

# **Microprocessors & Interfacing**

*Review*

Lecturer : Annie Guo

# Lecture Overview

- Course review
- About the final exam

# What have we learned?

- Basics of microprocessor systems
- AVR and AVR programming
- Interfacing
- Microcontroller application development

# Basics of Microprocessors

- Microprocessors are digital systems
  - Can be programmed to perform varied functions on different information.
- Information can be encoded
  - With binary digits
  - For example
    - Numbers in binary code, 2's complement code, ...
    - Characters in ASCII code

# Basics of Microprocessors (cont.)

- Five basic components of hardware computer systems
  - Datapath, control unit, memory, input/output devices
- Microprocessor
  - Datapath + control unit + → on one chip
- Microcontroller
  - Microprocessor + memory + peripherals → on one chip
- Computer application systems
  - Hardware + software
- ISA
  - Interface between hardware and software

# ISA

- Instruction set architecture
  - Things that assembly programmers can use and should know
  - Including
    - Instruction set
    - Supported native data type
    - Registers
      - What kinds of registers are available?
      - How to use them?
    - Memory models
      - How are instructions or data stored?
      - How can memory be accessed?
    - Interrupts
      - What kinds of interrupts are available?
      - How to use them?

# AVR and AVR Programming

- Assembly programming
  - can be constructed with
    - Sequential operations, if-then-else, loops
    - Function implementation
      - Stack and stack frames
      - Functions and function calls
- Data and data structure implementation
  - Constants, variables
  - Integers, characters, arrays, ...
  - Bits
    - Control bits

# Interfacing

- I/O devices
  - Input devices: switch, push button, keypad
  - Output devices: LED bar, LCD
- Basic interaction approaches:
  - Polling and Interrupt
- Basic communication types
  - Parallel and serial input/output
- Handling analog input/outputs
  - ADC, DAC, PWM

# Project Development

- You need to
  - Find out what resources are available
  - How to use available resources
  - Develop strategies and design for the project
    - Hardware and software
  - Test and Debug

# What can we do?

- Not only know what, but how, and more importantly why.
- For the practical problems encountered in the future work or re-learning process, you can
  - identify the basic knowledge needed and find the possible solutions.
- Maintain (existing) microprocessor application systems
  - Problem debugging
  - System enhancement
- Design a microprocessor application system
- Be able to proceed to extended/advanced areas for breath/in-depth study

# Where can we go now?

- Proceed to advanced computing courses
  - Compiler
    - For large software development, automatic machine code generation
  - Computer architecture
    - How to improve the processor and computer hardware system?
  - Operating system
    - How to make good and efficient use of the computer hardware system
  - Networking
    - How to handle complicated communications and make communication efficient?

# Where can we go now? (cont.)

- Proceed to advanced computing courses
  - Digital circuits and systems
    - Design digital circuits and systems, and implement your design on FPGA (field-programmable gate array).
  - Embedded system design
    - How to develop an efficient processor for a specific application
    - How to design and implement an embedded system with microcontrollers, microprocessors and other specific designed hardware components.

# About Final Exam

- Moodle Online
  - Open book exam
- Date: Thur., Dec. 8
- Exam open time: 13:45 (Sydney time)
- Exam close time: 16:00 (Sydney time)
- Duration: 2 hours

# About Final Exam

- Types of questions
  - Concept based
  - Programming based
  - Design based
- Question Format
  - Multiple-choice based questions
    - For each question, multiple answers are allowed. Penalty will be applied for each incorrect answer
      - You need also explain your answer for most questions
    - Programming & design related question
  - For each question, provide your answer in the provided textbox

# About Final Exam (cont.)

- Materials will not be covered in the exam
  - Slides marked with \*
  - Extended topic
    - AVR ADC
- Sample questions will be
  - Provided in Moodle

# Multiple Choice Question (1)

Question 1

Not yet  
answered

Marked out of  
4.00

To fetch a byte from the program memory, which of the following statements is/are true?.

- (a) The register indirect with post-increment addressing mode can be used.
- (b) Address register Z should be used.
- (c) Instruction *ld* can be used.
- (d) The immediate addressing mode can be used.

# Multiple Choice Question (2)

Question 2

Not yet  
answered

Marked out of  
6.00

 Flag  
question

 Edit  
question

Which of the following statements is/are true about the AVR processor? Briefly explain your answer.

- (a) If flag Z in the status register is 0, that means the result of the arithmetic instruction just executed is zero
- (b) Stack pointer can be used to hold a temporary value.
- (c) The program counter PC is not a register.
- (d) Instruction *brne* can be used for both signed and unsigned calculations.

# Multiple Choice Question (3)

Question 3

Not yet  
answered

Marked out of  
6.00

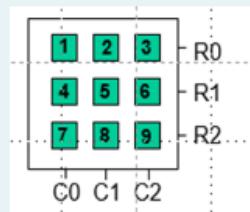
Flag  
question

Edit  
question

The figure below shows a 3x3 keypad. What value can be loaded into DDRA if Port A is to be used to drive the keypad?

Assume that bits 0-2 of Port A connect to Rows R0-R2 of the keypad and Bits 4-6 of Port A connect to Columns C0-C2 of the keypad. Explain your answer.

- (a) 0x71
- (b) 0x78
- (c) 0xF1
- (d) 0xF0



# Multiple Choice Question (4)

Question 4

Not yet  
answered

Marked out of  
4.00

 Flag  
question

 Edit  
question

Which of the following registers can be used to create an alias "zero" for a register?

- (a) .set
- (b) .def
- (c) .dw
- (d) .db

# Programming & Design Question 1

Question 5

Not yet  
answered

Marked out of  
8.00

Flag  
question

Edit  
question

The figure below shows a design, where the transducer is to be used to find the temperature over a range of  $-100^{\circ}\text{C}$  to  $+100^{\circ}\text{C}$ .



The design is required to read and display the temperature to a resolution of  $\pm 1^{\circ}\text{C}$ . The transducer produces a voltage from -5 to +5 volts over this temperature range with 5 millivolts of noise. What resolution (namely, the number of output bits) would you choose for the A/D converter? Explain your answer.

# Programming & Design Question 2

```
/*
 * The function, fib(m) calculates the fibonacci numbers. It
 assumes an integer is one byte.
 * The Parameter and return value are passed through r24.
 */

fib:
    ; prolog
    push r16
    push YL
    push YH
    in YL, SPL
    in YH, SPH
    sbiw Y, 1          ; Let Y point to the bottom of the
    stack frame
    out SPH, YH        ; Update SP so that it points to
    out SPL, YL        ; the new stack top
    std Y+1, r24        ; Pass the actual parameter to the
    formal parameter

    ; function body
    cpi r24, 2          ; Compare m with 2
    brsh L2             ; If m is not 0 or 1
    ldi r24, 1           ; m==0 or 1
    rjmp L1             ; Jump to the epilogue
L2: ldd r24, Y+1       ; m>=2
    subi r24, 1          ; Pass m-1 to the callee
    rcall fib            ; call fib(m-1)
    mov r16, r24          ; Store the return value in r16
    ldd r24, Y+1          ; Load the actual parameter n
    subi r24, 2           ; Pass n-2 to the callee
    rcall fib            ; call fib(m-2)
    add r24, r16          ; r25=fib(m-1)+fib(m-2)

L1:
    ; epilogue
    adiw Y, 1
    out SPH, YH
    out SPL, YL
    pop YH
    pop YL
    pop r16
```

- 7 The figure to the left shows an AVR function for calculating the Fibonacci number.

Based on the code, design to use **software interrupt** to detect the overflow generated by the function. When an overflow is detected, the LED bar will turn on and the function returns 0.

Upload your work for this question, which includes a brief description of your design and the assembly code. Your code should be well commented.

If you have more than one file, zip your files and upload the zip file.

Upload your file here. Maximum one file.  
All file types are allowed. Maximum file size is 1 GB

Maximum marks: 25



# Thank You!

# myExperience (CATEI) Survey

- Please participate
- If you have some feedback not covered by the Survey, please send your comments to me.
- Your feedback is much appreciated and will be considered for future improvement.

# **Microprocessors & Interfacing**

*Serial Input/Output (II)*

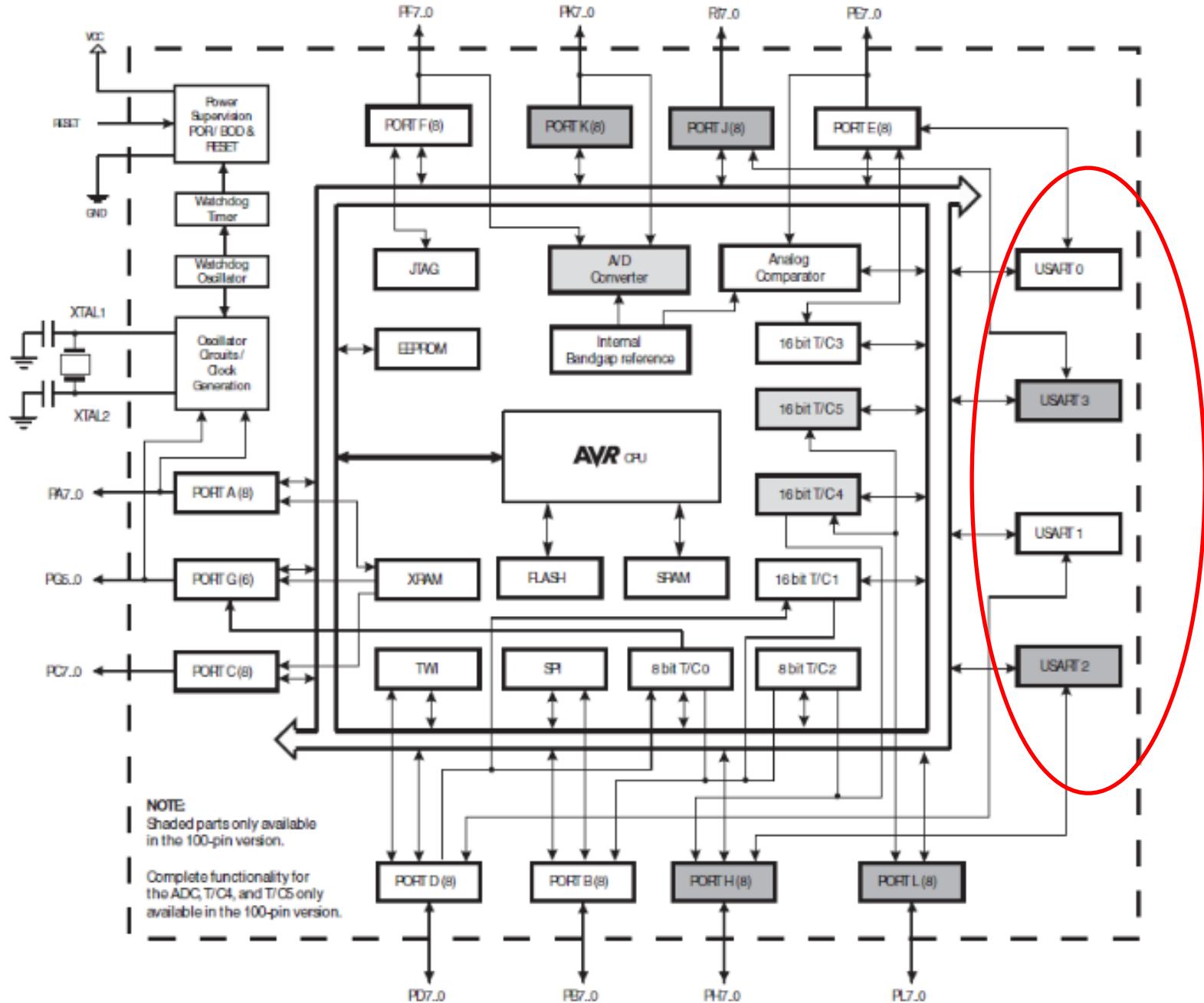
Lecturer : Annie Guo

# Lecture Overview

- USART in AVR

# AVR USART

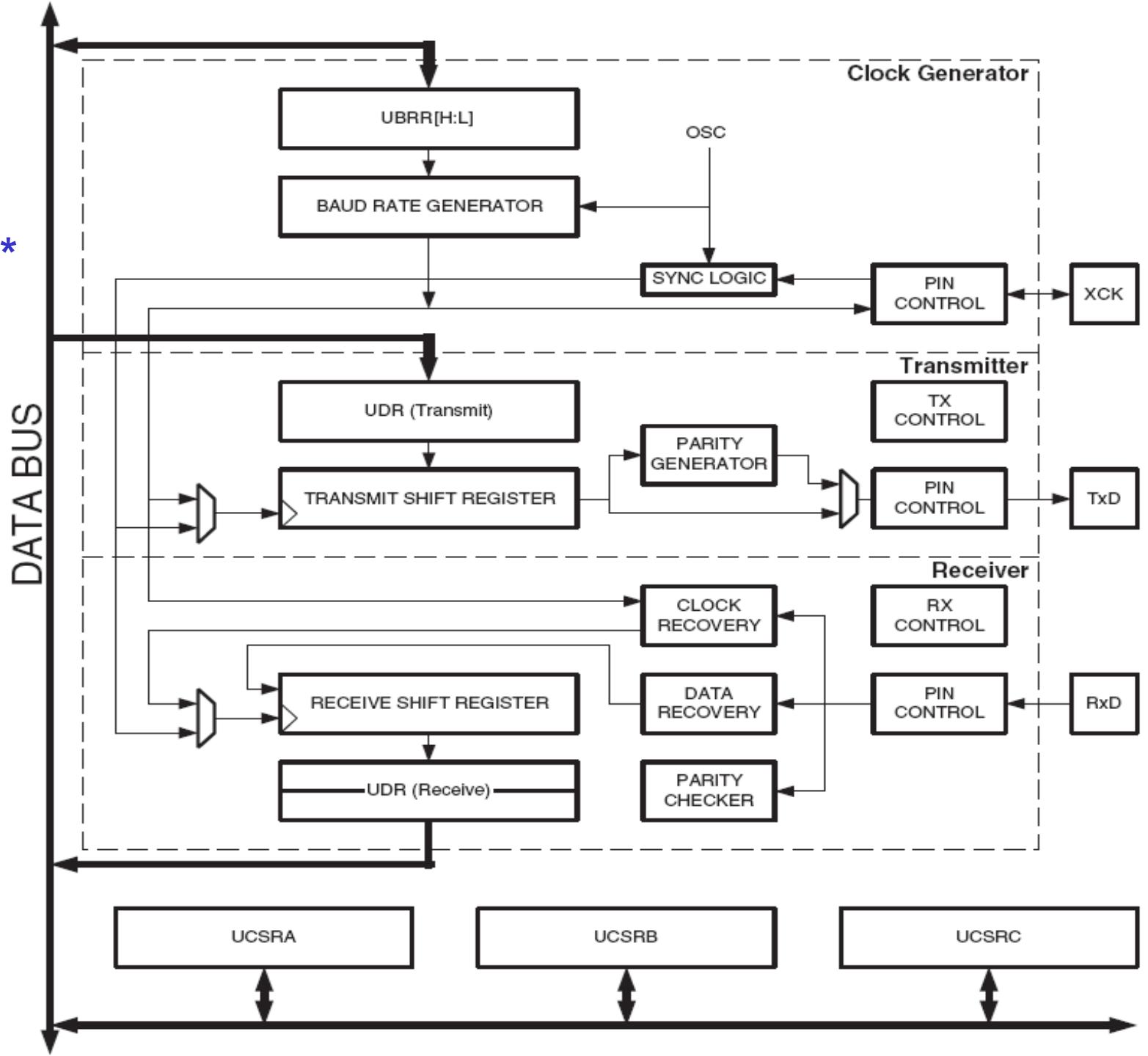
- USART: Universal Synchronous Asynchronous Receiver and Transmitter
- Four USART units
  - Units 0-3
- Each unit can be configured for synchronous or asynchronous serial communication



# AVR USART (cont.)

- Support many frames
- Have transmission error detection function
  - Odd or even parity error
  - Framing error
  - Overrun error
- Three interrupts on
  - TX (Transmit) Complete
  - RX (Receive) Complete
  - TX Data Register Empty

# USART Block Diagram\*



# AVR USART Structure

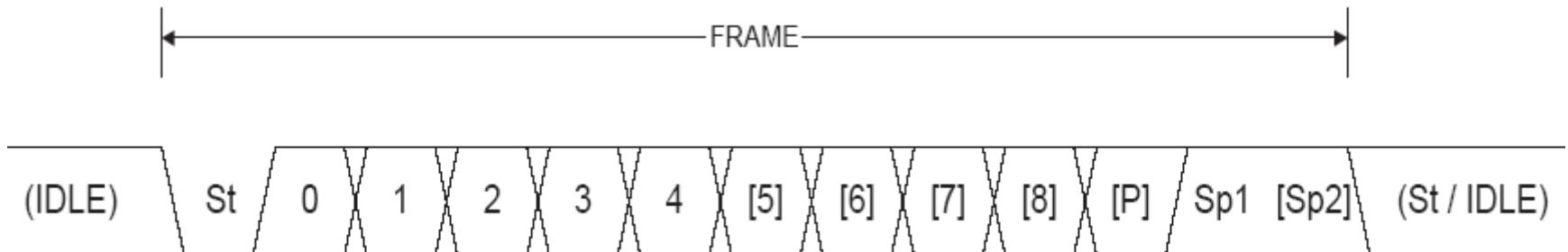
- The USART has three components: clock generator, transmitter and receiver
- Clock generator
  - consists of synchronization logic for external clock input and the baud rate generator
    - Baud rate: the maximum number of bits that can be transferred per second
- Transmitter
  - consists of a single write buffer, a serial Shift Register, Parity Generator and Control Logic for handling different frames.

