effect of the following state?

```
#define  BIT_POS   4
ch = ch & ~(1 << BIT_POS);
```

What is ~(1 << 4)?

```
0000 0001        << 4
0001 0000        ~
1110 1111
```

In general case: ~(1 << n) yields **a mask of a 0 at bit n**

Note: Compiler does the calculation at compilation time

Bitwise Operators: Shift and Clear

unsigned char my_mask = 0000 0111;

unsigned char shift_amount = 5;

unsigned char my_result = 1011 0101; Want to force bit 7-5 to a 0

my_result = my_result & ~(my_mask << shift_amount);

1011 0101 & ~ 11100000

1011 0101 & 00011111

1011 0101

AND 0001 1111

0001 0101

Shift the 0(s) of the MASK to the appropriate position, then AND with my_result to force corresponding bit positions to 0.

Exercise

unsigned char ch;

unsigned short n;

Divide n by 32 in an efficient way

Swap the upper half and lower half of ch

Bitwise Testing

Reminder: Conditionals (if, while) are evaluated on the basis of zero and non-zero (i.e. Boolean).

The quantity 0x80 is non-zero and therefore TRUE.

```
if (0x02 | 0x44)
```

TRUE or FASE?

Bitwise Testing

Example

Find out if bit 7 of variable nVal is set to 1

Bit 7 = 0x80 in hex

```
if ( nVal & 0x80 )
{     …
}
```

What happens when we want to test for multiple bits?

if statement looks only for a non-zero value, a non-zero value means at least one bit is set to TRUE

Bitwise Testing

Example 1: When nVal = 0x80 (Bit 7 = 1, Bit 6 = 0)
nVal = 0x80 = 1000 0000
Mask = 0xC0 = 1100 0000

Result = 1000 0000 (Nonzero → TRUE)

Example 2: When nVal = 0x40 (Bit 7 = 0, Bit 6 = 1)
nVal = 0x40 = 0100 0000
Mask = 0xC0 = 1100 0000

Result = 0100 0000 (Nonzero → TRUE)

Example 3: When nVal = 0x00 (Both Bits 7 and 6 = 0)
nVal = 0x00 = 0000 0000
Mask = 0xC0 = 1100 0000

Result = 0000 0000 (Zero → FALSE)

Bitwise Testing: Any Bit is Set to 1?

Example

See if bit 2 or 3 is set to 1

Bits 2,3 = 0x0C in hex

```
if (nVal & 0x0C)
{
    Some code…
}
```

What happens for several values of nVal?

nVal = 0x04	bit 2 is set	Result = 0x04	TRUE
nVal = 0x0A	bits 3,1 are set	Result = 0x08	TRUE
nVal = 0x0C	bits 2,3 are set	Result = 0x0C	TRUE

Bitwise Testing: All Bits Are Set to 1?

Why does this present a problem?

- What happens if we want to see if both bits 2 and 3 are set, not just to see if one of the bits is set to true?
- Won't work without some other type of test

Two solutions

- Test each bit individually

```
if ( (nVal & 0x08) && (nVal & 0x04))
```

- Check the result of the bitwise AND

```
if ((nVal & 0x0C) == 0x0C)
```

Why do these solutions work?

1. Separate tests – Check for each bit and specify logical condition
2. Equality test – Result will only equal 0x0C if bits 2 and 3 are set

Bitwise Testing

- Testing if any of a set of bits is set to 1
  1) Decide which bits you want to test
  2) Isolate those bits (i.e. force all other bits to 0)
- Testing if all bits of a set of bits are set to 1
  1) Decide which bits you want to test
  2) Isolate those bits (i.e. force all other bits to 0)
  3) Compare for equality with the Mask
- For the case of testing for bits set to 0. Follow bit(s) set to 1 testing procedure, but invert the variable that you are testing.
- Generic systematic checking example

```
if( (x & MASK_ALL1s) == MASK_ALL1 &&
    (~x & MASK_ALL0s) == MASK_ALL0s &&
    (x & MASK_ANY1s)  &&
    (~x & MASK_ANY0s) )
```

Exercise

char ch;

Test if any of bits 7, 6, 5, 4 is set to 1

Test if all of bits 7, 6, 5, 4 are set to 1

Test if all of bits 7 and 6 are set to 1, and if bits 5 and 4 are cleared to 0

Bitwise operations: Summary

- Forcing bits to 0: & (AND)
- Forcing bits to 1: | (OR)
- Toggle bits: ^ (XOR)
- Testing for bits set to 1
- Testing for bits cleared to 0
- Generic systematic testing approach:

```
if( (val & MASK_ALL1s) == MASK_ALL1 &&
    (~val & MASK_ALL0s) == MASK_ALL0s &&
    (val & MASK_ANY1s)   &&
    (~val & MASK_ANY0s) )
```

***Where: MASK_XXX is a mask with 1s in the positions being tested**

***Bitwise operations** are commonly used in embedded systems for **checking hardware status flags, GPIO states, and configuring registers**.

General Purpose Input Output (GPIO)

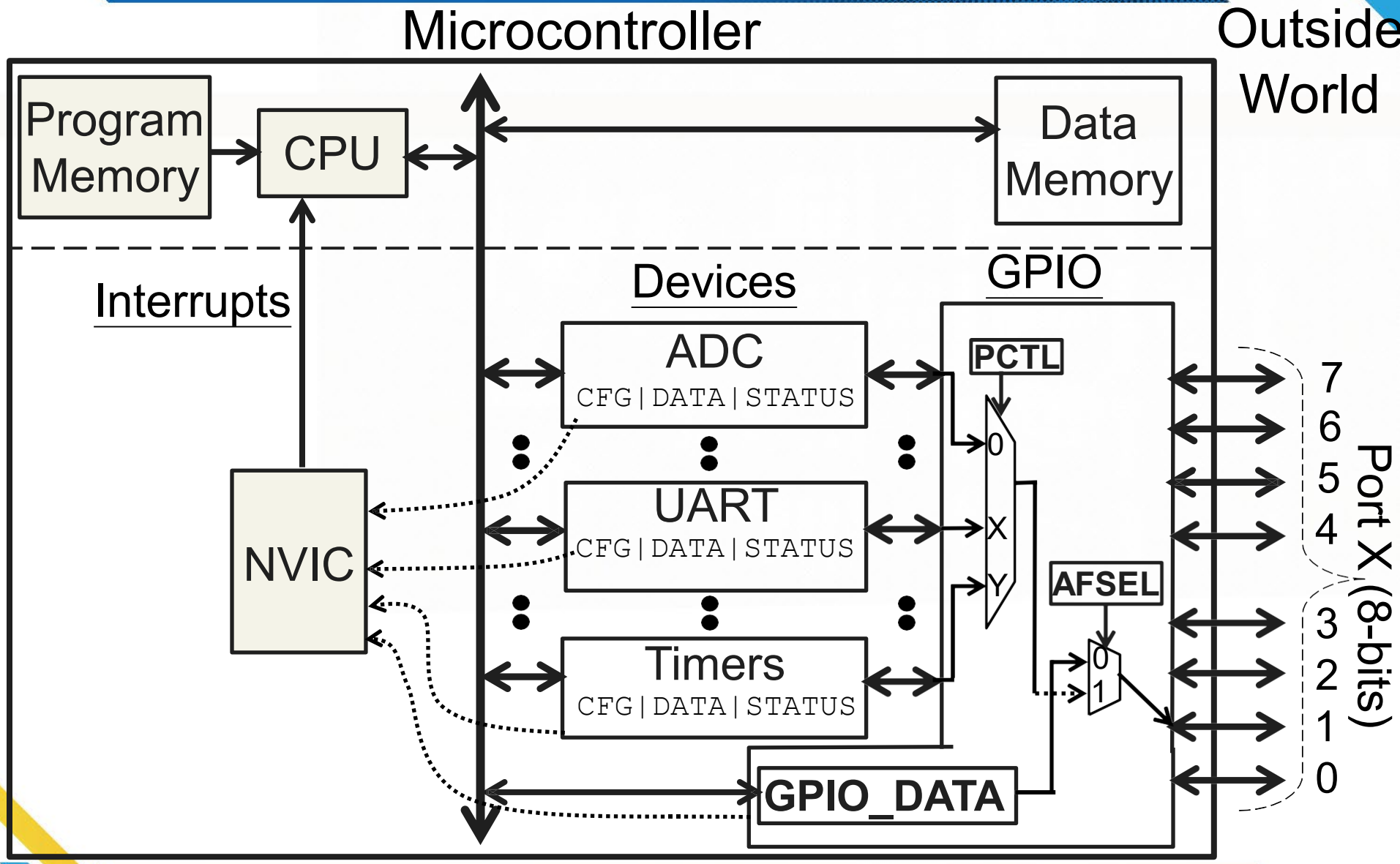
General Purpose Input Output (GPIO)

- General Purpose Input Output (GPIO) - its embedded programming as input/output pins are the only way to interface with the microcontroller.
- Each pin of GPIO can be set up to accept or source different logic voltages, with configurable drive strengths and pull ups/downs. Input and output voltages are usually, but not always, limited to the supply voltage of the device with the GPIOs, and may be damaged by greater voltages.
- GPIO pins can be used for driving loads, reading digital and analog signal, controlling external components, generating triggers for external devices etc.