# AVR USART Structure (cont.)

- Receiver
  - The receiver is the most complex part of the USART module due to its clock and data recovery operations.
  - In addition to the recovery units, the receiver includes Parity Checker, Control Logic, Shift Register and Receive Buffer (UDR).
  - The Receiver supports the same frame formats as the Transmitter, and can detect Framing Error, Data Overrun and Parity Error.

# Frame Format

- Up to 30 different formats available in the USART
  - combinations of
    - 1 start bit (St)
    - 5, 6, 7, 8, or 9 data bits
    - no, even or odd parity bit (P)
    - 1 or 2 stop bits (Sp)



# Parity Bit

- Used to check whether the received data is different from the sending data
  - A very simple way
- Two forms of the parity bit
  - Even parity

$$P_{\text{even}} = d_n \oplus d_{n-1} \oplus \dots \oplus d_1 \oplus d_0 \oplus 0$$

- Odd parity

$$P_{\text{odd}} = d_n \oplus d_{n-1} \oplus \dots \oplus d_1 \oplus d_0 \oplus 1$$

- Where  $d_i$  in the above two formulas is a data bit,  $n$  is the number of data bits.

# Control State Registers

- Three control state registers are used in the USART operation:
  - UCSRA
    - for storing the status flags of USART
    - for controlling transmission speed and use of multiple processors
  - UCSRB
    - for enabling interrupts, transmission operations
    - for setting frame format
    - for bit extension
  - UCSRC
    - For operation configuration

# UCSRA

- USART Control and Status Register A

Bit	7	6	5	4	3	2	1	0
	<b>RXC</b>	<b>TXC</b>	<b>UDRE</b>	<b>FE</b>	<b>DOR</b>	<b>UPE</b>	<b>U2X</b>	<b>MPCM</b>
Read/Write	R	R/W	R	R	R	R	R/W	R/W
Initial Value	0	0	1	0	0	0	0	0

# UCSRA Bit Description

- Bit 7 – RXC: USART Receive Complete
  - Set when the receive buffer is not empty
  - The RXC flag can be used to generate a Receive Complete interrupt
- Bit 6 – TXC: USART Transmit Complete
  - Set when the entire frame in the Transmit Shift Register has been shifted out and there are no new data present in the transmit buffer
  - TXC can generate a Transmit Complete interrupt
  - TXC is automatically cleared when a transmit complete interrupt is executed.

# UCSRA Bit Description (cont.)

- Bit 5 – UDRE: USART Data Register Empty
  - Set when the transmit buffer (UDR) is empty
  - Can be used to generate a Data Register Empty interrupt
- Bit 4 – FE: Frame Error
  - Set when the character in the receive buffer was transferred in a wrong frame.
- Bit 3 – DOR: Data OverRun
  - Set when a Data OverRun condition is detected.
  - A Data OverRun occurs when the receive buffer is full and a new start bit is detected.

# UCSRA Bit Description (cont.)

- Bit 2 – UPE: USART Parity Error
  - Set when the character in the receive buffer had a Parity Error when received and the Parity Checking was enabled
- Bit 1 – U2X: Double the USART Transmission Speed
  - Set for doubling the transfer rate for asynchronous communication
- Bit 0 – MPCM: Multi-processor Communication Mode
  - If set, all the incoming frames received by the USART Receiver that do not contain address information will be ignored.

# UCSRB

- USART Control and Status Register B

Bit	7	6	5	4	3	2	1	0
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R	R/W
Initial Value	0	0	0	0	0	0	0	0

# UCSRB Bit Description

- Bit 7 – RXCIE: RX Complete Interrupt Enable
  - Set to enable interrupt on the RXC flag
- Bit 6 – TXCIE: TX Complete Interrupt Enable
  - Set to enable interrupt on the TXC flag
- Bit 5 – UDRIE: USART Data Register Empty Interrupt Enable
  - Set to enable interrupt on the UDRE flag.

# UCSRB Bit Description (cont.)

- Bit 4 – RXEN: Receiver Enable
  - Set to enable the USART receiver.
  - The Receiver will override normal port operation for the RxD pin when enabled. Disabling the Receiver will flush the receive buffer invalidating the FE, DOR and UPE flags.
- Bit 3 – TXEN: Transmitter Enable
  - Set to enable the USART Transmitter
  - The Transmitter will override normal port operations for the TxD pin when enabled. Disabling the Transmitter will not become effective until transmissions are complete.

# UCSRB Bit Description (cont.)

- Bit 2 – UCSZ2: Character Size
  - The bit combined with the UCSZ1:0 bits in UCSRC sets the number of data bits in a frame.
- Bit 1 – RXB8: Receive Data Bit 8
  - The ninth data bit of the received character when operating with serial frames with 9-bit data. Must be read before reading the low bits from UDR
- Bit 0 – TXB8: Transmit Data Bit 8
  - The ninth data bit in the character to be transmitted when operating with serial frames with 9-bit data. Must be written before writing the low bits to UDR

# UCSRC

- USART Control and Status Register C

Bit	7	6	5	4	3	2	1	0
	-	UMSEL	UPM1	UPM0	USBS	UCSZ1	UCSZ0	UCPOL
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Initial Value	0	0	0	0	0	1	1	0

# UCSRC Bit Description

- Bit 6 – UMSEL: USART Mode Select
  - 0: Asynchronous Operation
  - 1: Synchronous Operation
- Bit 5:4 – UPM1:0: Parity Mode
  - Set to enable Parity bit operation

## UPM1 UPM0 Parity Mode

0	0	Disabled
0	1	Reserved
1	0	Enabled, Even Parity
1	1	Enabled, Odd Parity

# UCSRC Bit Description (cont.)

- Bit 3 – USBS: Stop Bit Select
  - 0: 1-bit
  - 1: 2-bit
- Bit 2:1 – UCSZ1:0: Character Size
  - Together with UCSZ2 to determine the number of bits for a character

UCSZ2	UCSZ1	UCSZ0	Character Size
0	0	0	5-bit
0	0	1	6-bit
0	1	0	7-bit
0	1	1	8-bit
1	0	0	Reserved
1	0	1	Reserved
1	1	0	Reserved
1	1	1	9-bit

# UCSRC Bit Description (cont.)

- Bit 0 – UCPOL: Clock Polarity

UCPOL Sampled	Transmitted Data Changed (Output of TxD Pin)	Received Data (Input on RxD Pin)
0	Rising XCK Edge	Failing XCK Edge
1	Failing XCK Edge	Rising XCK Edge

# USART Initialization

- Initialization process consists of
  - Setting the baud rate,
  - Setting the frame format and
  - Enabling the Transmitter and/or the Receiver
- For interrupt driven USART operations, the Global Interrupt Flag should be cleared when doing the initialization

# Baud Rate

**Table 22-9.** Examples of UBRRn Settings for Commonly Used Oscillator Frequencies

Baud Rate (bps)	$f_{osc} = 1.0000\text{MHz}$				$f_{osc} = 1.8432\text{MHz}$				$f_{osc} = 2.0000\text{MHz}$			
	U2Xn = 0		U2Xn = 1		U2Xn = 0		U2Xn = 1		U2Xn = 0		U2Xn = 1	
	UBRR	Error	UBRR	Error	UBRR	Error	UBRR	Error	UBRR	Error	UBRR	Error
2400	25	0.2%	51	0.2%	47	0.0%	95	0.0%	51	0.2%	103	0.2%
4800	12	0.2%	25	0.2%	23	0.0%	47	0.0%	25	0.2%	51	0.2%
9600	6	-7.0%	12	0.2%	11	0.0%	23	0.0%	12	0.2%	25	0.2%
14.4K	3	8.5%	8	-3.5%	7	0.0%	15	0.0%	8	-3.5%	16	2.1%
19.2K	2	8.5%	6	-7.0%	5	0.0%	11	0.0%	6	-7.0%	12	0.2%
28.8K	1	8.5%	3	8.5%	3	0.0%	7	0.0%	3	8.5%	8	-3.5%
38.4K	1	-18.6%	2	8.5%	2	0.0%	5	0.0%	2	8.5%	6	-7.0%

Baud Rate (bps)	$f_{osc} = 16.0000\text{MHz}$				$f_{osc} = 18.4320\text{MHz}$				$f_{osc} = 20.0000\text{MHz}$			
	U2Xn = 0		U2Xn = 1		U2Xn = 0		U2Xn = 1		U2Xn = 0		U2Xn = 1	
	UBRR	Error	UBRR	Error	UBRR	Error	UBRR	Error	UBRR	Error	UBRR	Error
2400	416	-0.1%	832	0.0%	479	0.0%	959	0.0%	520	0.0%	1041	0.0%
4800	207	0.2%	416	-0.1%	239	0.0%	479	0.0%	259	0.2%	520	0.0%
9600	103	0.2%	207	0.2%	119	0.0%	239	0.0%	129	0.2%	259	0.2%
14.4K	68	0.6%	138	-0.1%	79	0.0%	159	0.0%	86	-0.2%	173	-0.2%
19.2K	51	0.2%	103	0.2%	59	0.0%	119	0.0%	64	0.2%	129	0.2%
28.8K	34	-0.8%	68	0.6%	39	0.0%	79	0.0%	42	0.9%	86	-0.2%
38.4K	25	0.2%	51	0.2%	29	0.0%	59	0.0%	32	-1.4%	64	0.2%
57.6K	16	2.1%	34	-0.8%	19	0.0%	39	0.0%	21	-1.4%	42	0.9%
76.8K	12	0.2%	25	0.2%	14	0.0%	29	0.0%	15	1.7%	32	-1.4%
115.2K	8	-3.5%	16	2.1%	9	0.0%	19	0.0%	10	-1.4%	21	-1.4%
230.4K	3	8.5%	8	-3.5%	4	0.0%	9	0.0%	4	8.5%	10	-1.4%
250K	3	0.0%	7	0.0%	4	-7.8%	8	2.4%	4	0.0%	9	0.0%
0.5M	1	0.0%	3	0.0%	—	—	4	-7.8%	—	—	4	0.0%
1M	0	0.0%	1	0.0%	—	—	—	—	—	—	—	—
Max. <sup>(1)</sup>	1Mbps		2Mbps		1.152Mbps		2.304Mbps		1.25Mbps		2.5Mbps	

# Sample Code

- Initialize USART 1

```
.macro: USART_Init:  
    ; Set baud rate, which is stored in r17:r16  
    sts UBRR1H, r17  
    sts UBRR1L, r16  
  
    ; Enable receiver and transmitter  
    ldi r16, (1<<RXEN1)|(1<<TXEN1)  
    sts UCSR1B,r16  
  
    ; Set frame format: 8 bit data, 2 stop bits  
    ldi r16, (1<<USBS1)|(3<<UCSZ10)  
    sts UCSR1C,r16  
.endmacro
```

# Data Transmission

- The USART Transmitter is enabled by setting the *Transmit Enable* (TXEN) bit in the UCSRB Register.
  - A data transmission is initiated by loading the transmit buffer with the data to be transmitted.
    - The CPU can load the transmit buffer by writing to the UDR I/O location. The buffered data in the transmit buffer will be moved to the Shift Register when the Shift Register is ready to send a new frame.
      - The Shift Register is loaded with new data if it is in idle state (no ongoing transmission) or immediately after the last stop bit of the previous frame is transmitted. When the Shift Register is loaded with new data, it will transfer one complete frame at the rate given by the baud register, U2X bit or by XCK depending on mode of operation.

# Sample Code

- Data Transmission
  - The code below uses polling of the *Data Register Empty* (UDRE) flag.
    - When using frames with less than eight bits, the most significant bits written to the UDR are ignored.

```
.macro USART_Transmit

    ; Wait for empty transmit buffer
wait: lds r15, UCSR1A
      sbrs r15,UDRE1
      rjmp wait

    ; Put data (r16) into buffer, sends the data
      sts UDR1,r16

.endmacro
```

# Status of Data Transmission

- The USART Transmitter has two flags that indicate its state:
  - USART Data Register Empty (UDRE)
    - Set: when the transmit buffer is empty and ready to receive new data
    - Clear: when the transmit buffer contains data to be moved into the shift register
  - Transmit Complete (TXC)
    - Set: when data in the Transmit Shift Register has been shifted out and there are no new data currently present in the transmit buffer
    - Clear: otherwise
- Both flags can be used for generating interrupts.

# Data Reception

- The USART Receiver is enabled by writing the Receive Enable (RXEN) bit in the UCSRB Register
  - The baud rate, mode of operation and frame format must be set up before doing any transmissions. If synchronous operation is used, the clock on the XCK pin will be overridden and used as transmission clock.

# Sample code

- Data reception

```
.macro USART_Receive:  
    ; Wait for data to be received  
    wait: lds    r10, UCSR1A  
          sbrs   r10, RXC1  
          rjmp   wait  
  
    ; Get and return received data from buffer  
    lds    r16, UDR1  
.endmacro
```

# Status of Data Reception

- The Receive Complete (RXC) flag indicates if there are unread data present in the receive buffer.
  - Set: when the unread data exists in the receive buffer
  - Clear: otherwise
- If the receiver is disabled ( $RXEN = 0$ ), the receive buffer will be flushed and consequently the RXC bit will become zero.

# Error Detection

- Three errors are checked on the Receiver side:
  - Frame error
    - By checking whether the first stop bit is correctly received
  - Parity error
    - By checking whether the data received has the same (odd or even) number of 1's as in the data from the transmitter.
  - Data OverRun error
    - By checking if any data is yet read but is overwritten by incoming data frame.

# Error Recovery

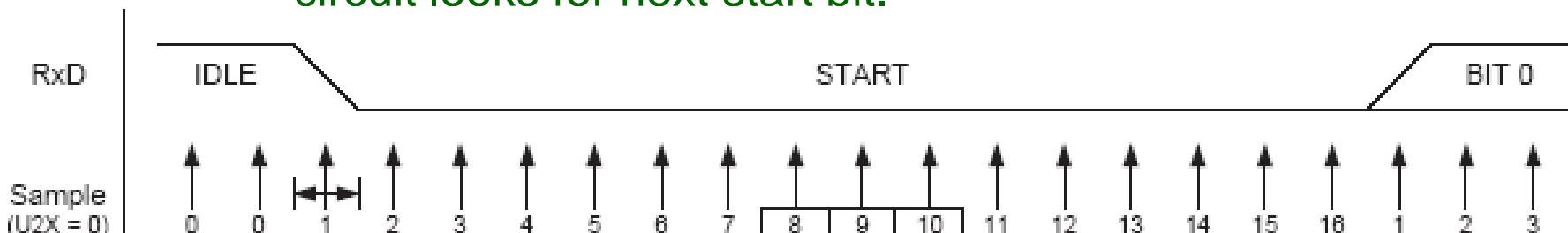
- Main sources of errors in asynchronous data transmission
  - Data reception is “out of sync” with transmission
  - Noise added to the data

# Error Recovery (cont.)

- AVR includes a clock recovery and a data recovery unit for handling errors in asynchronous transmission.
  - The clock recovery logic is used for synchronizing the internally generated baud rate clock to the incoming asynchronous serial frames at the RxD pin.
    - Based on the start bit
  - The data recovery logic samples and (low-pass) filters each incoming bit, thereby improving the noise immunity of the Receiver
  - Based on multiple sampling and majority policy for each incoming bit

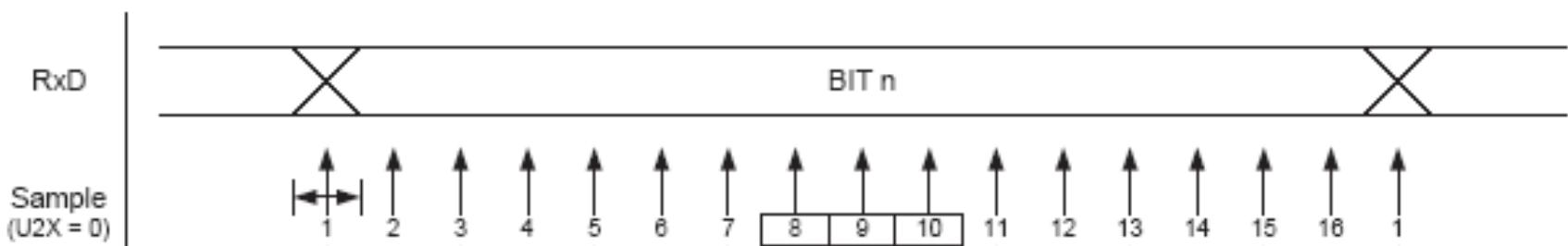
# Error Recovery (cont.)

- The following figure gives an illustration
  - The sample rate is 16 times the baud rate
  - When the Clock Recovery logic detects a high (idle) to low (start) transition on the RxD (receiver data) line, it uses samples 8, 9 and 10 to decide if it is a valid bit (namely, 0)
    - If the majority of the three bits are 0, the bit is valid; data recovery is followed.
    - If the majority of the three bits are 1, the bit is invalid; the circuit looks for next start bit.



# Error Recovery (cont.)

- When the receiver clock is synchronized to the start bit, the data recovery can begin for each subsequent data bit.
- Similarly, the decision of the logic level of each received bit is taken by doing a majority voting of the logic value to the three samples in the center of the received bit.