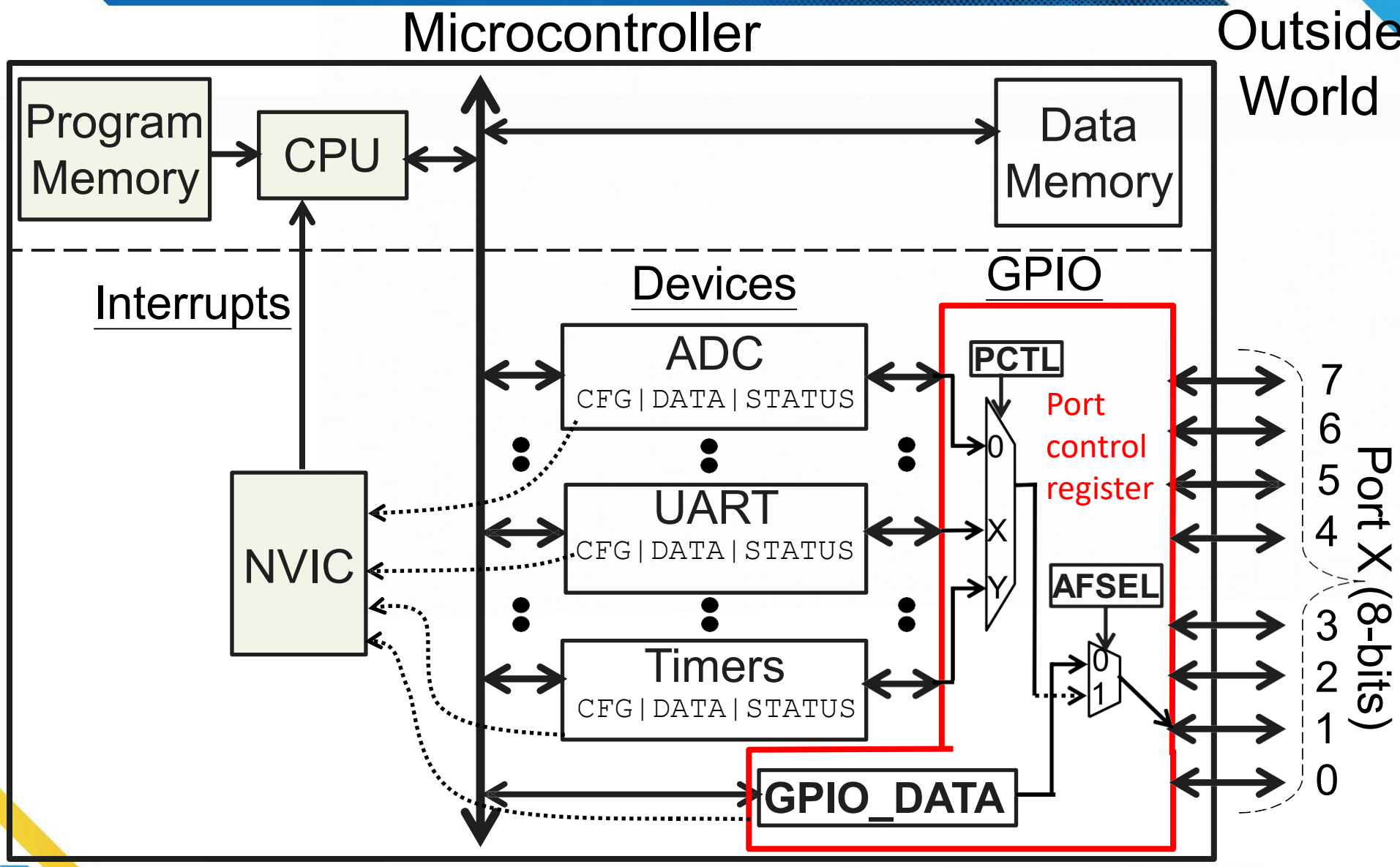
General Purpose Input Output (GPIO)

- GPIO abilities may include:
  - GPIO pins can be configured to be input or output
  - GPIO pins can be enabled/disabled
  - Input values are readable (usually high or low)
  - Output values are writable/readable
  - **Alternate Function Mode:** Some GPIOs can be repurposed for **communication** protocols (UART, I2C, SPI).
- Input values can often be used as **Interrupt Request (IRQs),** which is usually for wakeup events. GPIO pins can be used for driving loads, reading digital and analog signal, controlling external components, generating triggers for external devices etc.

Microcontroller / System-on-Chip (SoC)



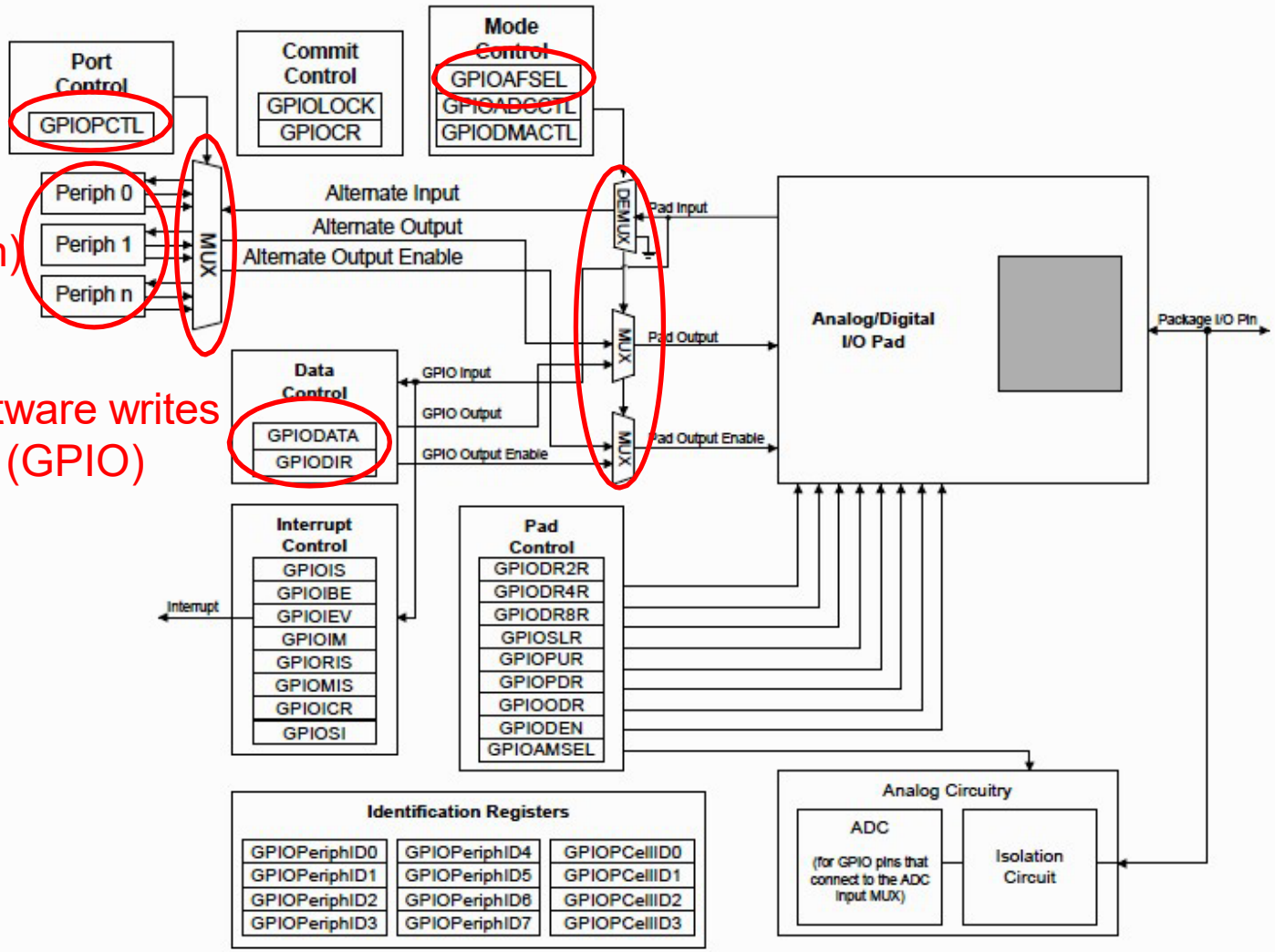
Microcontroller / System-on-Chip (SoC)