# Reading Material

- Mega2560 Data Sheet
  - USART

# **Microprocessors & Interfacing**

*Serial Input/Output (I)*

Lecturer : Annie Guo

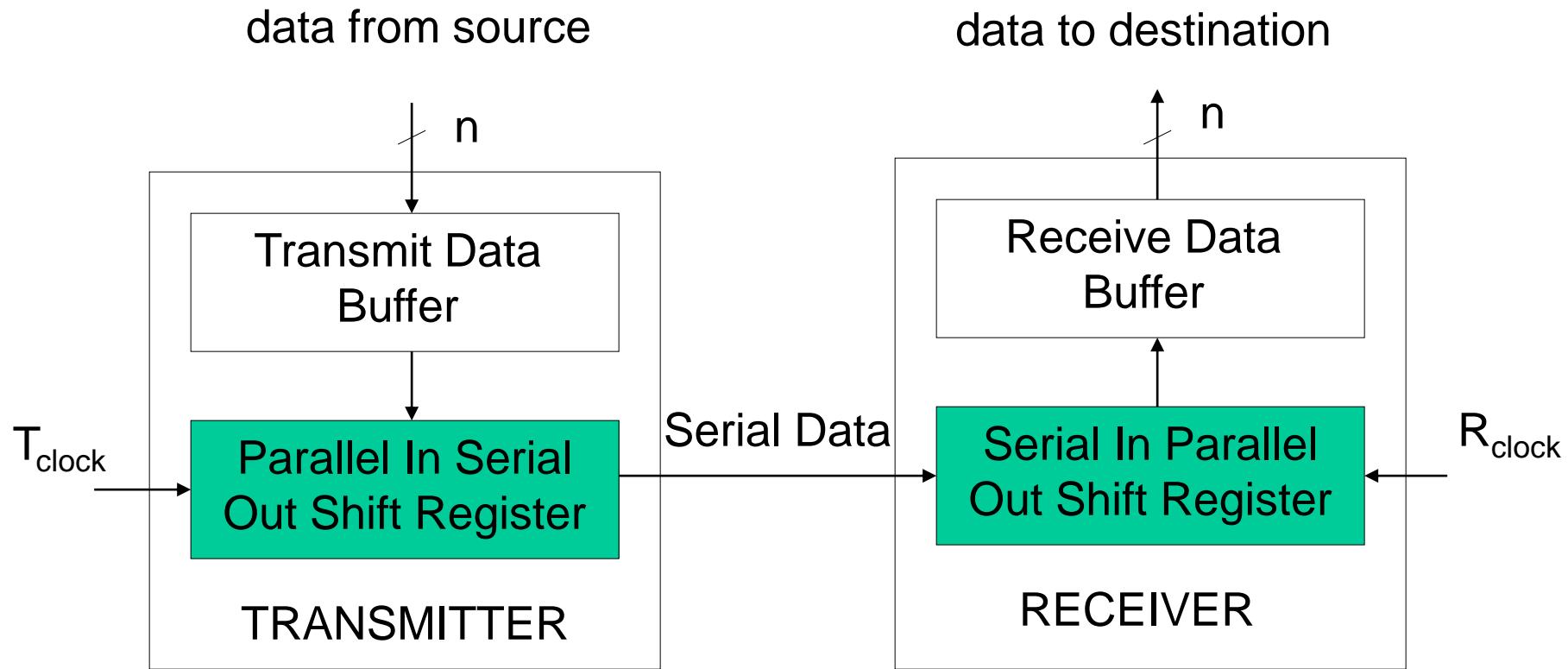
# Lecture Overview

- Serial Communication
  - Concepts
  - Standards\*

# Why Serial I/O?

- Problems with Parallel I/O:
  - Need one wire for each bit.
  - When the source and destination are far from each other, the parallel cable can be bulky and expensive.
  - Susceptible to reflections and induced noises for long distance communication.
- Serial I/O overcomes these problems.

# Serial Communication System Structure



# Serial Communication System Structure (cont.)

- At the communication source:
  - The parallel interface transfers data to the transmit data buffer.
  - The data is loaded into the Parallel In Serial Out (PISO) register and  $T_{clock}$  shifts the data bits out from the register to the receiver.

# Serial Communication System Structure (cont.)

- At the communication destination:
  - $R_{clock}$  shifts each bit received into the Serial In Parallel Out (SIPO) register.
  - After all data bits have been shifted in, they are transferred to the receive data buffer.
  - The data in the receive data buffer can be read by an input operation via the parallel interface.

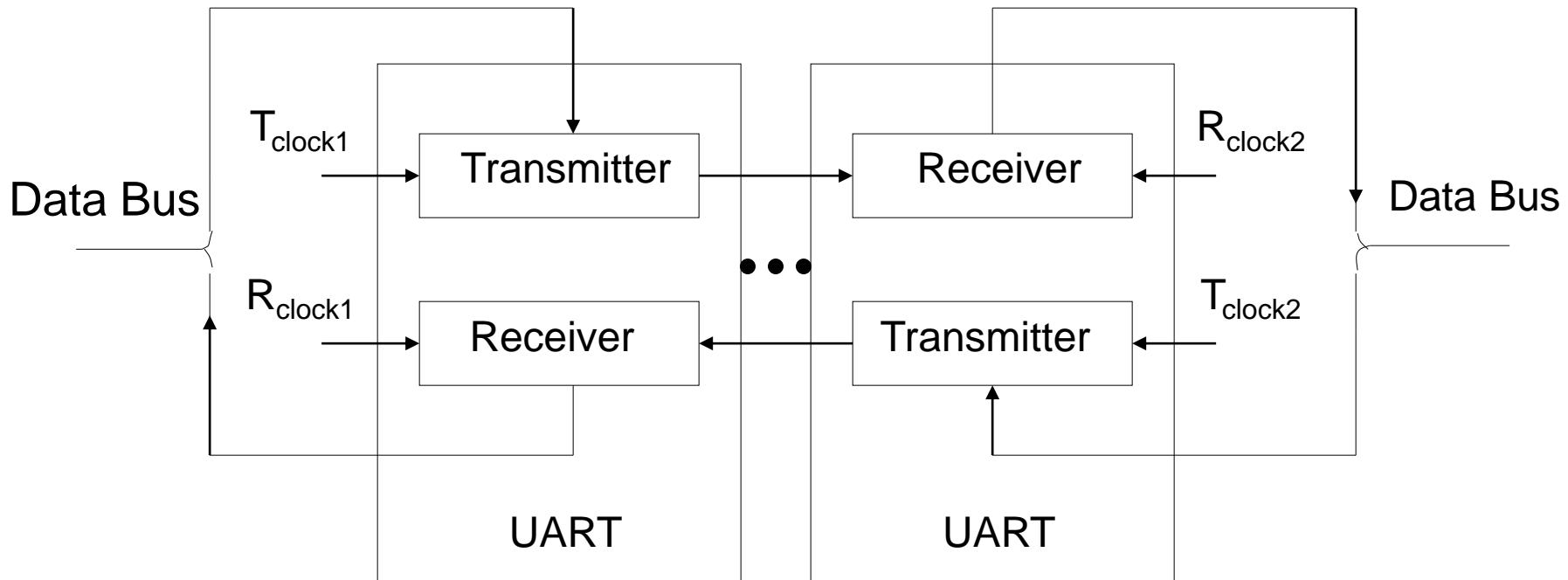
# Synchronous vs Asynchronous Transmission

- Synchronous
  - Transmitter and receiver clocks are synchronized
    - Need extra hardware for clock synchronization
  - Have faster data transfer rate
- Asynchronous
  - Transmitter and receiver use different clocks. No clock synchronization is required.
  - Used in many applications, such as keyboard, mouse and modem.
  - The rest of this lecture mainly focuses on Asynchronous communication

# UART

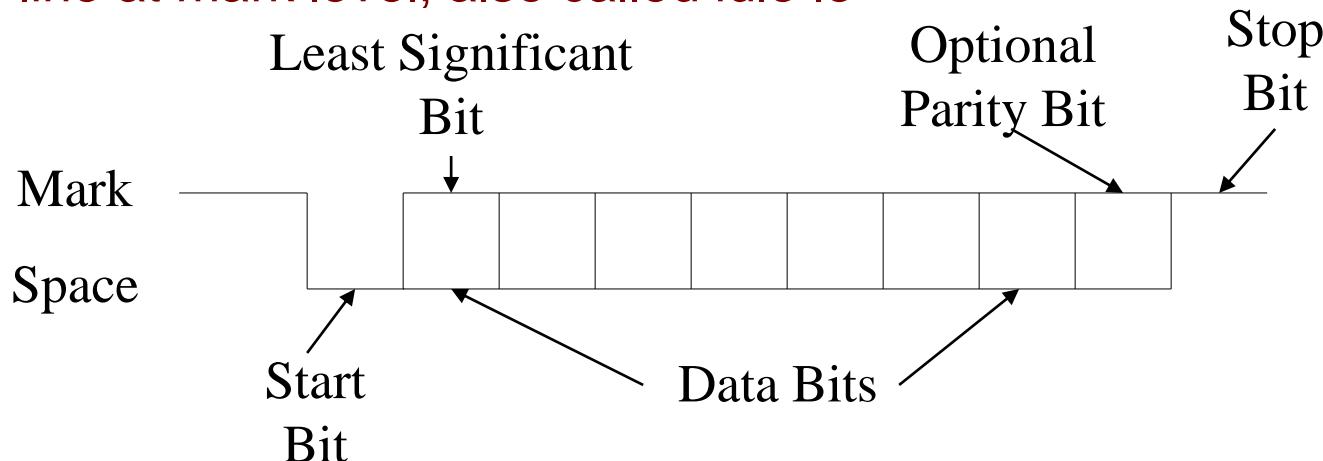
- UART is a basic serial communication hardware
- It has both transmitter and receiver
- Data are transmitted asynchronously
  - Clocks on both sides are not synchronized
  - But the receiver has a way to synchronize the data receiving operation with the data transmitting operation
- Hence called UART (Universal Asynchronous Receiver/Transmitter).

# Communication Logic Structure with UART



# UART Data Frame Format

- Before transmission, data should be encoded
  - Use an encoding scheme, such as ASCII
- Each encoded data item is encapsulated with two types of bits
  - Start bit and stop bit
- **Mark and space:** the logic one and zero levels are, respectively, called mark and space.
  - When the transmitter is not sending anything, it holds the line at mark level, also called idle level.



# UART Data Frame Format (cont.)

- Typical bits in data transmission:
  - Start bit:
    - When the transmitter has data to send, it first changes the line from the mark to the space level for one-bit time. This is to synchronise the receiver with the transmitter. When the receiver detects the start bit, it starts to clock in the serial data bits.
  - Data bits:
    - representing a data item, such as a character

# UART Data Frame Format (cont.)

- Typical bits in data transmission:
  - Parity bit: used to detect errors in the data
    - For odd parity: this bit appended to the data to make the total number of 1s in the data odd
    - For even parity: this bit appended to the data to make the total number of 1s in the data even.
    - E.g. ‘9’ → ASCII: 0x39=0b00111001  
 $P_{\text{even}} = 0$   
 $P_{\text{odd}} = 1$
  - Stop bit: added at the end of data frame.
    - It separates successive data transmissions.
    - Some systems require more than one stop bit.

# Data Transmission Rate

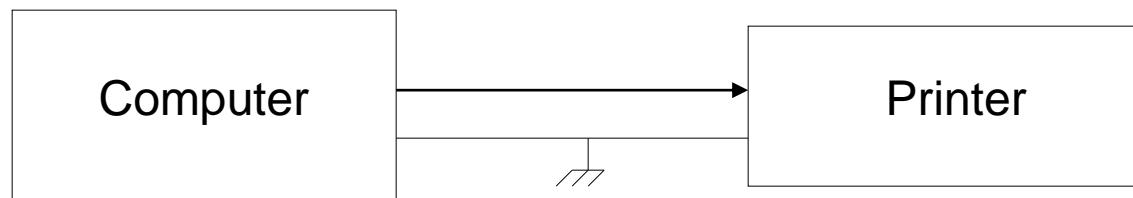
- The rate at which bits are transmitted, also called ***baud rate***, measured in *bits per second*.

# Communication Connection Types

- Three serial communication connection types:
  - Simplex
  - Full-duplex (FDX)
  - Half-duplex (HDX)

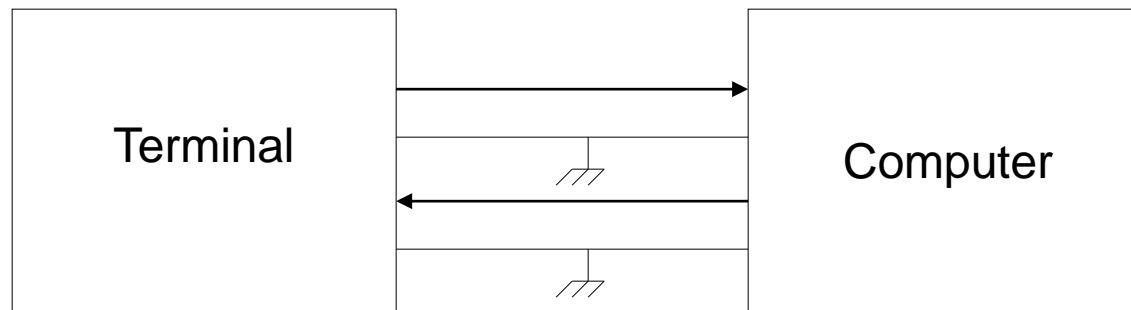
# Simplex Connection

- Data are sent in one direction only
  - For example, computer to a serial printer.
- Simple
  - If the sender does not send data faster than the receiver can accept it, no handshaking signals are required.



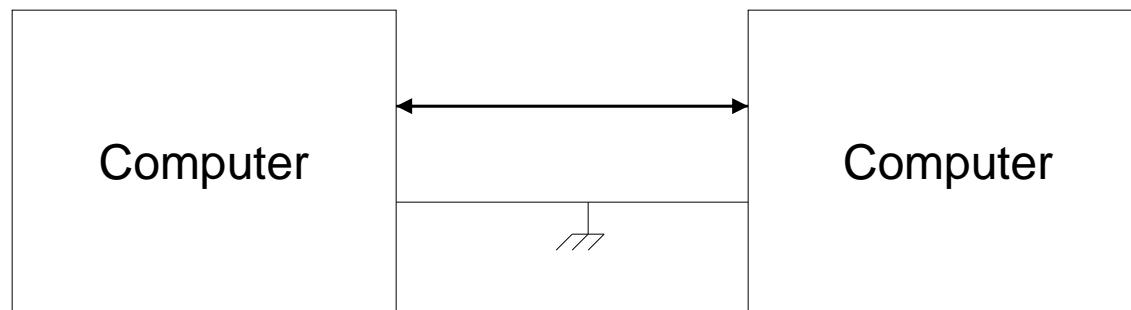
# Full-Duplex (FDX) Connection

- Data are transmitted in two directions, each with a separate data line.



# Half-Duplex (HDX) Connection

- Data are transmitted in two directions with only one data line.

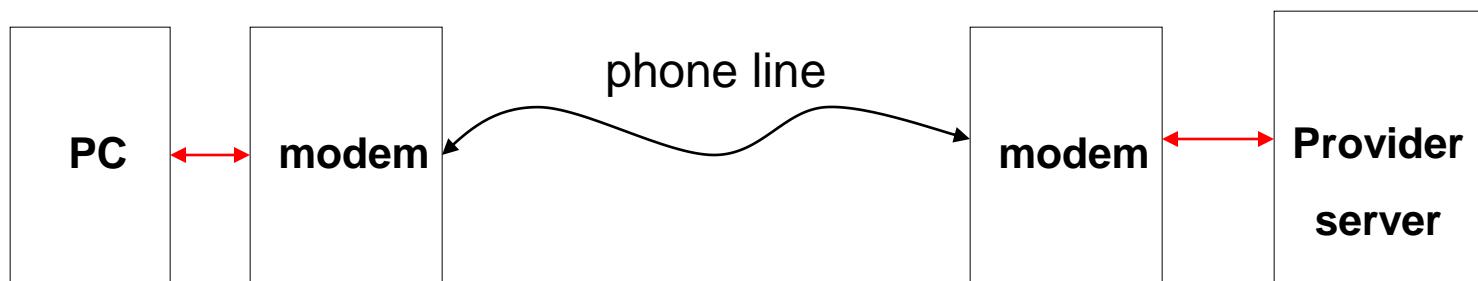


# DTE & DCE\*

- There are two typical devices in Serial communications
- DCE
  - Data Communications Equipment
    - E.g. Modem
- DTE
  - Data Terminal Equipment
    - End on the communication path

# DTE & DCE – example\*

- A communication system for internet access
  - PC and Internet provider server are DTEs
  - Two modems are DCEs



# Modem\*

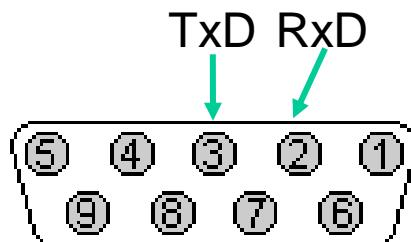
- A modulator/demodulator
  - It converts logic levels into tones to be sent over a telephone line.
  - At the other end of the telephone line, a demodulator converts the tones back to logic levels.

# Standards for Serial I/O Interface

- Interface standards are needed to allow different manufacturers' equipment to be interconnected
  - E.g. modem to computer
- Must define the following elements:
  - Handshaking signals
  - Direction of data flow
  - Types of communication devices
  - Physical interface
  - Electrical signal levels

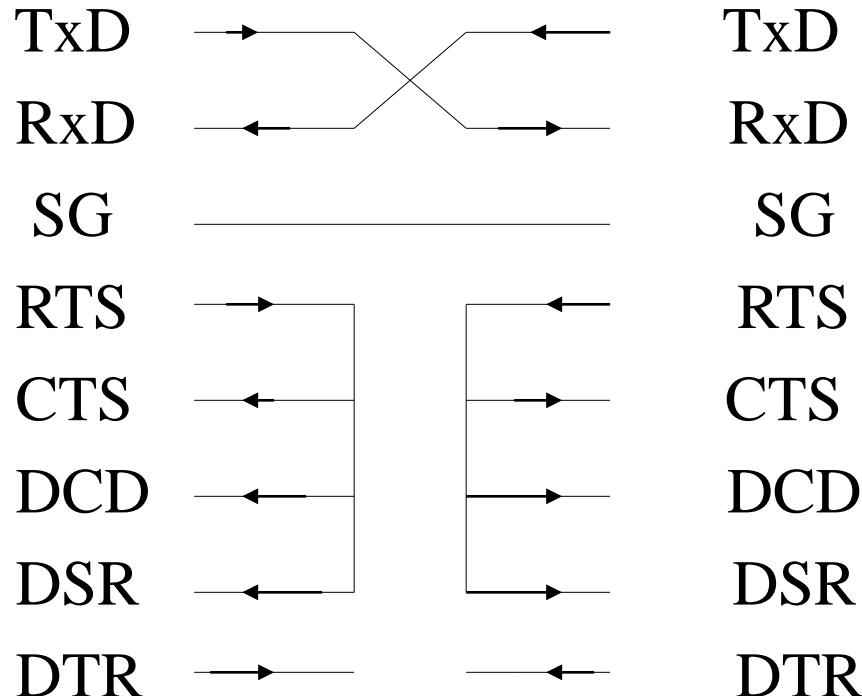
# Standards for Serial I/O Interface (cont.)