GPIO Port: Alternative Function Architecture

Hardware writes (Alternative Function)

Software writes (GPIO)



INTERRUPT SERVICE ROUTINES(ISR)

Interrupts and Interrupt Service Routines (ISRs)

Interrupt (IRQ): A mechanism that allows hardware to inform the CPU that an event has occurred. The CPU stays free until the event happens. Example hardware events:

- Button connected to a GPIO port pressed
- Data arrives to the system. e.g. new byte on UART interface, new sample from analog to digital converter (ADC)
- Timer expires or Timer becomes equal to a given value
- CPU tries to execute an invalid assembly instruction

Interrupt Service Routine (ISR): code to be executed to deal with the hardware event that has occurred. Also referred to as an Interrupt handler.

General Interrupt to ISR flow

1. The **CPU runs normal tasks**.
2. Hardware event causes an interrupt to occur
3. CPU pauses the program
4. CPU disables interrupts and saves its "state"
   - The CPU state is the information the CPU needs to un-pause the program properly. e.g. location of the instruction when the paused occurred, current CPU register values
5. CPU re-enables interrupts and executes the proper ISR
6. Once the ISR completes, the CPU disables interrupts and restores its state to what it was before the interrupt occurred
7. CPU re-enables interrupts, and continues the program from where it paused.

Polling vs. Interrupts: Key Differences

Feature	Polling	Interrupt (IRQ)
CPU Usage	High (always checking)	Low (responds only when needed)
Response Time	Slower (fixed intervals)	Faster (immediate reaction)
Power Efficiency	Poor (wastes energy in loops)	Excellent (CPU sleeps until needed)
Use Case	Simple applications, low-priority tasks	Real-time systems, event-driven tasks
Complexity	Easier to implement	Requires ISR and handling

When to Use Polling vs. Interrupts

Scenario	Best Choice
Checking a simple sensor periodically	Polling
High-speed real-time response needed	Interrupt
Power-constrained embedded devices	Interrupt (lower energy consumption)
Time-critical applications (motor control, network packets, audio processing, etc.)	Interrupt

✅ **Polling is simple but inefficient**, best for low-priority tasks.
✅ **Interrupts are powerful and efficient**, best for real-time and event-driven systems.
✅ **Most modern embedded systems use a mix of polling and IRQs** to balance performance and power efficiency.

Nested Vector Interrupt Controller (NVIC)

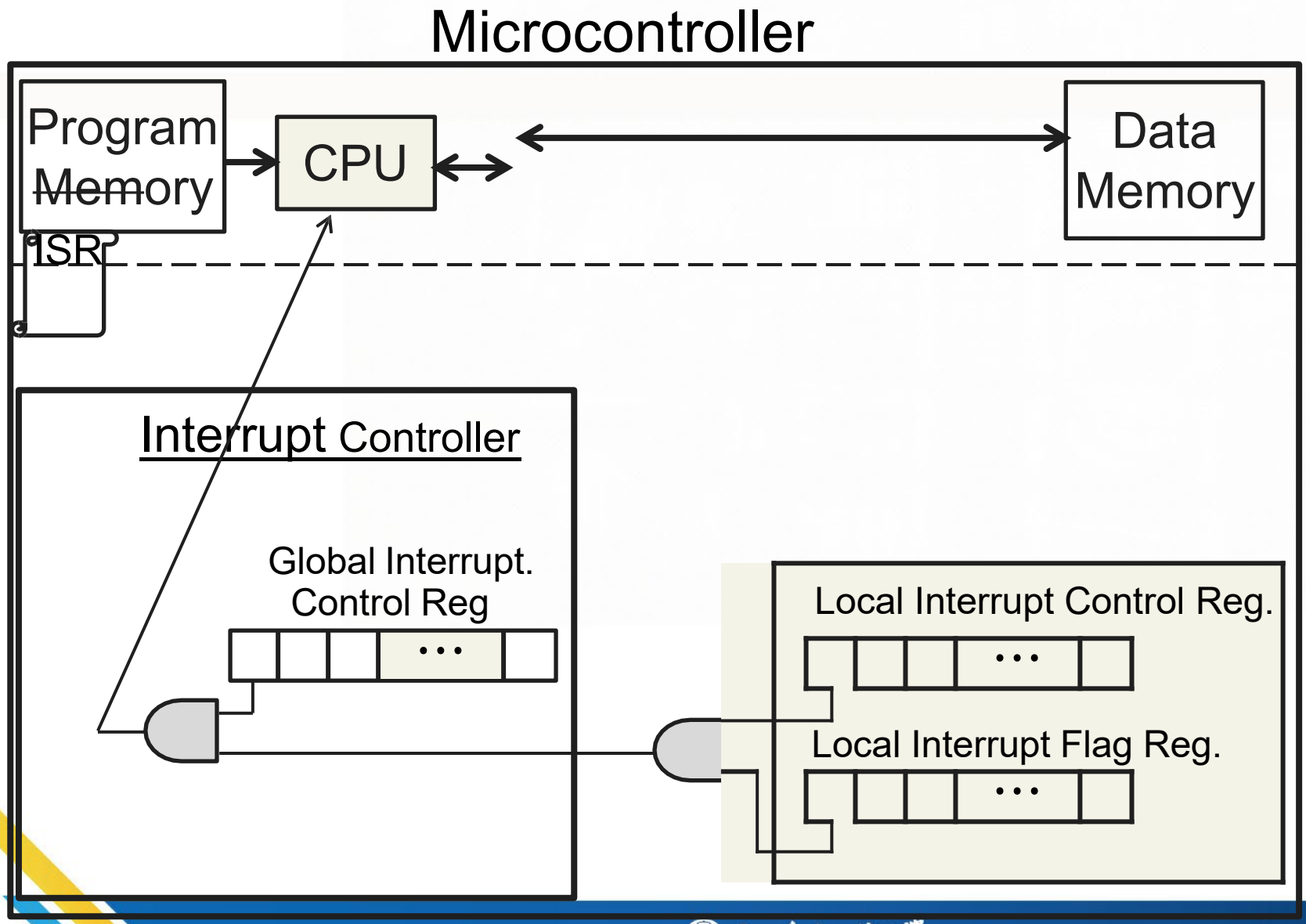
- **NVIC**: the name of the hardware on the ARM (e.g. CPRE 288) microcontroller chip that manages interrupts
  - Notifies CPU when an interrupt occurs
  - Programmer configures to enable/disable specific interrupts
  - Programmer configures to give interrupts priorities
  - Provides the CPU with information for accessing an *Interrupt Vector Table*, which stores the starting address (i.e. entry point) of each ISR.
- **Interrupt Vector Table**: Each row in this table (located in memory) contains the address of the starting instruction for each ISR. The CPU uses this information to start execution of the ISR that has been "triggered" by a corresponding Interrupt (i.e. hardware event)