- Typical standards include RS-232-C, RS-422, RS-423 and RS-485.
  - RS-232-C standard is used in most serial interface.
    - Standard interconnection cable should be used.
      - E.g. 9-pins
    - Can be in different connection types

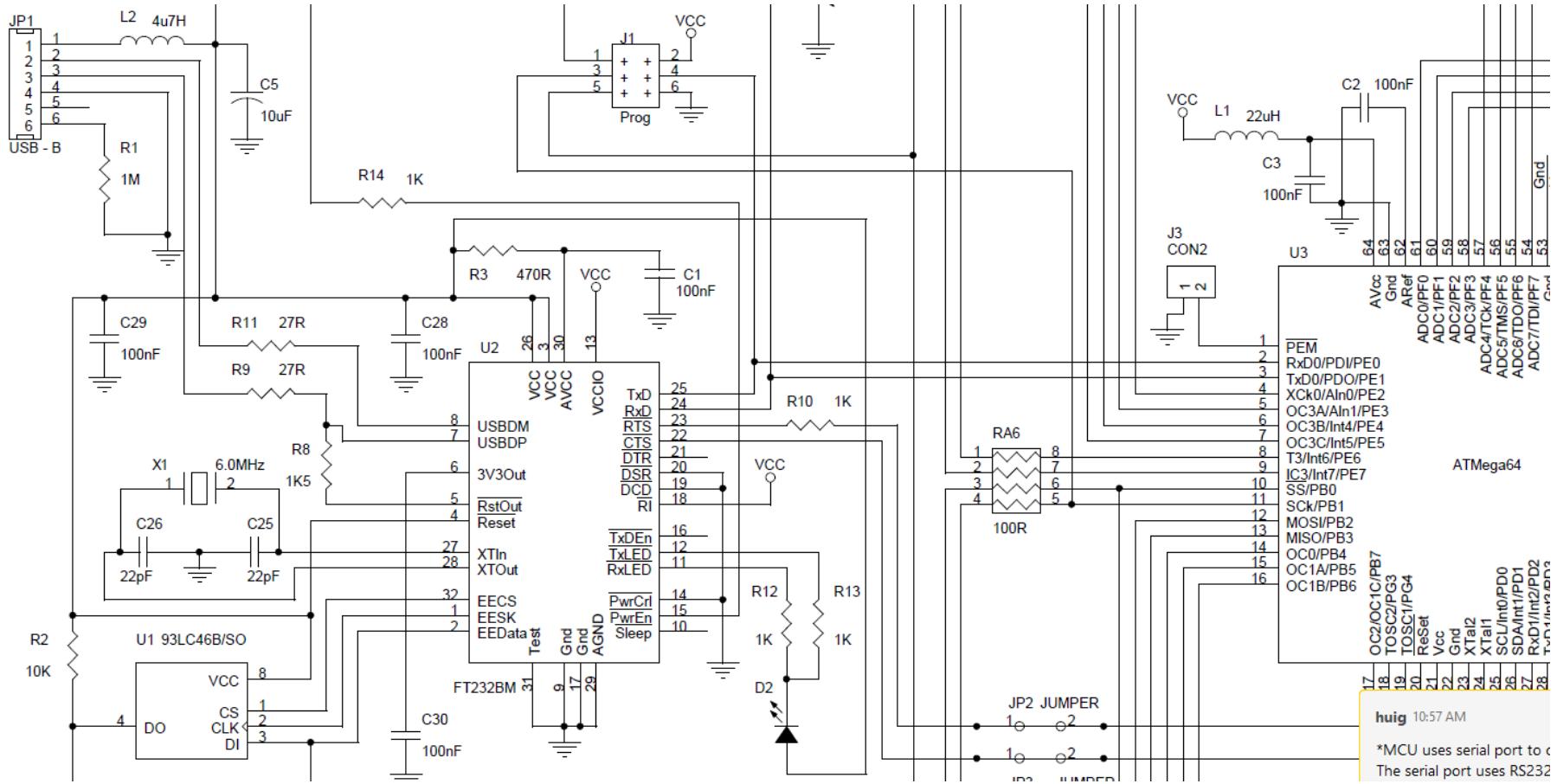


# RS-232-C Interconnection\*

- Example: minimal null modem cable
  - Used in our lab board



# Example of UART to USB Connection\*



# Reading Material

- Chapter 12: Serial Input/Output.  
Microcontrollers and Microcomputers by  
Fredrick M. Cady.

# **Microprocessors & Interfacing**

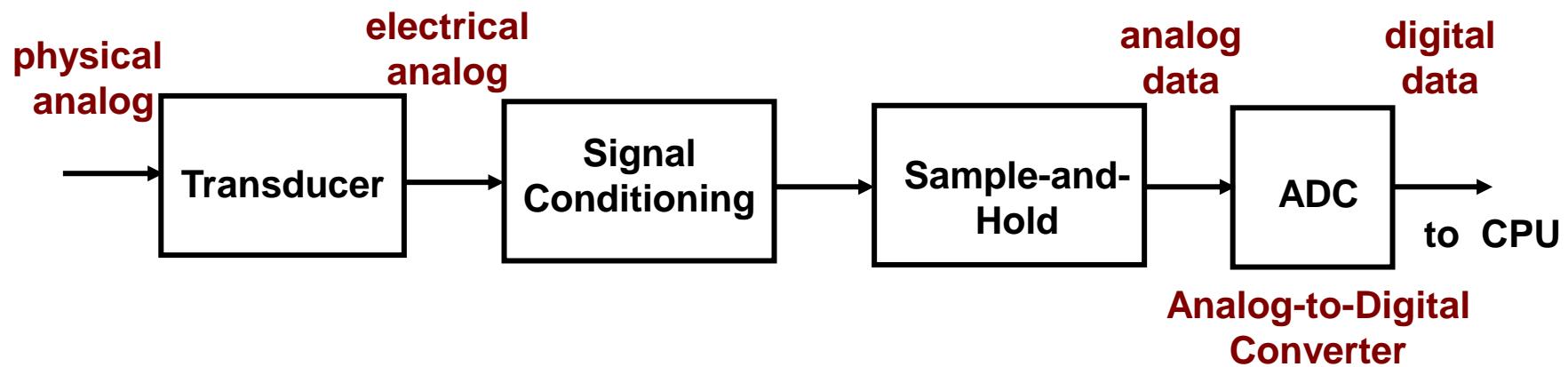
*Analog Input/Output (III)*

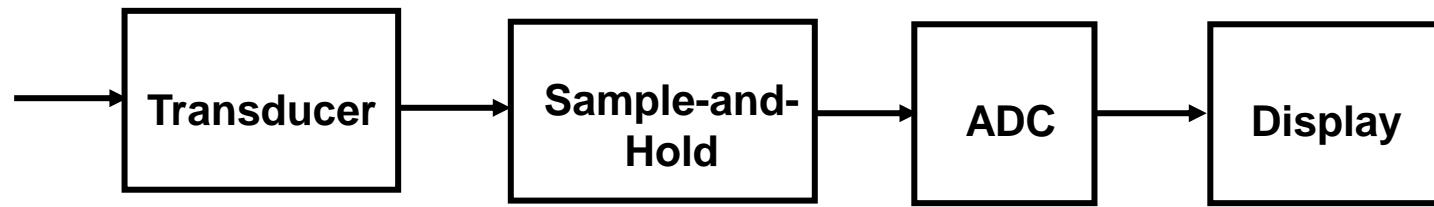
Lecturer : Annie Guo

# Lecture Overview

- Analog input
  - Analog-to-Digital (A/D) Conversion
  - AVR ADC\*
    - extended topic

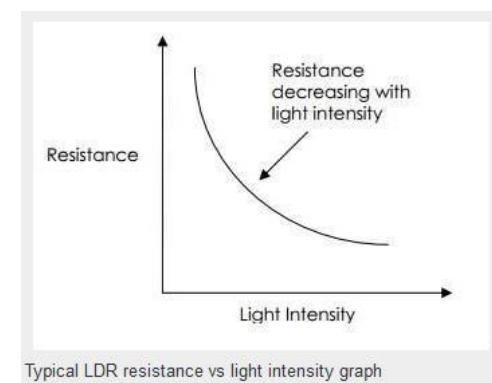
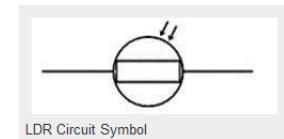
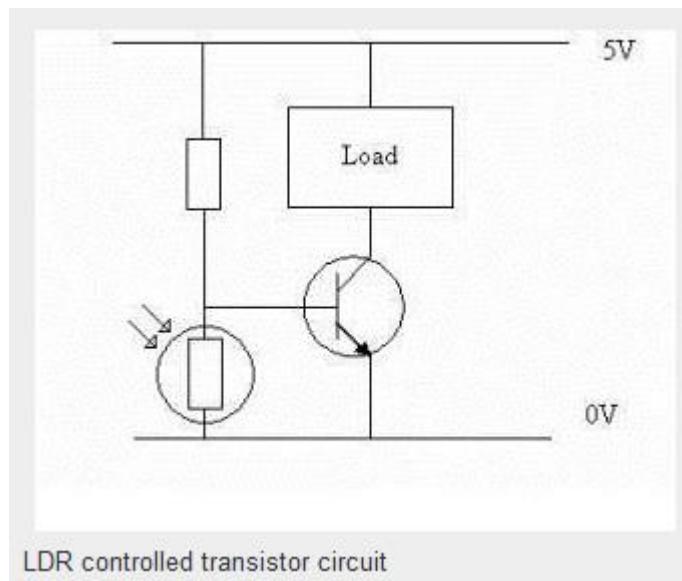
# A/D Conversion





# Sensor\*

- LDR (Light Dependent Resistor)



# Data Acquisition and Conversion

- A transducer converts physical values to electrical signals, either voltages or currents.
- Signal conditioner performs the following tasks:
  - Isolation and buffering
    - The input to ADC may need to be protected from dangerous voltages such as static charges or reversed polarity voltages.
  - Amplification
    - To ensure the full-scale signal from the analog results in a full-scale signal to ADC.
  - Bandwidth limiting
    - The signal conditioning provides a low-pass filter to limit the range of frequencies that can be digitized.

# Data Acquisition and Conversion (cont.)

- The sample-and-hold circuit samples the signal and holds it steady for A/D conversion.
  - What is the sample frequency?
- The ADC converts the sampled signal to digital data.
  - The output of ADC connected to CPU through three-state buffers.

# Shannon's Sampling Theorem

- To preserve the full information in the signal, it is necessary to sample at the frequency *at least twice the maximum frequency of the signal.*
  - This minimum sampling frequency is known as the *Nyquist rate*.
  - A signal can be exactly reproduced if it is sampled at a frequency greater than or equal to its Nyquist rate.

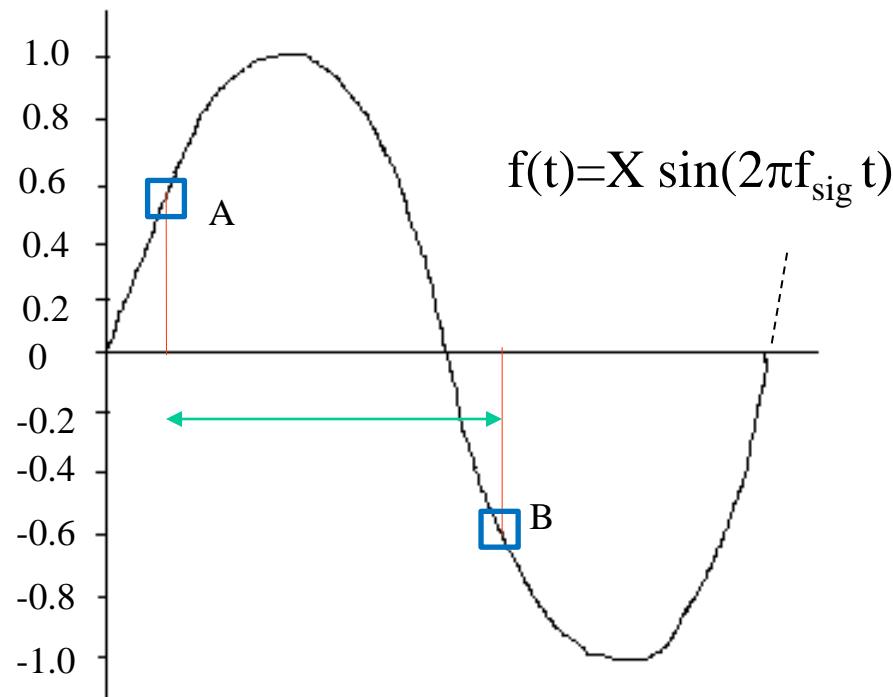
Question: What sample frequency would you choose for signal  $f(t) = 2\sin(2\pi 10^6 t) + 4\sin(2\pi 10^4 t)$  ?

# Aliasing

- If the sampling frequency is less than Nyquist rate, the waveform is said to be undersampled.
- Undersampled signal, when converted back into a continuous time signal, will exhibit a phenomenon called ***aliasing***.

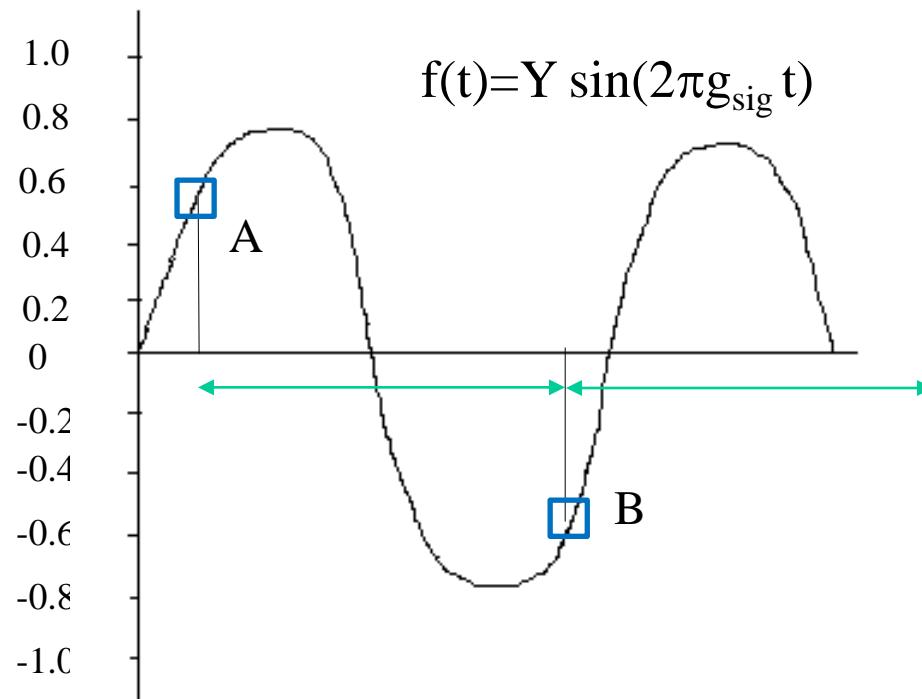
# Sample Examples

- Sampled at twice of the signal frequency.