Polling vs. Interrupts vs. ISR vs. NVIC

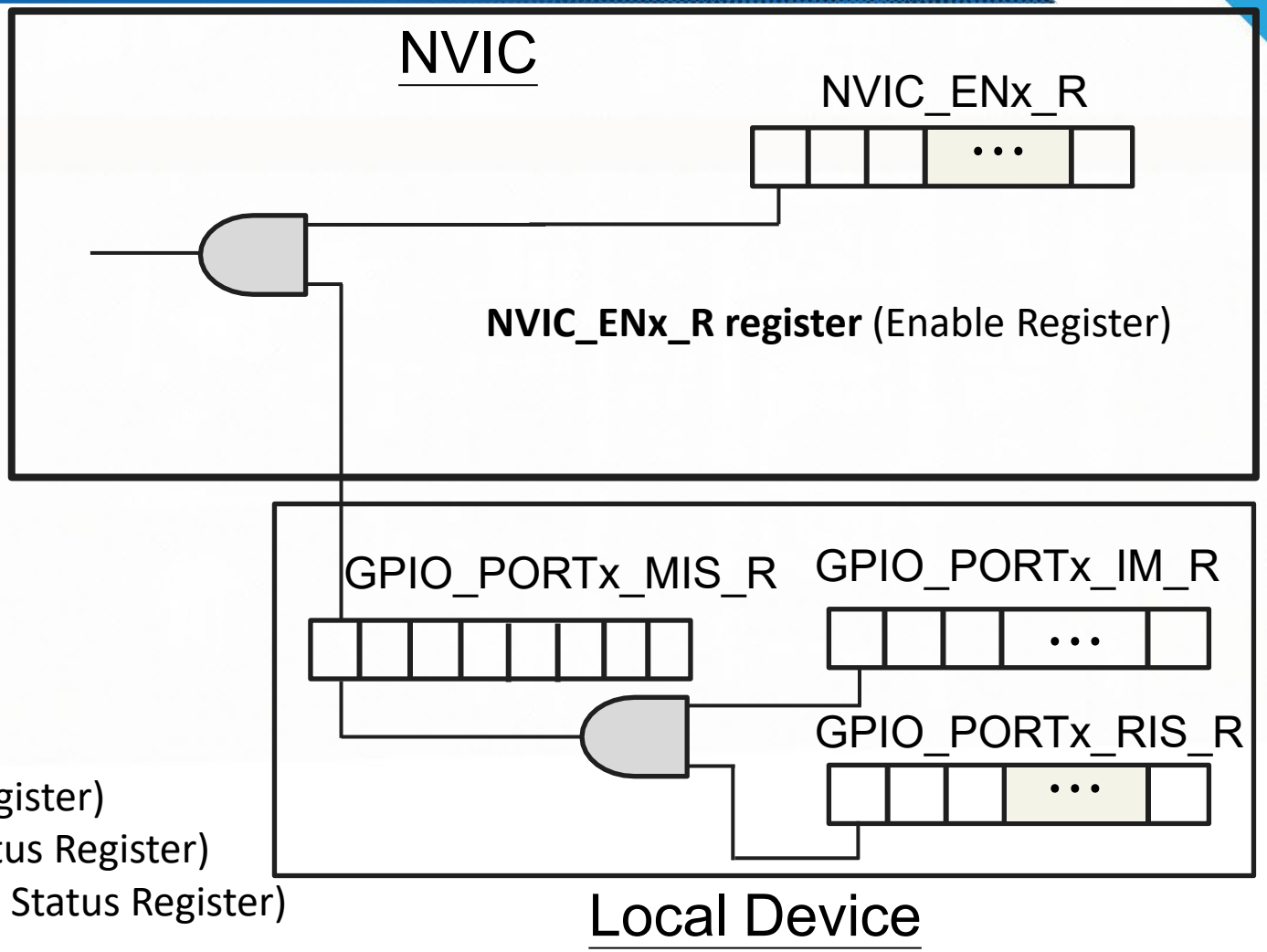
Concept	Definition	How It Works	Use Case
Polling	CPU repeatedly checks a device	Uses a loop to check for an event	Good for simple, non-time-critical tasks
Interrupt (IRQ)	A hardware signal tells the CPU to stop and respond	CPU stops normal execution and calls an ISR	Used for real-time and event-driven applications
ISR (Interrupt Service Routine)	The function that executes when an interrupt occurs	Handles the event, then CPU resumes normal tasks	A short function handling hardware events
NVIC (Nested Vectored Interrupt Controller)	Manages multiple interrupts and their priorities	Determines which interrupt should execute first	ARM Cortex-M systems for efficient interrupt handling

***Interrupts + NVIC help build efficient real-time systems** (e.g., motor control, sensor data processing).

Generic Interrupt Controller



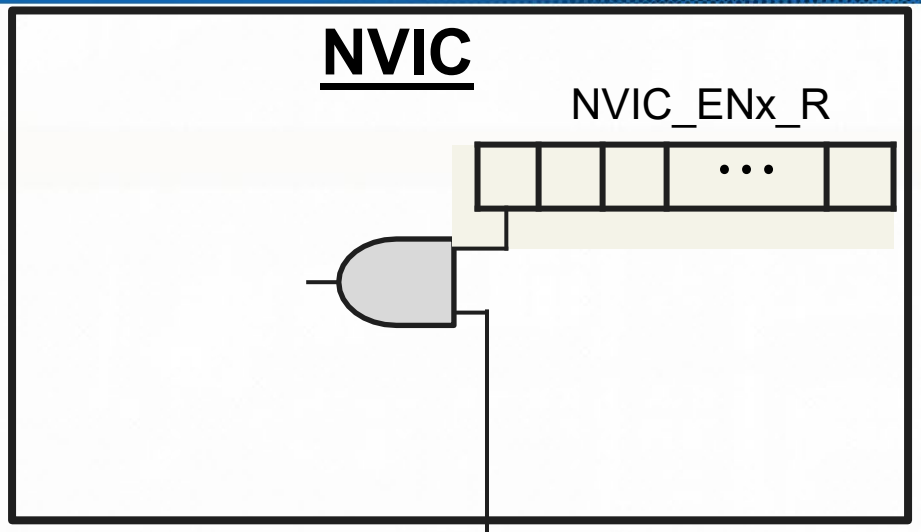
NVIC – GPIO Example



*IM_R (Interrupt Mask Register)
RIS_R (Raw Interrupt Status Register)
MIS_R (Masked Interrupt Status Register)

- ✅ **NVIC manages which interrupts get processed by the CPU.**
- ✅ **ISRs must clear the interrupt flag to prevent re-triggering.**

NVIC – GPIO Example

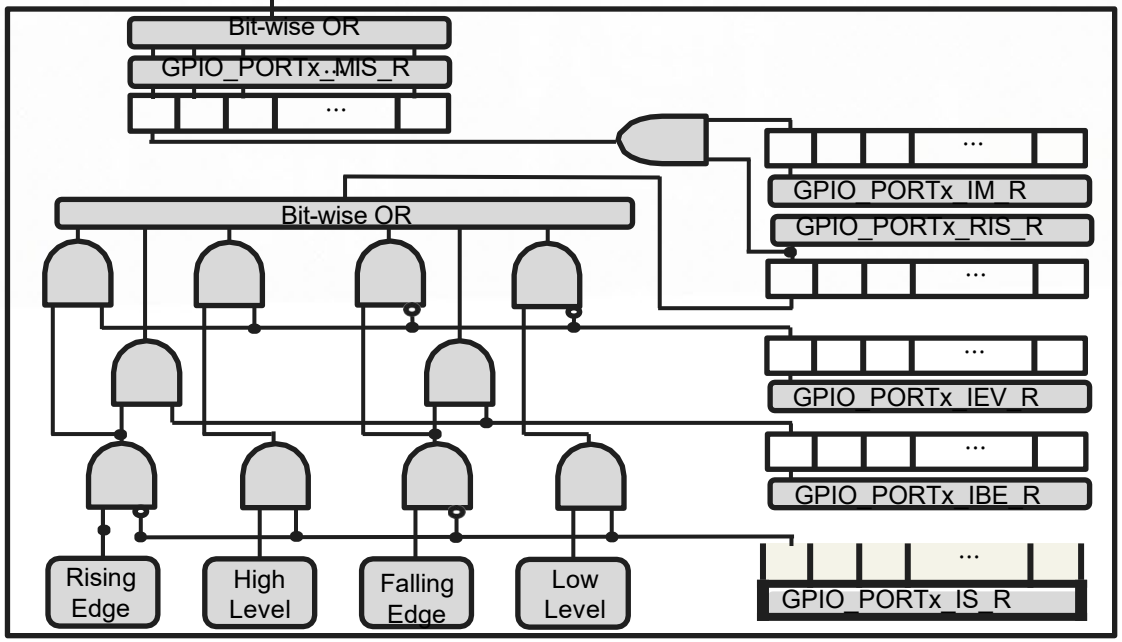


Which input triggers an interrupt?