# Sample Examples

- Un-dersampled, with the sample frequency less than twice of the signal frequency

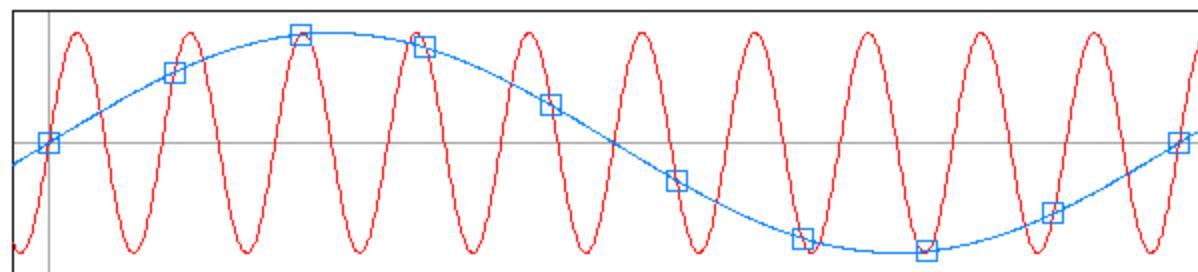


## Alias

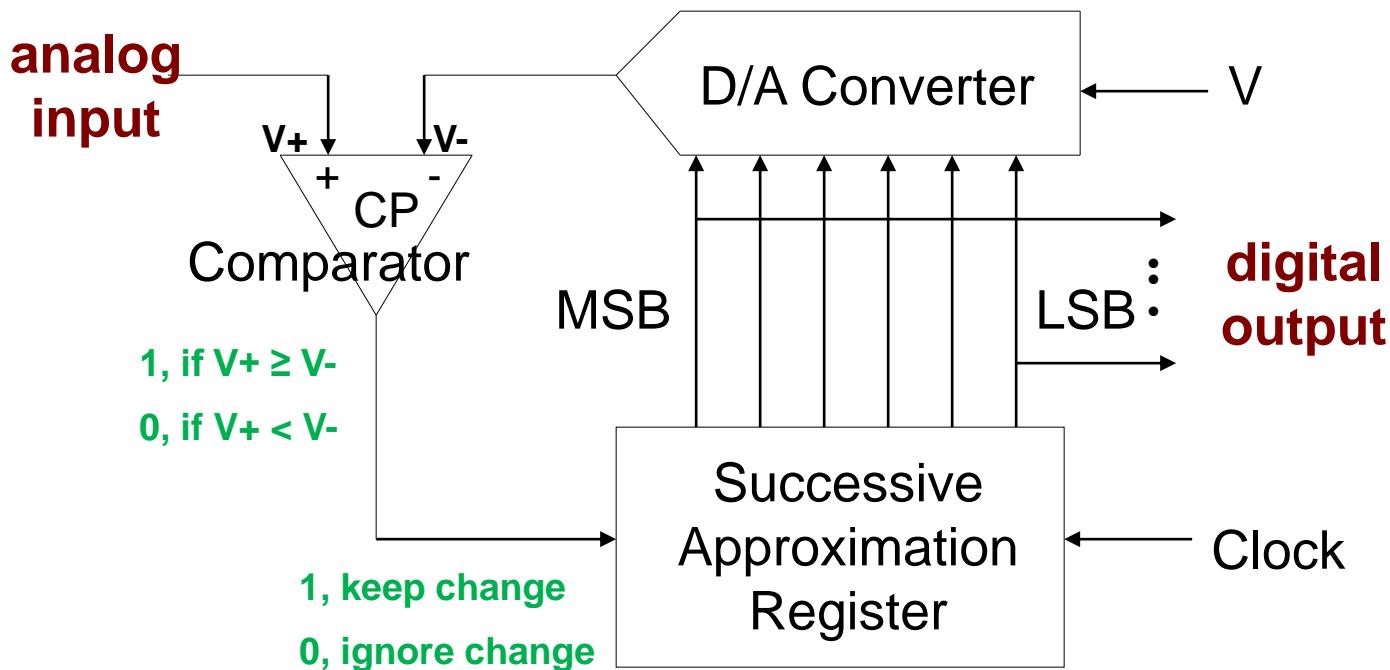
- Alias

- If the sampling frequency is less than Nyquist rate, the waveform is said to be undersampled.
- Undersampled signal, when converted back into a continuous time signal, will exhibit a phenomenon called **aliasing**.

- the presence of unwanted components in the reconstructed signal. These components were not present when the original signal was sampled.
- See the demonstration below



# Successive Approximation Converter

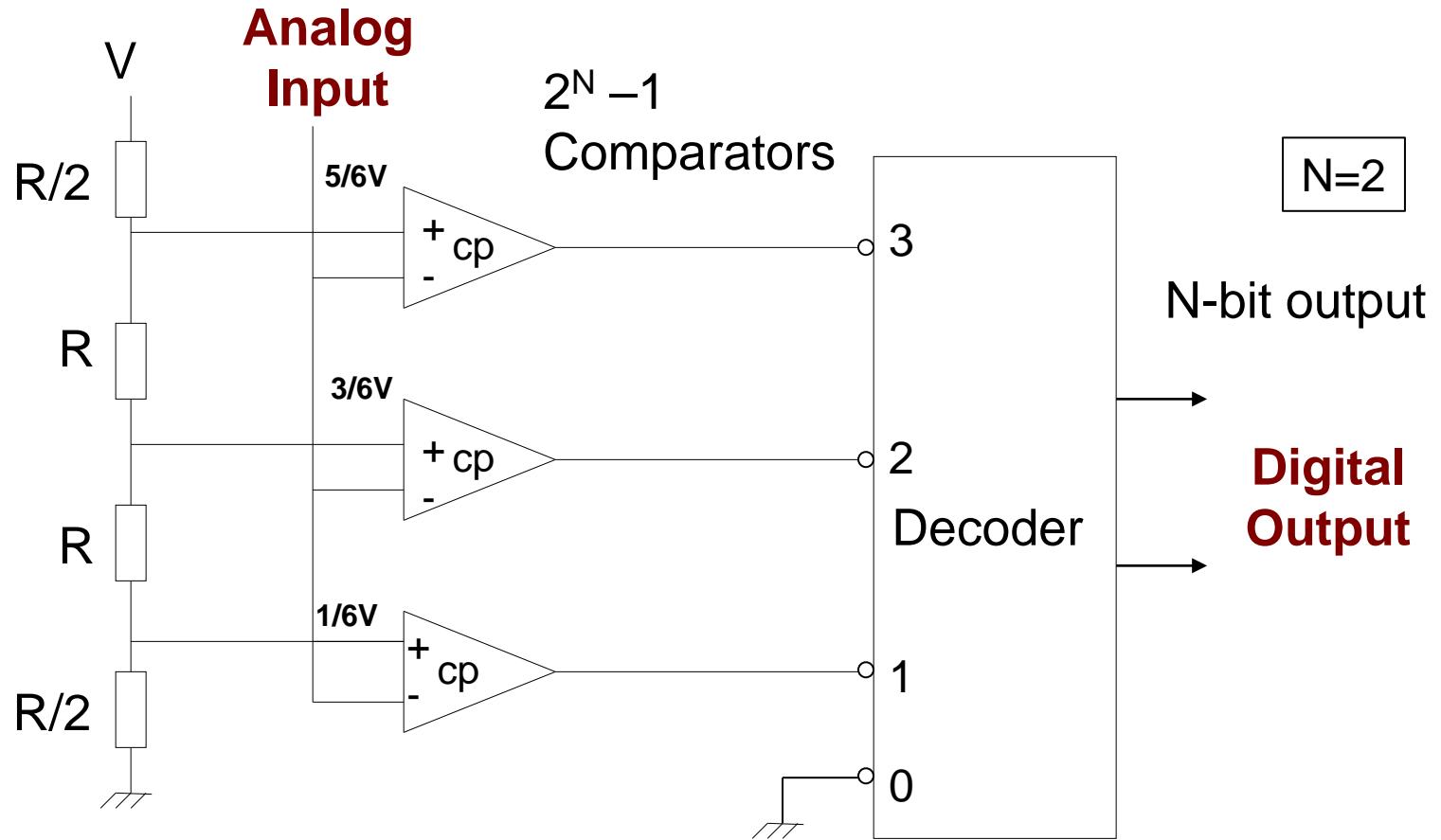


- MSB: most significant bit
- LSB: least significant bit

# Successive Approximation A/D Converter (cont.)

- Each bit in the *successive approximation register* is tested, starting with the most significant bit and working toward the least significant bit.
  - As each bit is set, the output of the D/A converter is compared (by the comparator) with the analog input.
  - If the D/A output is lower than the input signal, the bit remains set and the next bit is tried.
- For an N-bit output, such a bit test needs to be performed N times.

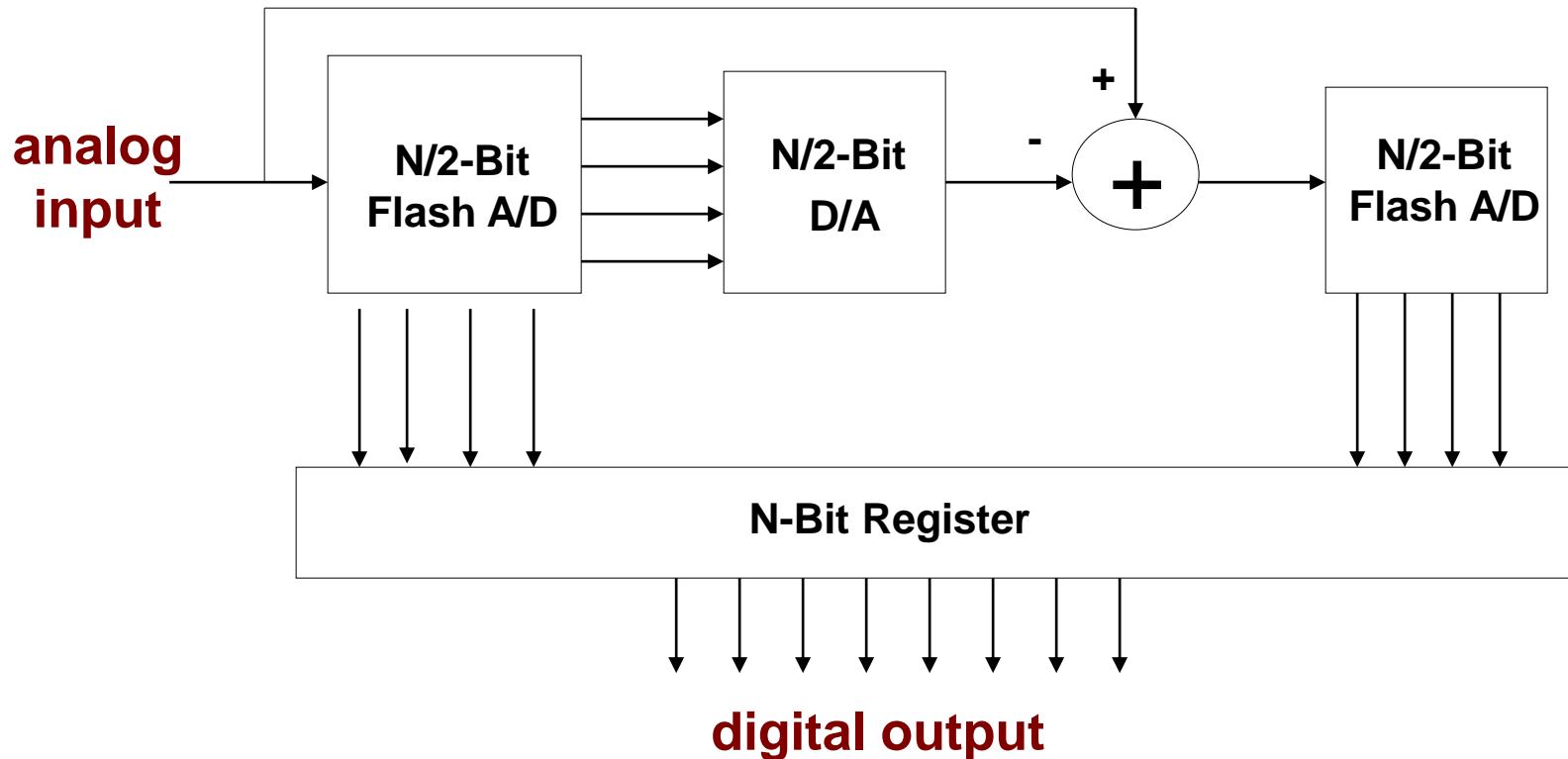
# Parallel A/D Converter



# Parallel A/D Converter (cont.)

- For an N-bit output, the ADC consists of
  - an array of  $2^N-1$  comparators
    - produces a  $(2^N-1)$ -bit code
  - a  $2^N$ -to-N decoder
    - converts  $2^N$ -bit input code to N-bit binary value
- The design is
  - fast
    - hence called **flash ADC**
  - but more costly than the successive approximation ADC

# Two-Stage Parallel A/D Converter\*



# Two-Stage Parallel A/D Converter\*

- The input signal is converted into two steps:
  - First, a coarse estimate is found by the first parallel A/D converter. This digital value is sent to the D/A converter and the adder, where it is subtracted from the original analog value.
  - Next, the difference from the subtraction is converted by the second parallel converter and the result combined with that from the first ADC gives the digital value.

# Two-Stage Parallel A/D Converter\*

- The two-stage ADC has nearly the performance of the parallel converter but without the need of  $2^N - 1$  comparators.
- It offers high resolution and high-speed conversion for applications like video signal processing.

# A/D Converter Specifications

- Conversion time
  - The time required to complete a conversion of the input signal.
  - Determines the upper signal frequency limit that can be sampled without aliasing.  
 $f_{MAX}=1/(2 \times \text{conversion time})$
- Resolution
  - the smallest analog input signal that can be digitized
  - is determined by the number of bits used by DAC

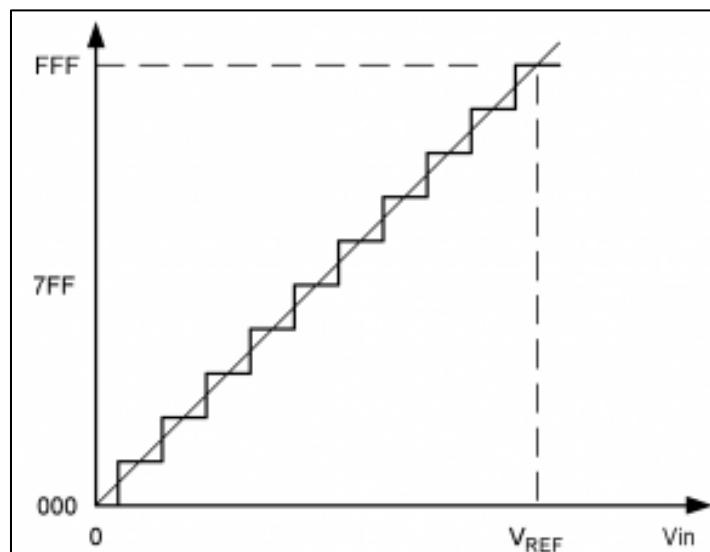
# A/D Converter Specifications (cont.)

- Accuracy
  - Describes how close the digital output is to the theoretically expected digital output for a given analog input.
  - Determines how many bits in the digital output code represent useful information about the input signal.

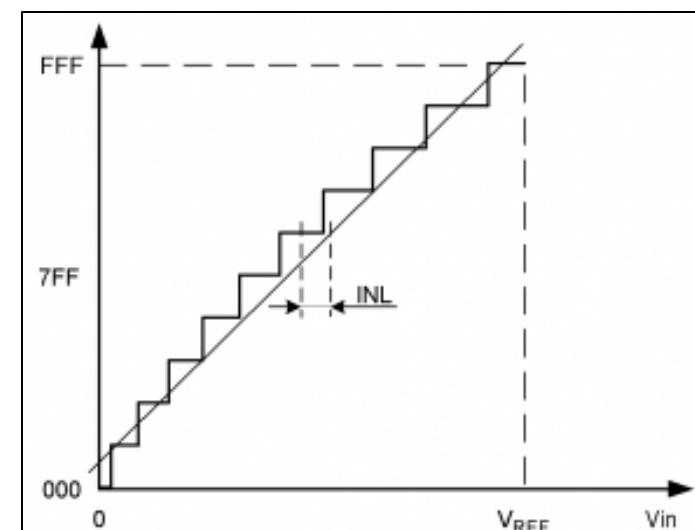
An n-bit ADC may have less than n bits of accuracy!