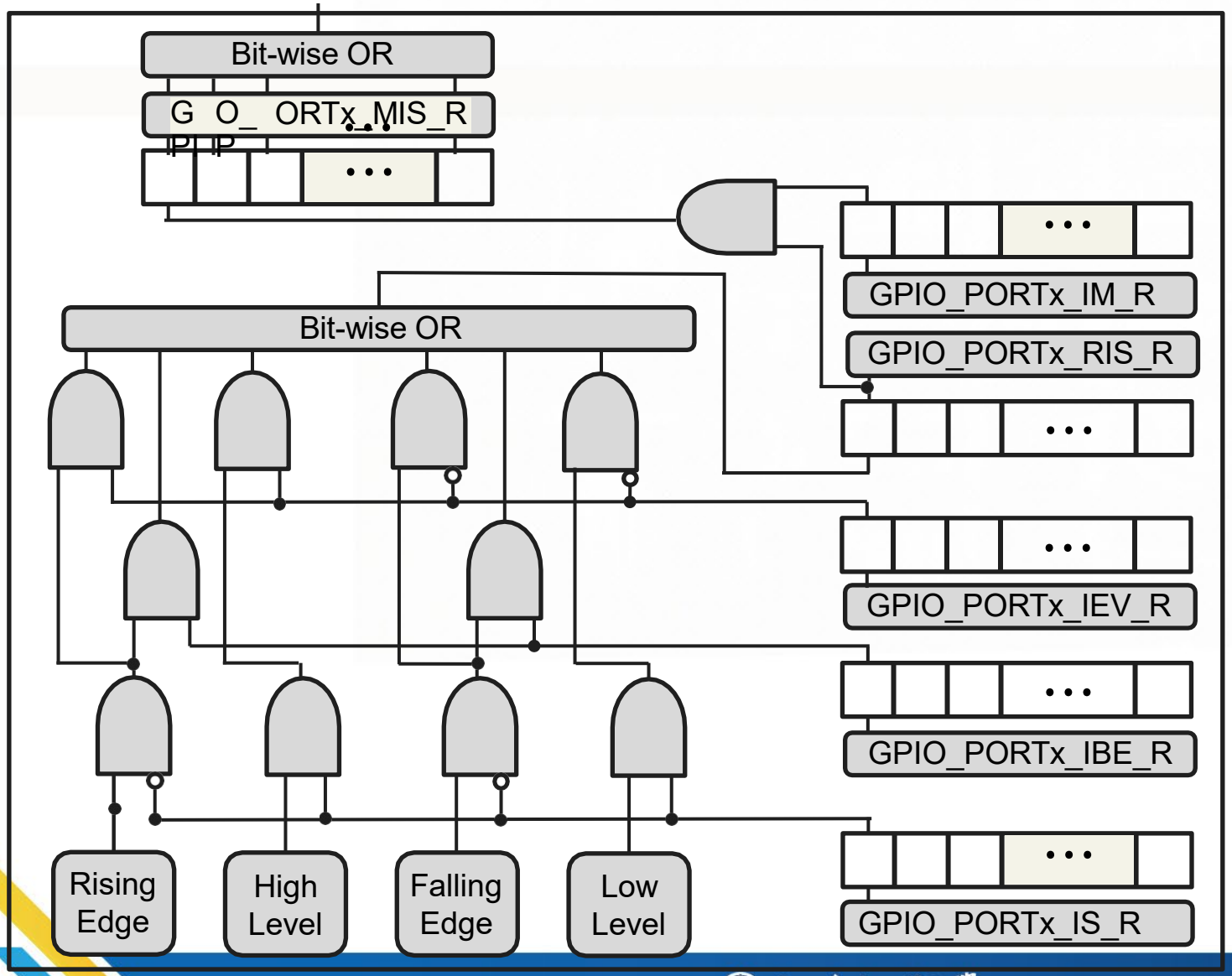
*GPIO_PORTx_IS_R
(Interrupt Sense Register)
If 1 → Level-sensitive
If 0 → Edge-sensitive

*GPIO_PORTx_IEV_R (Interrupt Event Register)
If 1 → High Level or Rising Edge
If 0 → Low Level or Falling Edge

*IBE_R (Interrupt Both Edges Register)
- If 1 → Interrupt triggered on both edges.
If 0 → Controlled by the IEV register.



NVIC – GPIO Example



Which input triggers an interrupt?

UART Interface Overview

Overview of UART

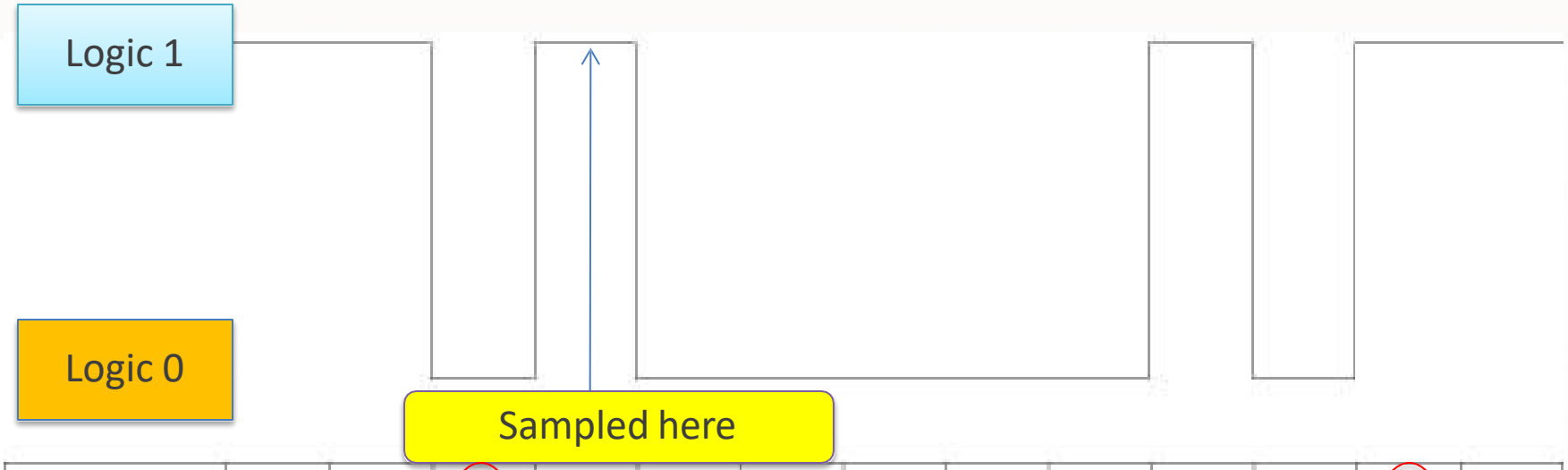
- Concepts behind Serial Communication
- TM4C123g UART Programming Interface
- Initializing UART, transmitting and receiving data

Serial Communication

- USART = Universal Synchronous & Asynchronous Serial Receiver & Transmitter
- Asynchronous (no common clock)
- Can transmit over long link distances
- Uses *start* and *stop* to sandwich data bits
- *parity bit* can be used for error detection