# A/D Converter Specifications (cont.)

- Linearity
  - The derivation in output codes from the real value



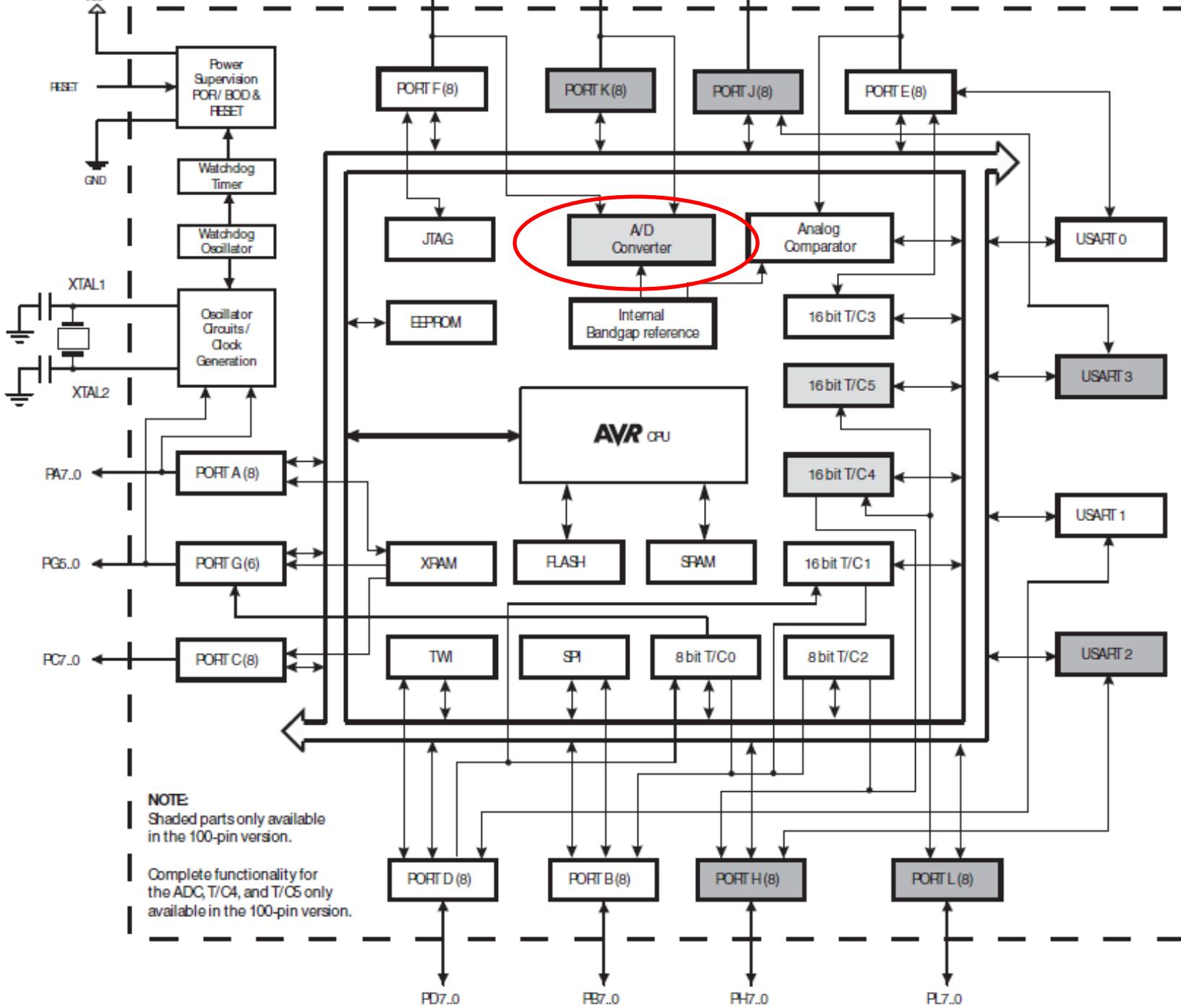
Linear



Non-linear

# A/D Converter Specifications (cont.)

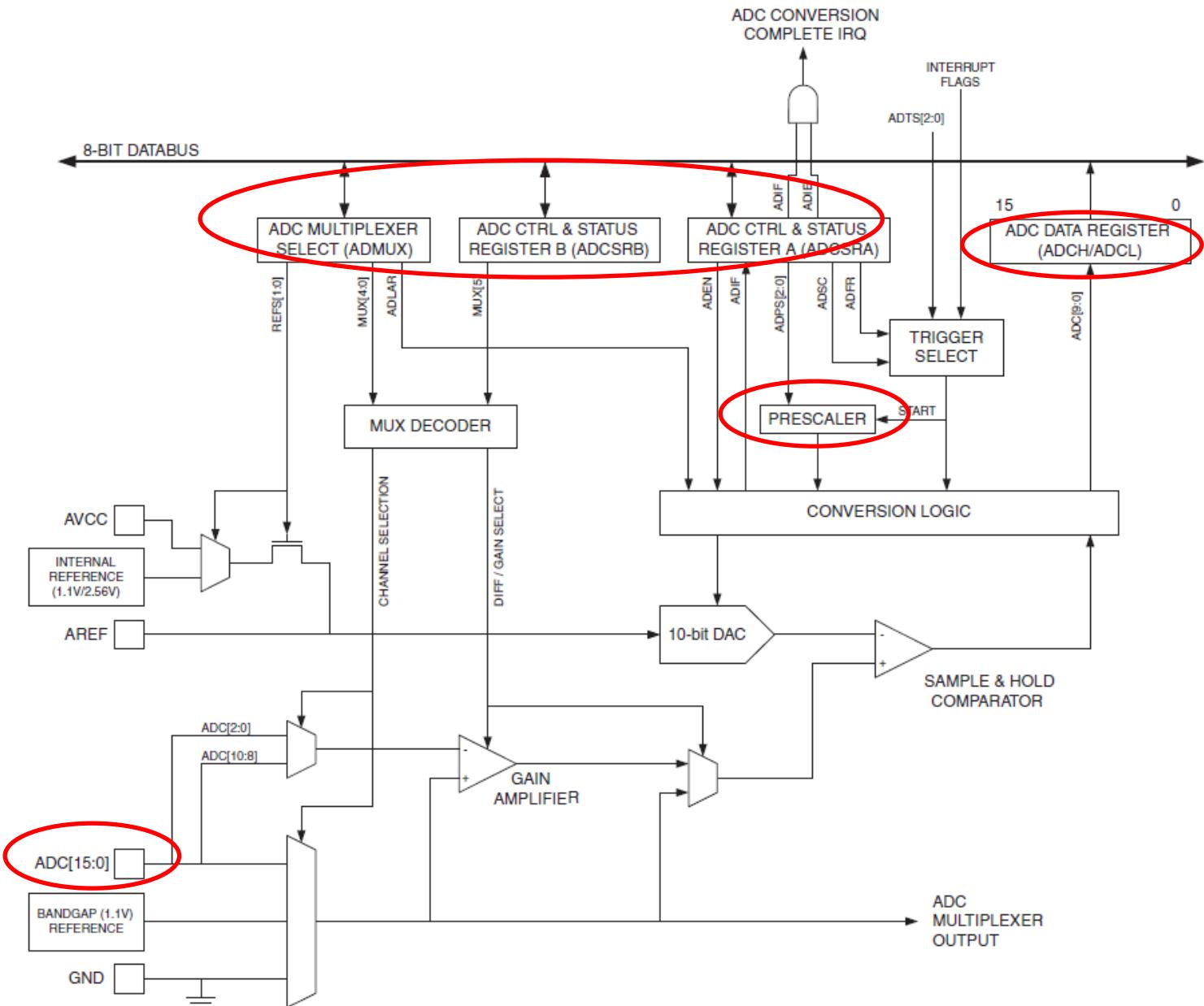
- Aperture time
  - The time that the A/D converter is “looking” at the input signal.
  - It is usually equal to the conversion time.



# ADC on AVR MCU

- 10-bit successive approximation ADC
- Multiple channels
  - Allowing 8/16 single-ended voltage inputs
    - Each refers to 0V (GND)
- Multiple choices of supply voltage
- A brief description of its operation is presented in the next slides.
  - More information can be found from the MCU datasheet.

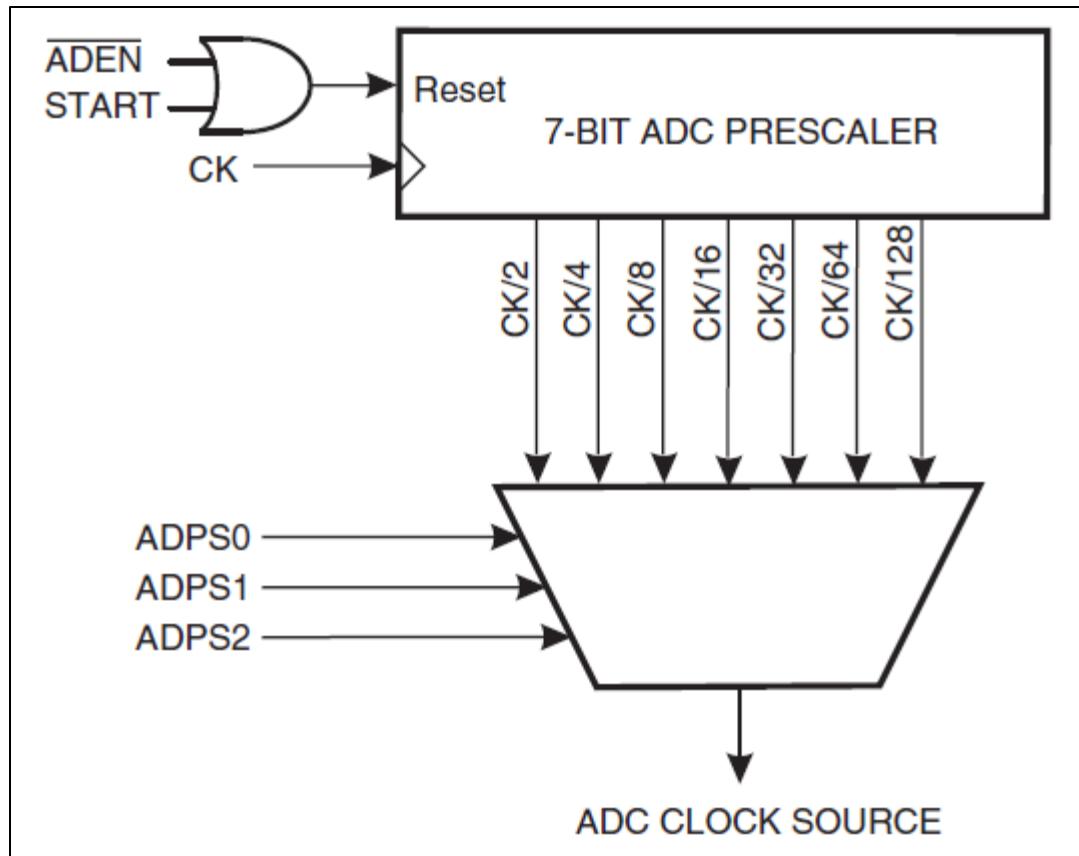
# ADC Structural Diagram



# Key Parameters

- To use ADC, the following parameter values should be specified
- Clock signals
- Reference voltage
- Pins for input signals

# Prescaler



# ADMUX

- ADC Multiplexor Select Register
  - ADLAR (bit 5): ADC Left Adjust Result
    - When the bit is set, the conversion result in ADCH:ADCL aligned to the left; otherwise right
  - MUX4:0: Analog Channel Selection
    - Determine which analog channels are connected to ADC

Bit	7	6	5	4	3	2	1	0
(0x7C)	REFS1	REFS0	ADLAR	MUX4	MUX3	MUX2	MUX1	MUX0
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Initial Value	0	0	0	0	0	0	0	0

# ADMUX (cont.)

- ADC Multiplexor Select Register
  - REFS1-0: voltage reference selection

Bit	7	6	5	4	3	2	1	0
(0x7C)	REFS1	REFS0	ADLAR	MUX4	MUX3	MUX2	MUX1	MUX0
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Initial Value	0	0	0	0	0	0	0	0

Table 26-3. Voltage Reference Selections for ADC

REFS1	REFS0	Voltage Reference Selection <sup>(1)</sup>
0	0	AREF, Internal $V_{REF}$ turned off
0	1	AVCC with external capacitor at AREF pin
1	0	Internal 1.1V Voltage Reference with external capacitor at AREF pin
1	1	Internal 2.56V Voltage Reference with external capacitor at AREF pin

# ADCSA

- ADC Control and Status Register
  - ADEN: ADC Enable
    - 1: enable; 0: disable
  - ADSC: Start Conversion
  - ADIE: ADC Interrupt Enable

Bit	7	6	5	4	3	2	1	0
(0x7A)	<b>ADEN</b>	<b>ADSC</b>	<b>ADATE</b>	<b>ADIF</b>	<b>ADIE</b>	<b>ADPS2</b>	<b>ADPS1</b>	<b>ADPS0</b>
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Initial Value	0	0	0	0	0	0	0	0

# ADCSA

- ADC Control and Status Register
  - ADPS2:0: ADC Prescaler Select Bits

Bit	7	6	5	4	3	2	1	0
(0x7A)	<b>ADEN</b>	<b>ADSC</b>	<b>ADATE</b>	<b>ADIF</b>	<b>ADIE</b>	<b>ADPS2</b>	<b>ADPS1</b>	<b>ADPS0</b>
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Initial Value	0	0	0	0	0	0	0	0

Table 26-5. ADC Prescaler Selections

ADPS2	ADPS1	ADPS0	Division Factor
0	0	0	2
0	0	1	2
0	1	0	4
0	1	1	8
1	0	0	16
1	0	1	32
1	1	0	64
1	1	1	128

# Example

- Convert an analog voltage input from POT to digital value and display the value on LEDs.
  - POT: Potentiometer

# Example

```
.include "m2560def.inc"

.def temp=r16

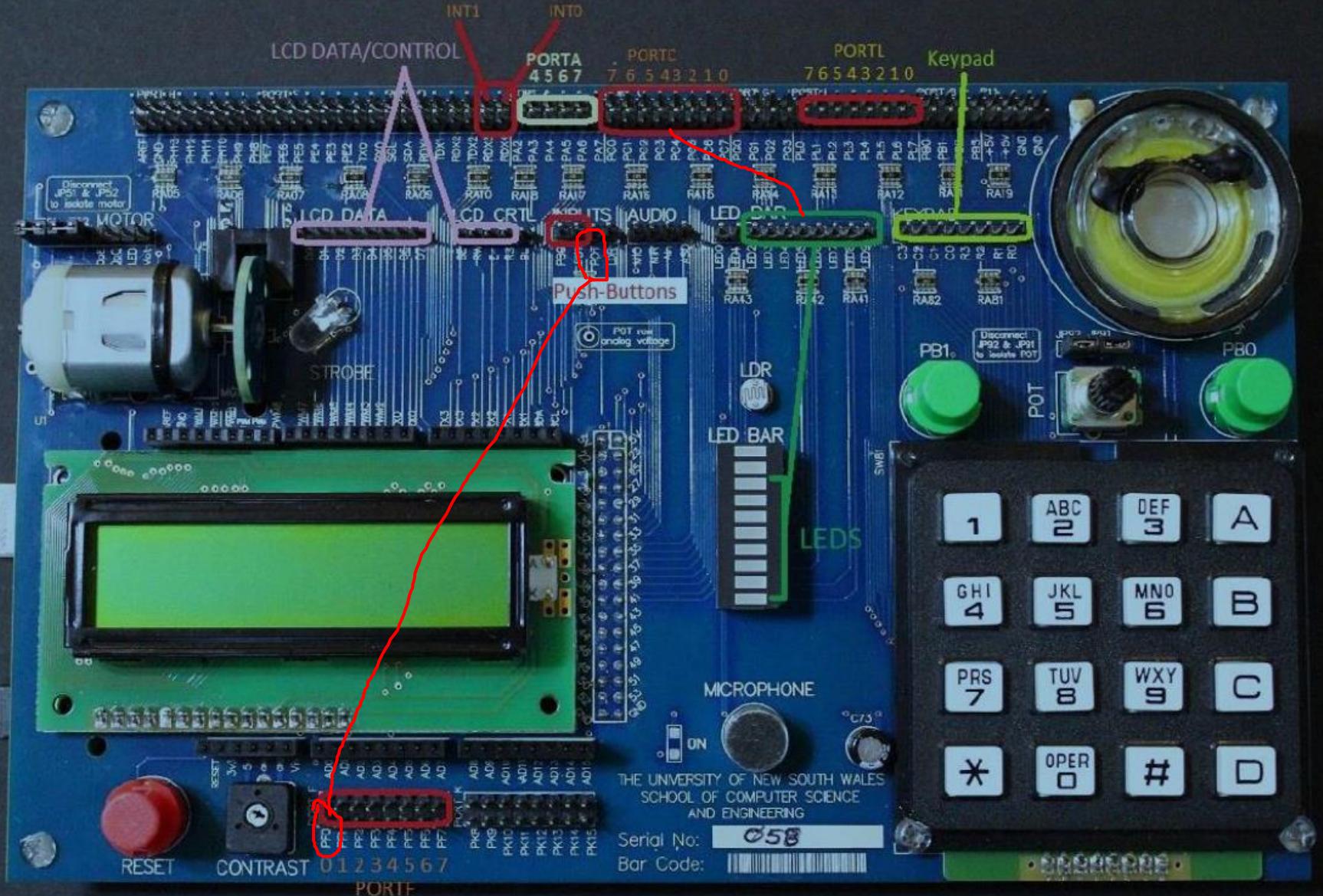
    ser temp
    out DDRC, temp

    ; Vref=AVcc
    ldi temp, 1<<REFS0
    sts ADMUX, temp
    ;set prescaller to 128 and enable ADC
    ldi temp, (1<<ADPS2)|(1<<ADPS1)|(1<<ADPS0)|(1<<ADEN)
    sts ADCSRA, temp

    ; Left-adjust, Ref. = AVCC, Single-ended on ADC0 is default
    ldi temp, (1<<ADLAR) | (1<<REFS0)
    sts ADMUX, temp
```

# Example (cont.)

```
new_conversion:  
    ; start conversion, with the single conversion mode  
    lds temp, ADCSRA  
    ori temp, (1<<ADSC)  
    sts ADCSRA, temp  
  
    ;wait until the conversion is complete  
loop:  
    lds temp, ADCSRA  
    sbrc temp, ADSC ; also the two lines can be used.  
    jmp loop  
  
    ; on completion of the conversion, display the digital output on Port C  
    lds temp, ADCH  
    out PORTC, temp  
  
    rjmp new_conversion
```



# Reading Material

- Chapter 13: Analog Input and Output.  
Microcontrollers and Microcomputers by  
Fredrick M. Cady.
- AVR Mega2560 Data Sheet.
  - ADC

# Homework

1. The A/D converter conversion time is 100 us. What is the maximum frequency of a signal that can be digitalized without aliasing occurring?