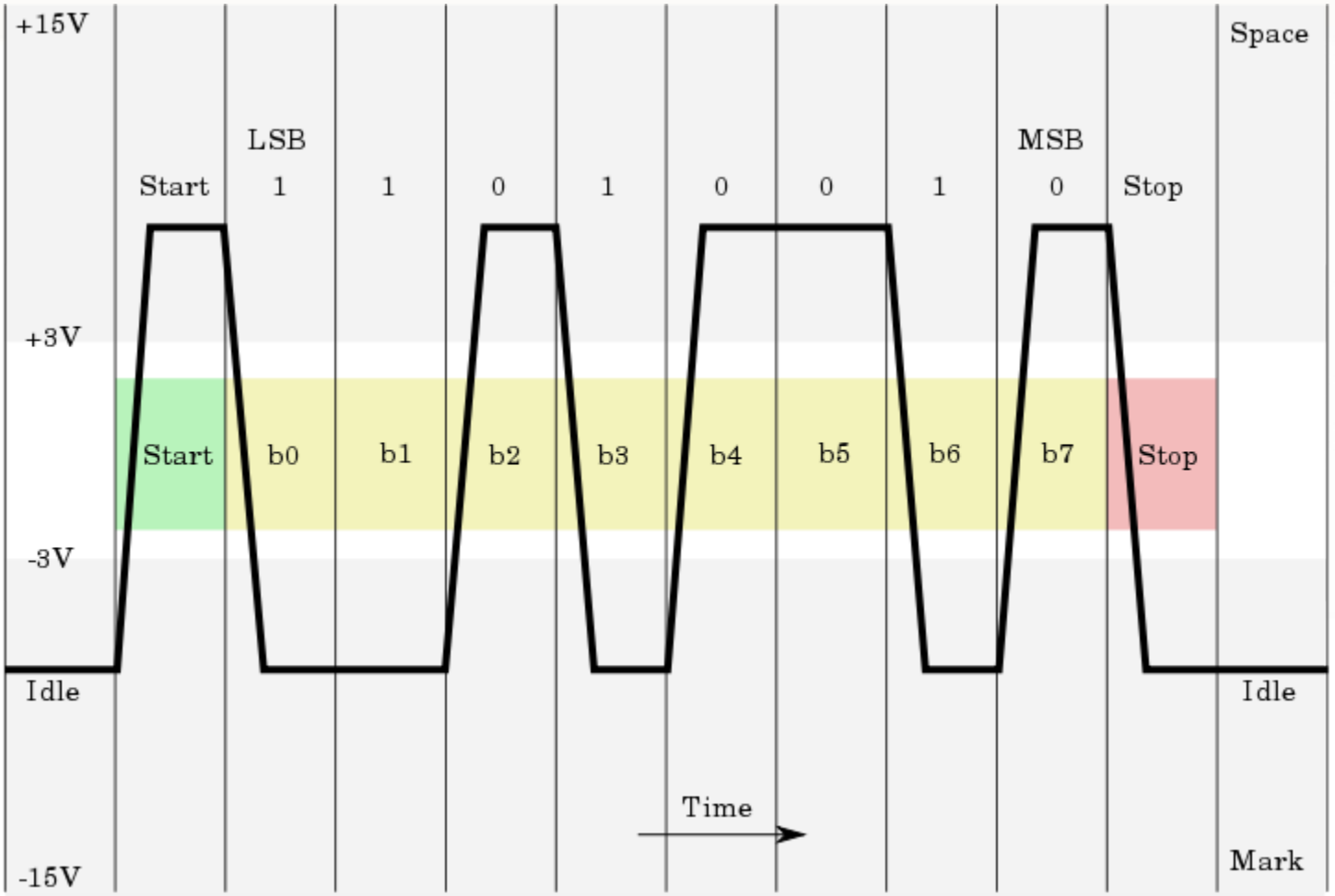
  - (on right) RS-232 Serial Cable

Serial Byte Format

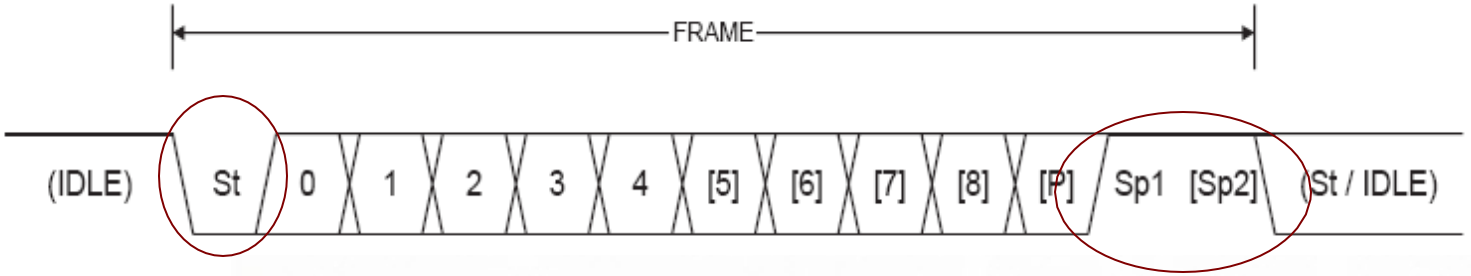


State	Idle	Idle	Active	Active	Active	Active	Active	Active	Active	Active	Active	Active	Idle
Logic Level	1	1	0	1	0	0	0	0	0	1	0	1	1
Bit Type			Start	Data	Data	Data	Data	Data	Data	Data	Data	Stop	
Data Bit Number				0	1	2	3	4	5	6	7		

Serial Communication



Start and Stop Bits

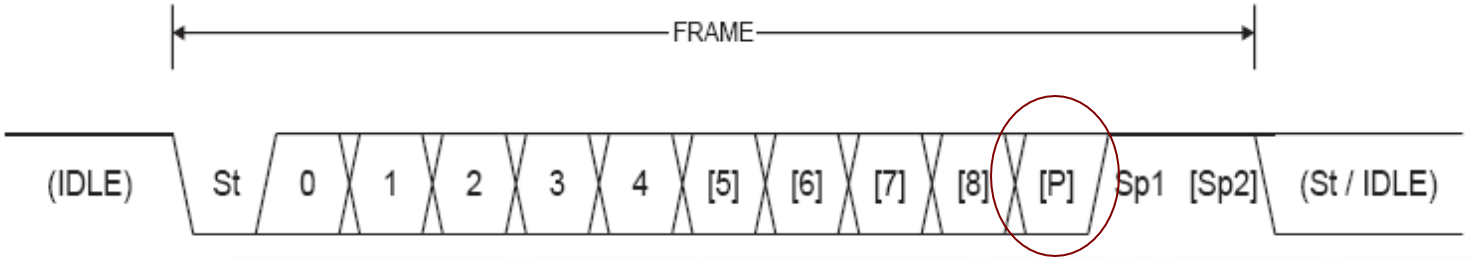


Idle period: logic high

Start bit: **logic low**, 1 bit

Stop bit: **logic high**, 1 bit or 2 bits

Parity Bit



Three choices: **even, odd, or none**

$$P_{even} = d_{n-1} \oplus \ldots \oplus d_3 \oplus d_2 \oplus d_1 \oplus d_0 \oplus 0$$
$$P_{odd} = d_{n-1} \oplus \ldots \oplus d_3 \oplus d_2 \oplus d_1 \oplus d_0 \oplus 1$$

Baud Rate

How to define communication speed?

Baud rate: Number of symbols(/signal units) transferred per second

Baud rate is **not** data rate

Baud Rate vs Bit Rate

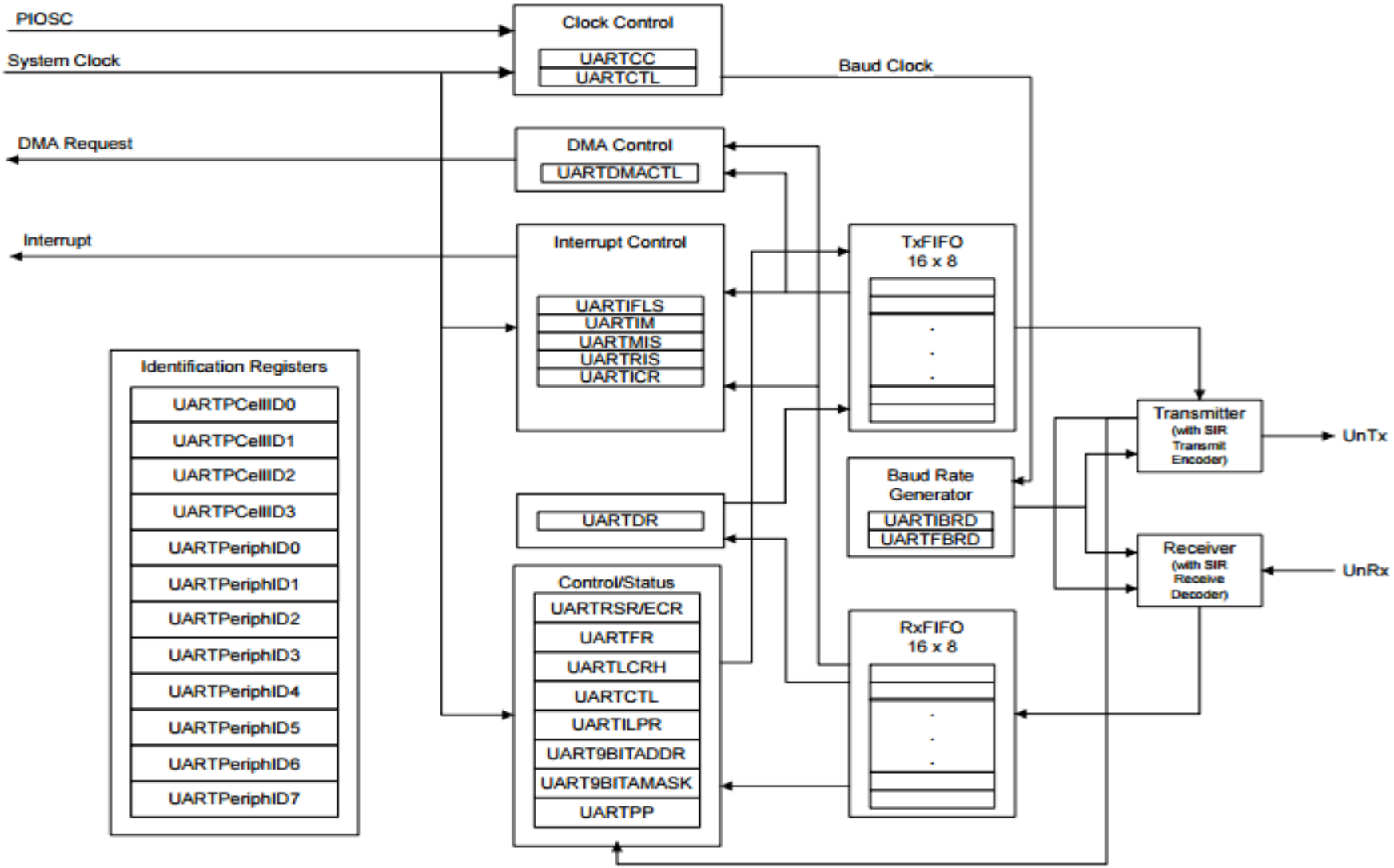
Baud rate, then, is the measure of the number of changes to the signal (per second) that propagate through a transmission medium. The baud rate may **be higher or lower** than the bit rate, which is the number of bits per second that the **user can push through the transmission system**. Bits will be converted into baud for transmission at the sender side and the reverse conversion will happen at the receiver end so that the user receives the bitstream that was sent. A few simple definitions before we move ahead:

Bit rate – **the number of binary 'bits',** 1s or 0s to be transmitted per second
Baud rate – the number of line '**symbols**' transmitted per second
Channels – the number of transmission channels

So to convert bit rate to baud rate you multiple baud rate by the number of bits per symbol by the number of channels being used:

Bit rate = baud rate * bits per symbol (/baud)* Channels

Diagram of UART Module



Programming USART

Both sides of communication should use the same **frame format** and **baud rate**

Frame format:

- Number of **data** bits in the frame: 5, 6, 7, 8 or 9
- Number of stop bits: 1 or 2
- Parity bit: Odd, Even, or None

USART Programming Interface

Control and Status Registers:

UARTCTL, UARTCC, UARTLCRH, UARTFR

- 32-bit registers for control and status checking
- *n is 0 to 7*, e.g. is UART0_CTL_R for USART0
- There are eight USART units

Baud Rate Registers:

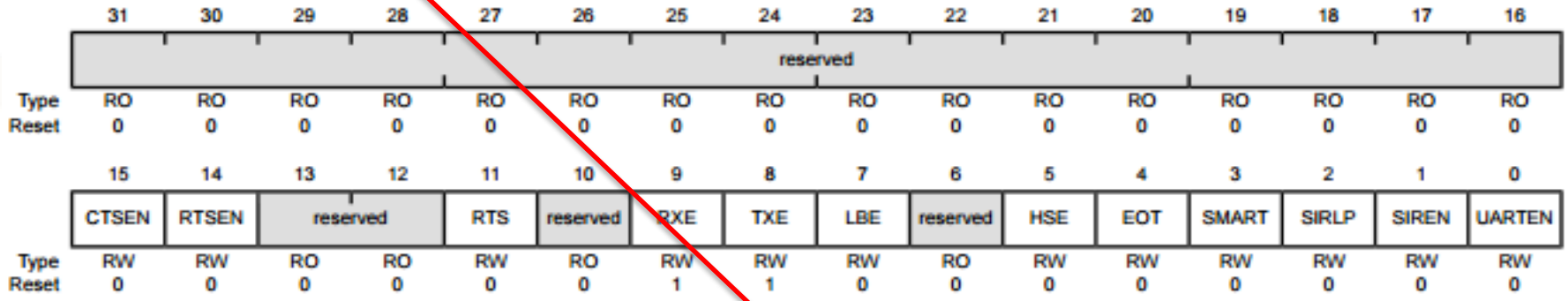
UARTIBRD, UARTFBRD

- Two 32-bit registers used together to set baud rate

32-bit Register for reading and writing data:

UARTDR

UARTCTL: Control Register



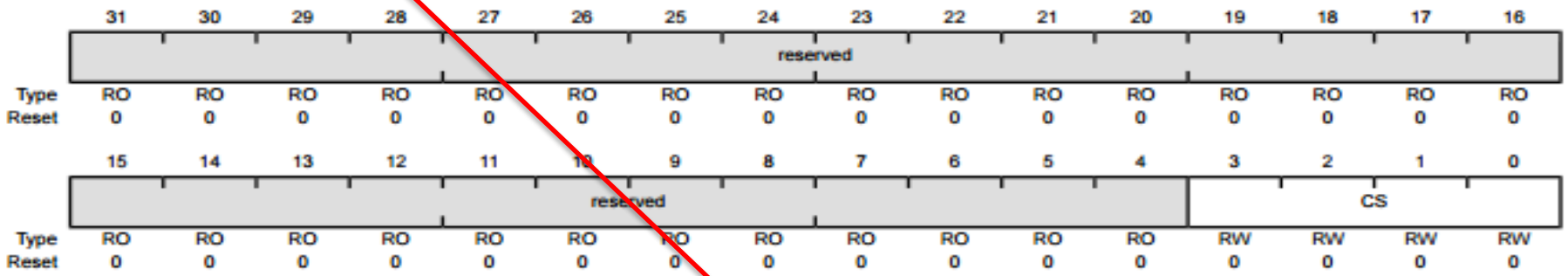
	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
	reserved															
Type	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	CTSEN	RTSEN	reserved		RTS	reserved	RXE	TXE	LBE	reserved	HSE	EOT	SMART	SIRLP	SIREN	UARTEN
Type	RW	RW	RO	RO	RW	RO	RW	RW	RW	RO	RW	RW	RW	RW	RW	RW
Reset	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0