# **Microprocessors & Interfacing**

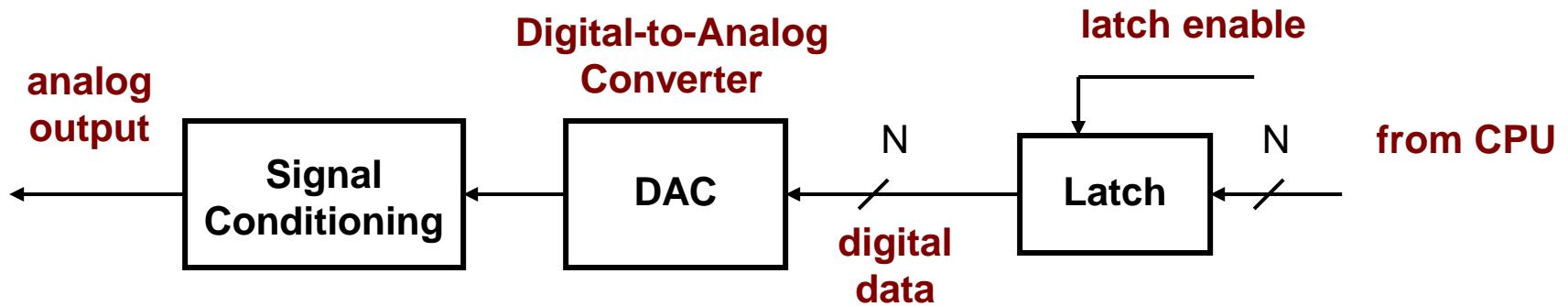
*Analog Input/Output (II)*

Lecturer : Annie Guo

# Lecture Overview

- Analog output
  - DAC

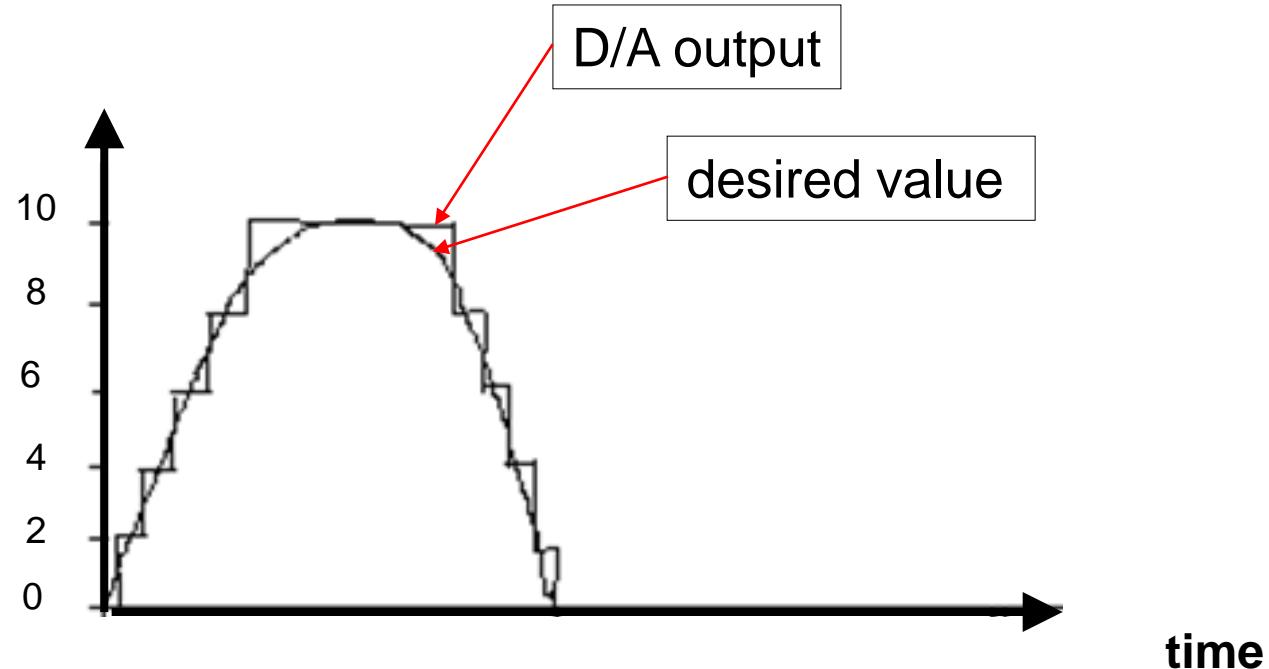
# Digital-to-Analog Conversion (DAC)



# Digital-to-Analog Conversion (cont.)

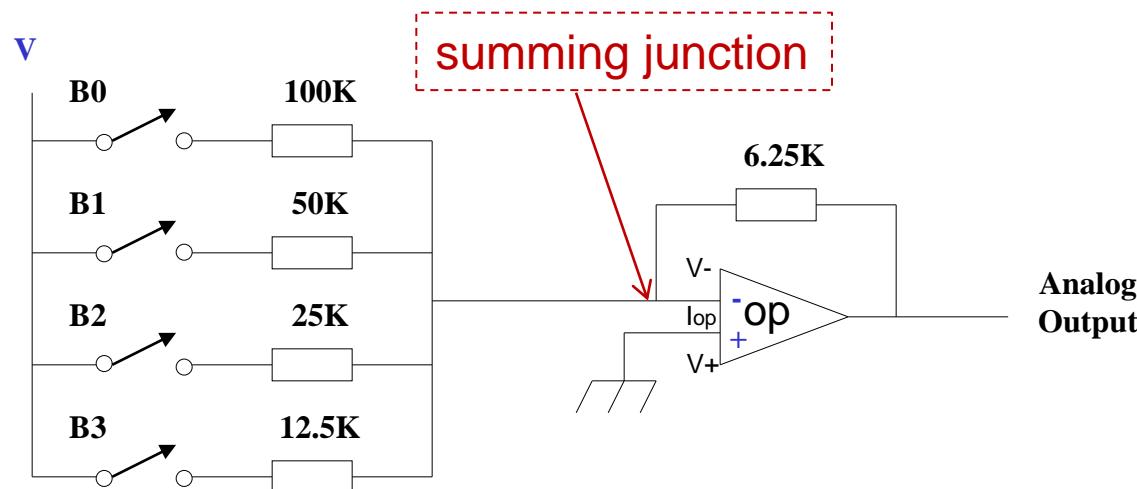
- A parallel output interface connects Digital-to-Analog Converter (**D/A Converter**, or **DAC**) to CPU.
- The latches may be part of DAC or the output interface.
- **Digital value is converted into “continuous” value.**
  - **Quantized**
- A signal conditioning block may be used as a filter to smooth the quantized nature of the output.
  - The signal conditioning block also provides isolation, buffering and voltage amplification if needed.

# Quantized D/A Output



# Binary-Weighted D/A Converter

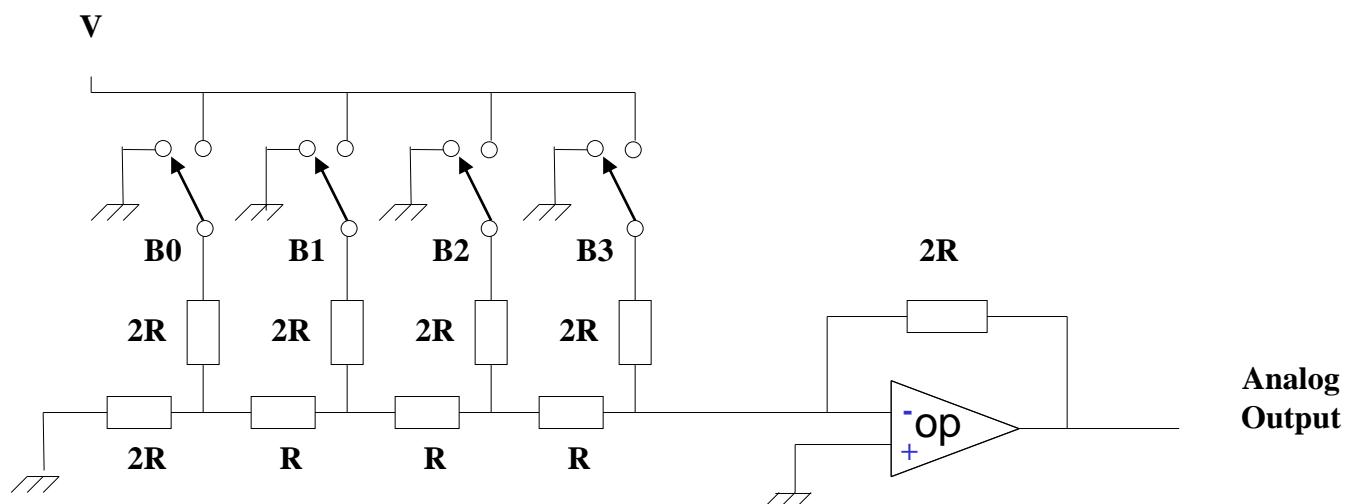
- Example: 4-bit DAC
  - As a switch for a bit is closed, a *weighted current* is supplied to the *summing junction* of the amplifier (OP).
  - For high-resolution D/A converters, the binary-weighted type must have a wide range of resistors. This may affect the output accuracy.



• OP: operational amplifier  
• Resolution: number of bits in the digital value.

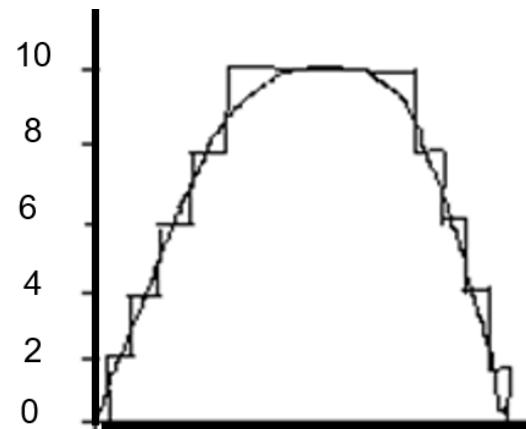
# R-2R Ladder D/A Converter

- As a switch changes from the grounded position to the reference position, a binary-weighted current is supplied to the summing junction.
- For high-resolution D/A converters, a wide range of resistors are not required, providing better accuracy for the output.



# D/A Converter Specifications

- **Resolution**
  - The resolution is determined by the number of bits and is given as the output voltage corresponding to the smallest digital step, i.e. 1 LSB.

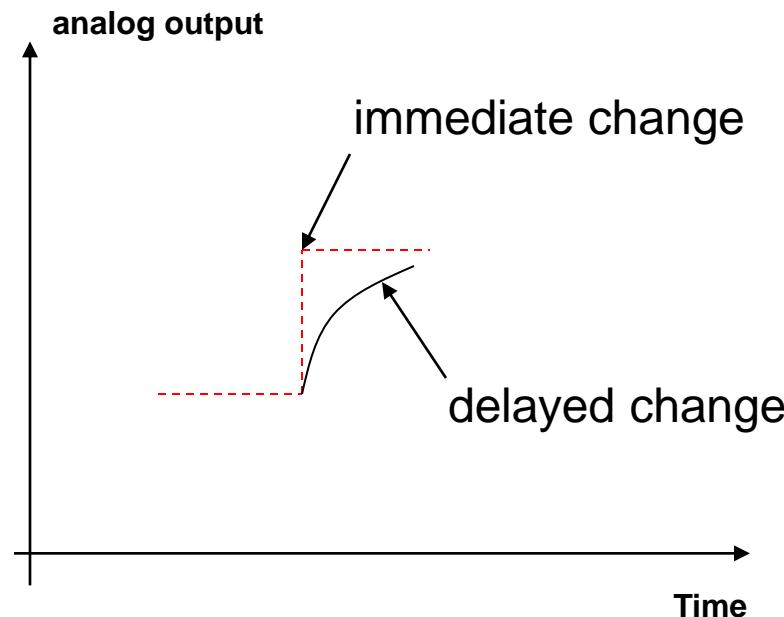


# D/A Converter Specifications

- **Linearity**
  - Linearity shows how close the output voltage to the idea values (a straight line drawn through zero and full-scale).
    - A way of measuring accuracy of DAC
      - Ideally, any two adjacent digital codes correspond to output analog voltages that are exactly one LSB apart.

# D/A Converter Specifications

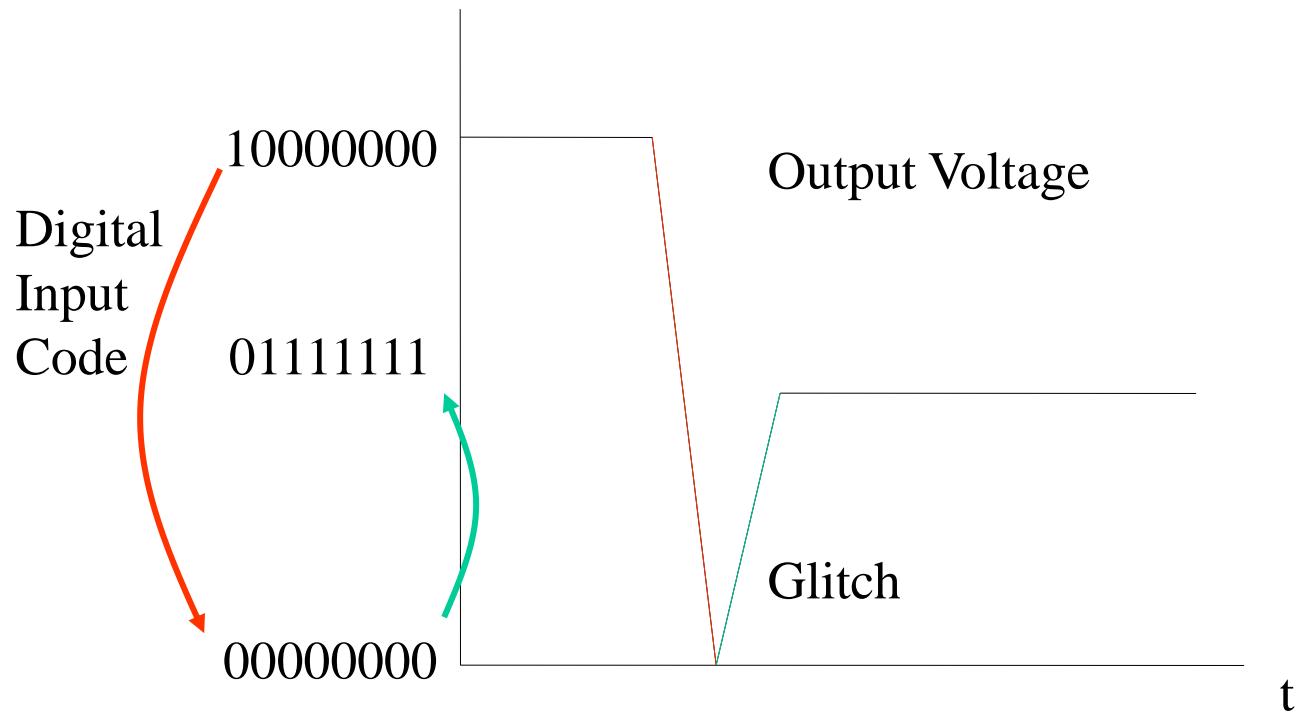
- **Settling Time**
  - The time taken for the output voltage to settle to within a specified error band, usually  $\pm \frac{1}{2}$  LSB.



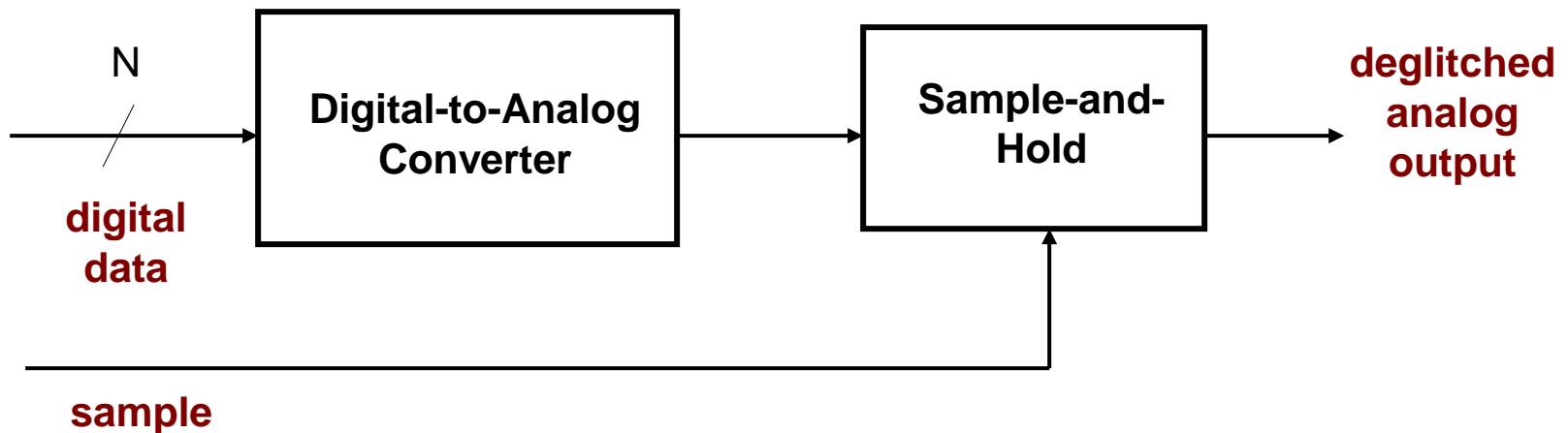
# Glitches

- A glitch is caused by asymmetrical switching in the D/A switches. If a switch changes from 1 to 0 faster than from 0 to 1, a glitch may occur.
  - Consider changing the input digital code of a 4-bit D/A from 1000 to 0111 in the next slide.
- The D/A converter glitch can be eliminated by using a sample-and-hold circuit.

# D/A Output Glitch



# Deglitched D/A



# Reading Material

- Chapter 13: Analog Input and Output.  
Microcontrollers and Microcomputers by  
Fredrick M. Cady.

# **Microprocessors & Interfacing**

*Analog Input/Output (I)*

Lecturer : Annie Guo

# Lecture Overview

- Analog output
  - PWM

# PWM

- PWM (Pulse Width Modulation) is a way of digitally encoding analog signal levels.
  - By using high-resolution counters, the duty cycle (pulse width/period) of a pulse wave is modulated to encode a specific analog signal level.
- PWM is a powerful technique for controlling analog circuits/devices with the microcontroller's digital output.
- It is used in a wide variety of applications
  - E.g. motor speed control

# PWM (cont.)

- The PWM signal is still digital
  - Its value is either full high or full low.
- A low-pass filter is required for obtaining an analog value
  - The output value is directly proportional to the pulse width.
    - By changing the pulse width of the PWM waveform, we can control the analog value.
- A low-pass filter can be of different forms
  - e.g. circuit, motor



# PWM Analog Output

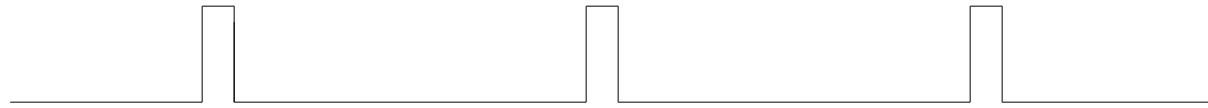
- PWM (Pulse Width Modulation) is a way of digitally encoding analog signal levels.
  - By using high-resolution counters, the duty cycle (pulse width/period) of a pulse wave is modulated to encode a specific analog signal level.
- PWM is a powerful technique for controlling analog circuits/devices with the processor's digital output.
- It is used in a wide variety of applications
  - E.g. motor speed control

# PWM Analog Output (cont.)

- The PWM signal is still digital
  - Its value is either full high or full low.
- A low-pass filter is required to smooth the input signal and eliminate the inherent noise components in PWM signal.
- The output voltage is directly proportional to the pulse width.
  - By changing the pulse width of the PWM waveform, we can control the output value.