UART operation control

15 CTSEN – Enable clear to send
14 RTSEN – Enable request to send
11 RTS – Request to send
9 RXE – UART receive enable
8 TXE – UART transmit enable
7 LBE – UART loop back enable
5 HSE – High-Speed enable
4 EOT – End of transmission
3 SMART – Smart card support
2 SIRLP – UART SIR low-power mode
1 SIREN – UART SIR enable
0 UARTEN – UART enable
31:16, 13:12, 10, 6 Reserved – Read only

UARTCC: Control Register

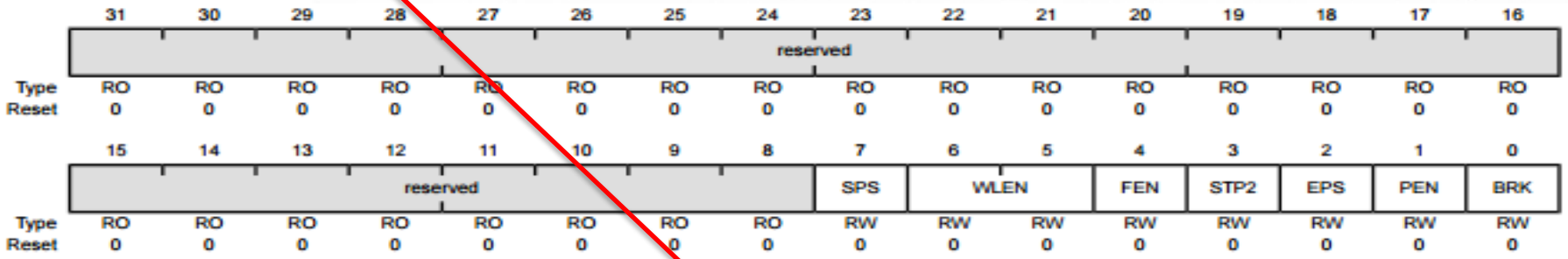


3:0 UART - baud clock source

31:4 Reserved

Clock Configuration

UARTLCRH: Control Register



	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
	reserved															
Type	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

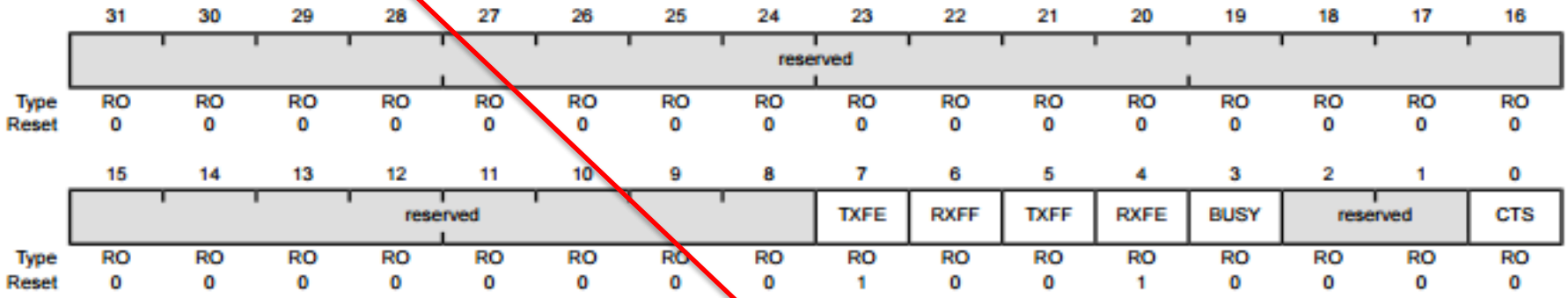
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	reserved								SPS	WLEN		FEN	STP2	EPS	PEN	BRK
Type	RO	RO	RO	RO	RO	RO	RO	RO	RW	RW	RW	RW	RW	RW	RW	RW
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

7 SPS – UART stick parity select
6:5 WLEN – UART word length
4 FEN – UART enable FIFOs
3 STP2 – UART two stop bits select
2 EPS – UART even parity select
1 PEN – UART parity enable
0 BRK – UART send break
31:8 Reserved

LINE Control Register: Set up data framing parameters

Everything else will default to 1 stop bit, no parity, no FIFO

UARTFR: Status Register



	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
	reserved															
Type	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	reserved								TXFE	RXFF	TXFF	RXFE	BUSY	reserved		CTS
Type	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO
Reset	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0	0

7 TXFE – UART transmit FIFO empty

6 RXFF – UART receive FIFO full

5 TXFF – UART transmit FIFO full

4 RXFE – UART receive FIFO empty

3 BUSY – UART busy

0 CTS – Clear to send

31:8, 2:1 Reserved

Flag register-> Provide working status

Serial

- Baud rate
  - 1 *baud* = 1 *symbol per second*
  - In our case, 8 data bits are book ended by start and stop bits

- **Baud rate** is different from **data rate**
  - Baud rate includes overhead of start/stop/parity bits

Calculating Baud

- Two 32 bit registers UARTIBRD and UARTFBRD
- **BRDI = integer portion, BRDF = fractional portion**
- **Baud Rate = UARTSysClk/((BRD) * ClkDiv)**
  - UARTSysClk = 16Mhz
  - ClkDiv = **16 with HSE bit = 0 (8 with HSE bit = 1)**
  - Baud Rate used = 115200
- BRDI = (int)(BRD)
- BRDF = (int)(fraction of BRD)*64 +.5)

Example BRDI and BRDF

- Set a baud rate of 9600 bps for 16Mhz SysClk, HSE = 0
- BRD = 16,000,000 / (16 * 9600) = 104.16666
- BRDI = 104
- BRDF = .1666*64+.5 = 11.16666 = 11

Presentation and Report

Presentation_Contents to be covered

1. Storyline (Optional: AI brain)
2. Abstract Representation (& flows & flow chart)
3. Devices
4. Resource (Software), Protocols?
5. Job roles of each member
6. Gantt Chart
7. Budget estimation (Ceiling: NT 1000 per group)
8. 抬頭: 國立臺灣師範大學; 統編: 03735202
9. Others

Final Report_Contents to be covered

1. Introduction
2. Objectives
3. Procedures of execution of the project
4. Result & Discussion
5. Conclusion
6. Maximum pages: 5 pages (pictures are not included)
7. Title of the project, duration, lab location, your name, and equipment and electronic components used should also be included.

Lucky Draw

Presentation List (Proposal)

Random Generator: https://www.random.org/

Group	Time Slot	Group	Time Slot
Group xx	4/02 14:20- 14:32 (P) 14:32-14:35 (Q/A)	Group xx	4/02 15:45- 15:57 (P) 15:57-16:00 (Q/A)
Group xx	4/02 14:35- 14:47 (P) 14:47-14:50 (Q/A)	Group xx	4/02 16:00- 16:12 (P) 16:12-16:15 (Q/A)
Group xx	4/02 14:50- 15:02 (P) 15:02-15:05 (Q/A)	Group xx	4/02 16:15- 16:27 (P) 16:27-16:30 (Q/A)
Group xx	4/02 15:15- 15:27 (P) 15:27-15:30 (Q/A)	Group xx	4/02 16:30- 16:42 (P) 16:42-16:45 (Q/A)

* 12 mins for presentation; 3 mins for Q n A

Presentation List (Demo)

Random Generator: https://www.random.org/

Group	Time Slot	Group	Time Slot
Group xx	4/09 14:20- 14:32 (P) 14:32-14:35 (Q/A)	Group xx	4/09 15:45- 15:57 (P) 15:57-16:00 (Q/A)
Group xx	4/09 14:35- 14:47 (P) 14:47-14:50 (Q/A)	Group xx	4/09 16:00- 16:12 (P) 16:12-16:15 (Q/A)
Group xx	4/09 14:50- 15:02 (P) 15:02-15:05 (Q/A)	Group xx	4/09 16:15- 16:27 (P) 16:27-16:30 (Q/A)
Group xx	4/09 15:15- 15:27 (P) 15:27-15:30 (Q/A)	Group xx	4/09 16:30- 16:42 (P) 16:42-16:45 (Q/A)

* 20 mins for presentation & demo; 5 mins for Q n A