# PWM Signal Examples

Duty cycle=10%



Duty cycle=50%



Duty cycle=90%



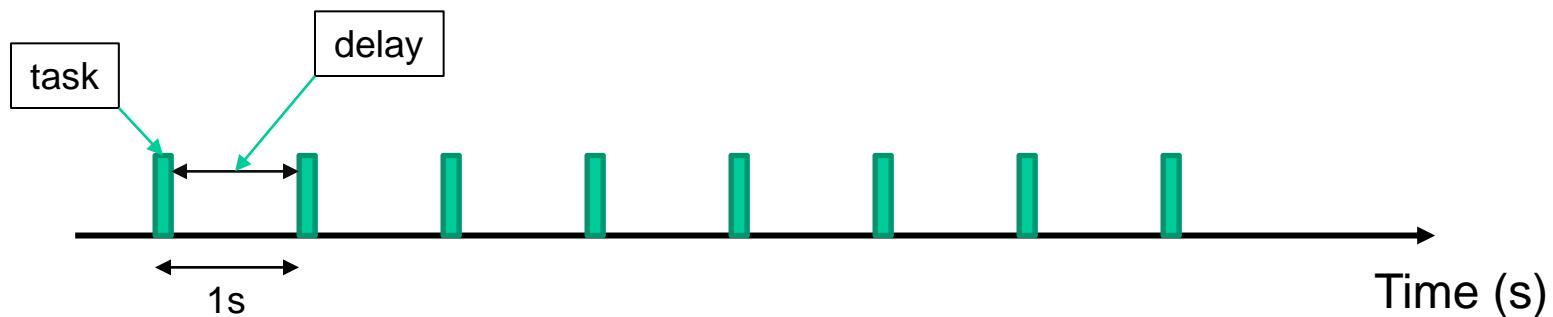
- Duty cycle:  $(\text{pulse width})/\text{period}$

# PWM Generation in AVR

- PWM can be obtained through the provided timers.

# Recall: Example 1 (Mon. Week 7)

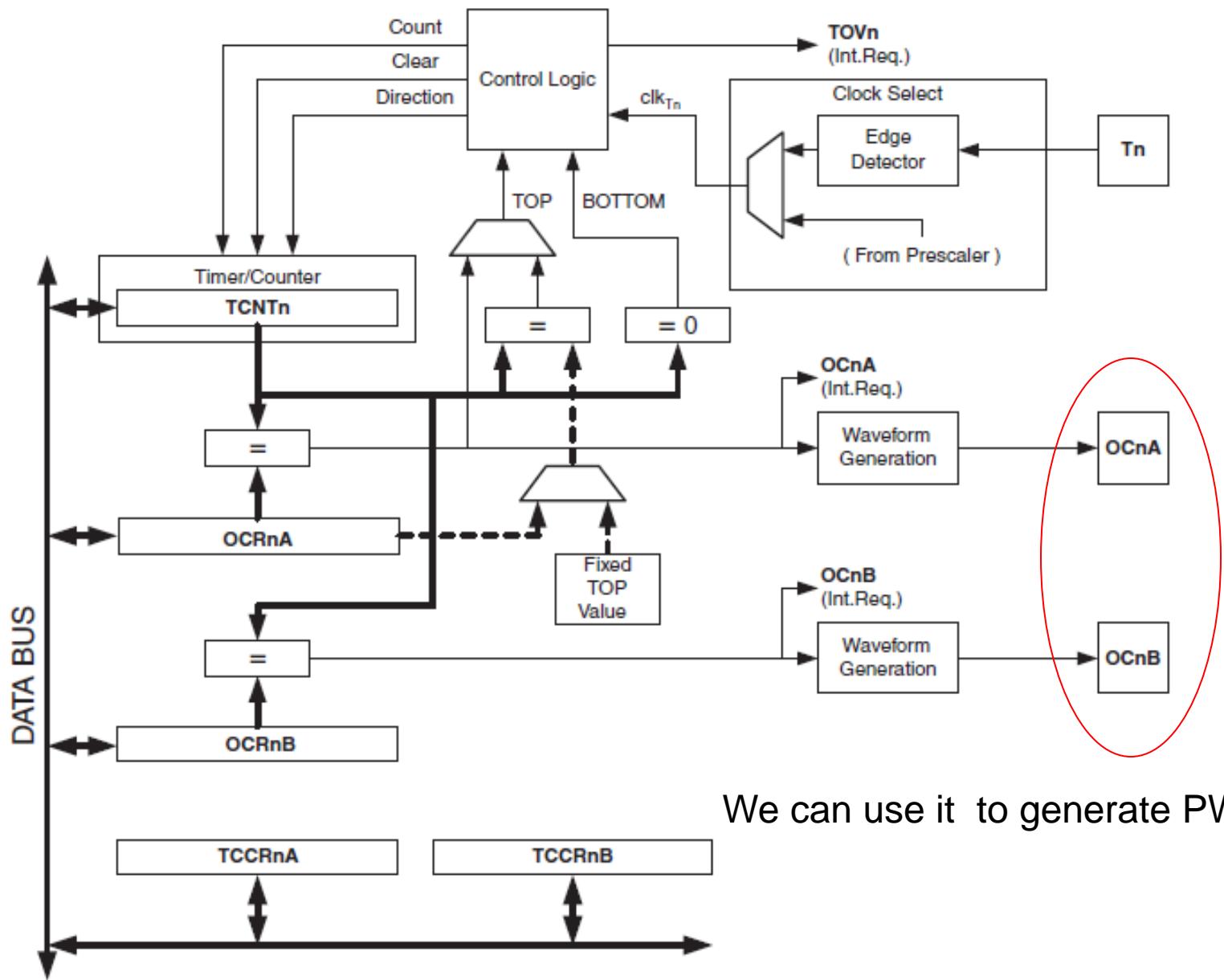
- Implement a scheduler that can execute a task every one second.
  - Can be realized with
    - software design,
      - Software generates the delay
        - » With nop instructions
        - » With other tasks of known execution time
      - hardware design
        - Used here and solution is given in the next slides



# Recall: Example 1 Solution

- Use **8-bit Timer0** to “count” the time
  - Let’s set Timer0 prescaler to /64 (i.e. the system frequency is divided by 64)
    - The full counting duration (time-out) for the setting should be
      - $256 \times (\text{clock period}) = 256 \times 64 / (16 \text{ MHz})$   
= 1024 us
        - » Namely, we can set the Timer0 overflow interrupt that is to occur every 1024 us.
        - » Note, clock period = 1/16 MHz (obtained from the data sheet); the 8-bit counter can count 256 clock cycles.
    - For one second, there are
      - $1000000 / 1024 \approx 1000$  interrupts

# Recall: Timer0



# Configuration for PWM

- TCCR0A/B

Bit	7	6	5	4	3	2	1	0	
0x24 (0x44)	COM0A1	COM0A0	COM0B1	COM0B0	-	-	WGM01	WGM00	TCCR0A
Read/Write	R/W	R/W	R/W	R/W	R	R	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	
0x25 (0x45)	FOC0A	FOC0B	-	-	WGM02	CS02	CS01	CS00	TCCR0B
Read/Write	W	W	R	R	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Mode	WGM2	WGM1	WGM0	Timer/Counter Mode of Operation	TOP	Update of OCRx at	TOV Flag Set on <sup>(1)(2)</sup>
0	0	0	0	Normal	0xFF	Immediate	MAX
1	0	0	1	PWM, Phase Correct	0xFF	TOP	BOTTOM
2	0	1	0	CTC	OCRA	Immediate	MAX
3	0	1	1	Fast PWM	0xFF	TOP	MAX
4	1	0	0	Reserved	-	-	-
5	1	0	1	PWM, Phase Correct	OCRA	TOP	BOTTOM
6	1	1	0	Reserved	-	-	-
7	1	1	1	Fast PWM	OCRA	BOTTOM	TOP

# Configuration for PWM (cont.)

- TCCR0A/B
  - Phase Correct PWM

Bit	7	6	5	4	3	2	1	0	
0x24 (0x44)	COM0A1	COM0A0	COM0B1	COM0B0	–	–	WGM01	WGM00	TCCR0A
Read/Write	R/W	R/W	R/W	R/W	R	R	R/W	R/W	
Bit	7	6	5	4	3	2	1	0	
0x25 (0x45)	FOC0A	FOC0B	–	–	WGM02	CS02	CS01	CS00	TCCR0B
Read/Write	W	W	R	R	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Table 16-4. Compare Output Mode, Phase Correct PWM Mode<sup>(1)</sup>

COM0A1	COM0A0	Description
0	0	Normal port operation, OC0A disconnected
0	1	WGM02 = 0: Normal Port Operation, OC0A Disconnected WGM02 = 1: Toggle OC0A on Compare Match
1	0	Clear OC0A on Compare Match when up-counting. Set OC0A on Compare Match when down-counting
1	1	Set OC0A on Compare Match when up-counting. Clear OC0A on Compare Match when down-counting

# Configuration for PWM (cont.)

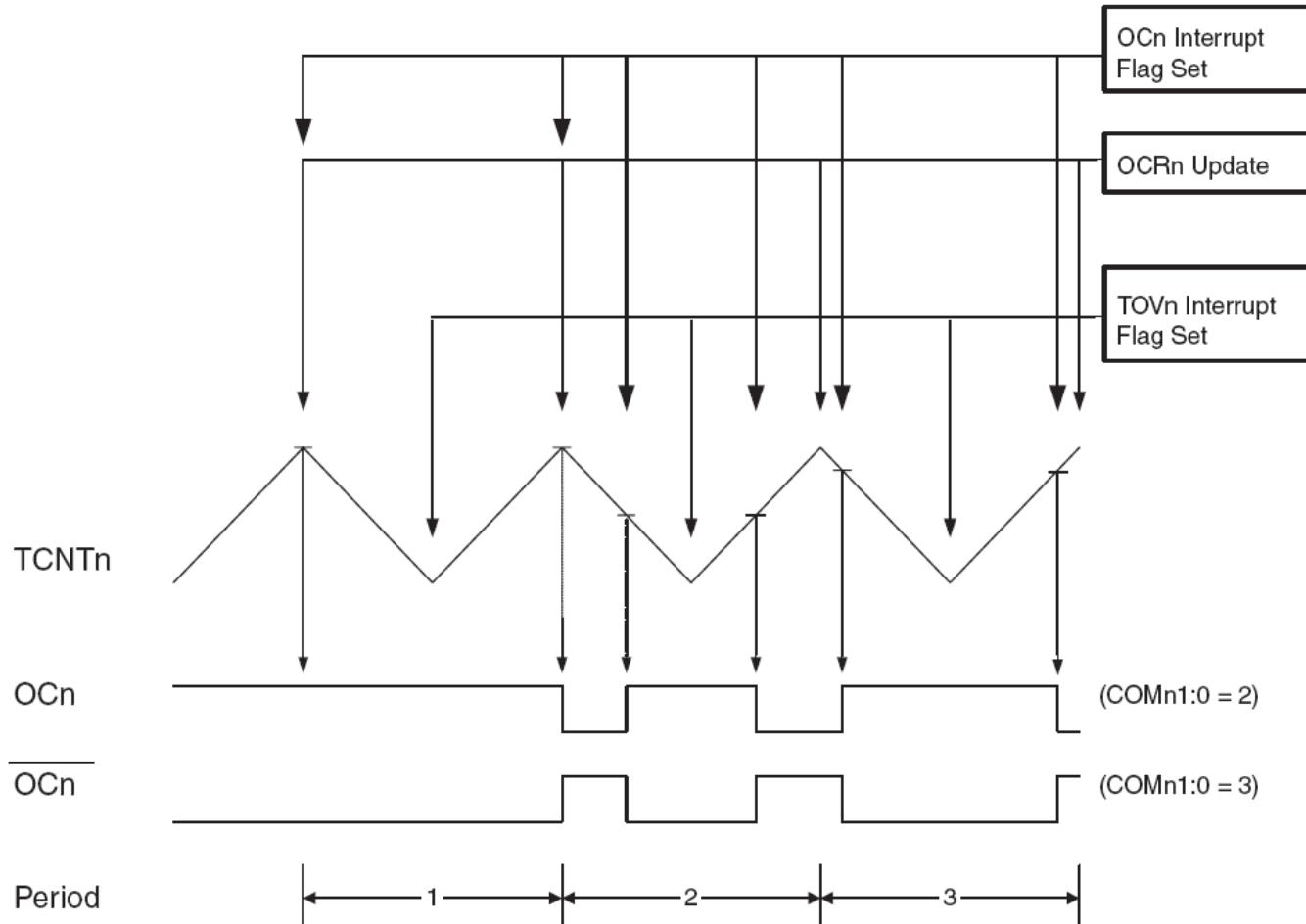
- TCCR0A/B
  - Fast PWM

Bit	7	6	5	4	3	2	1	0	
0x24 (0x44)	COM0A1	COM0A0	COM0B1	COM0B0	–	–	WGM01	WGM00	TCCR0A
Read/Write	R/W	R/W	R/W	R/W	R	R	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	
0x25 (0x45)	FOC0A	FOC0B	–	–	WGM02	CS02	CS01	CS00	TCCR0B
Read/Write	W	W	R	R	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

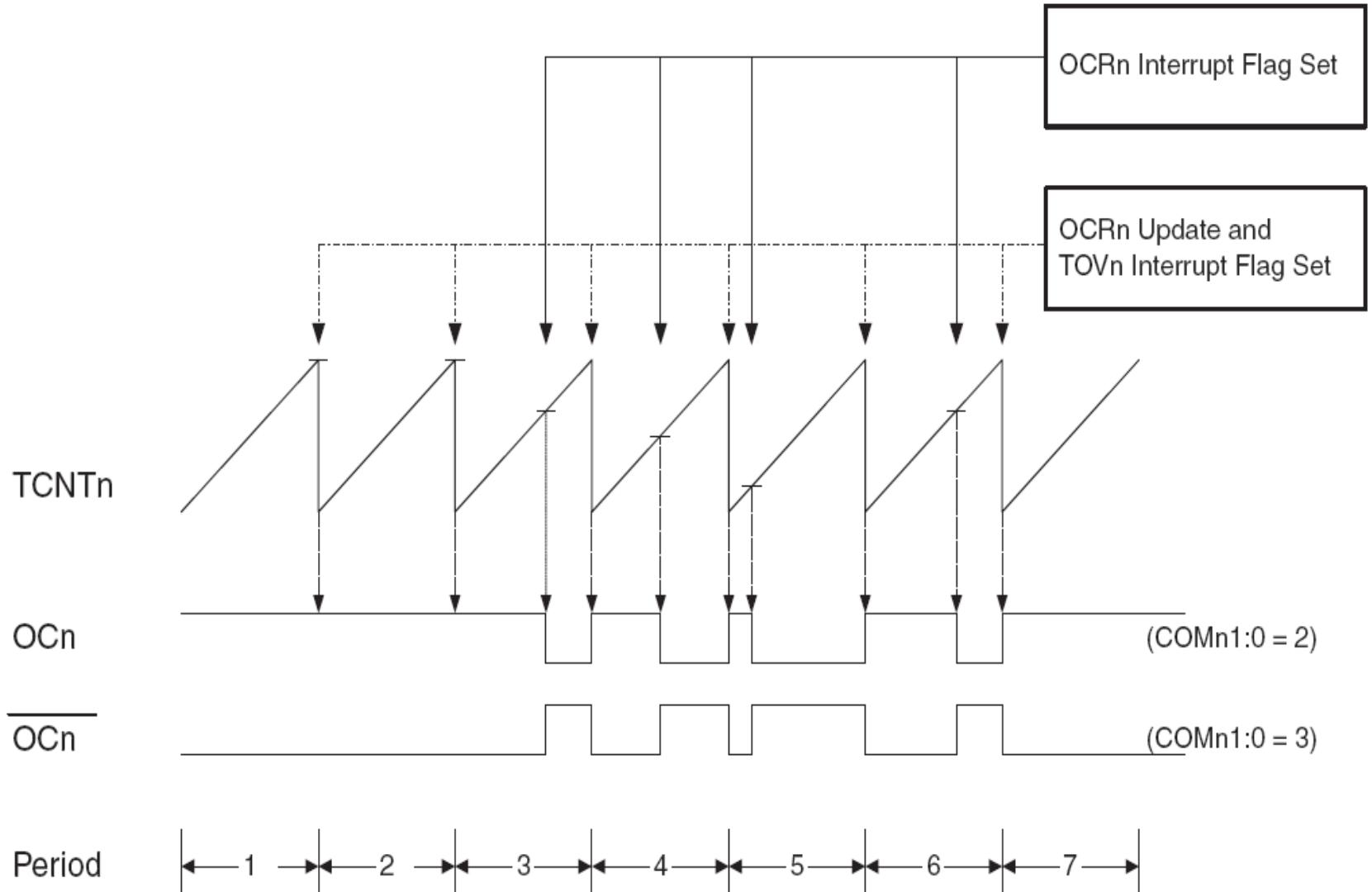
Table 16-3. Compare Output Mode, Fast PWM Mode<sup>(1)</sup>

COM0A1	COM0A0	Description
0	0	Normal port operation, OC0A disconnected
0	1	WGM02 = 0: Normal Port Operation, OC0A Disconnected WGM02 = 1: Toggle OC0A on Compare Match
1	0	Clear OC0A on Compare Match, set OC0A at BOTTOM (non-inverting mode)
1	1	Set OC0A on Compare Match, clear OC0A at BOTTOM (inverting mode)

# Phase Correct PWM

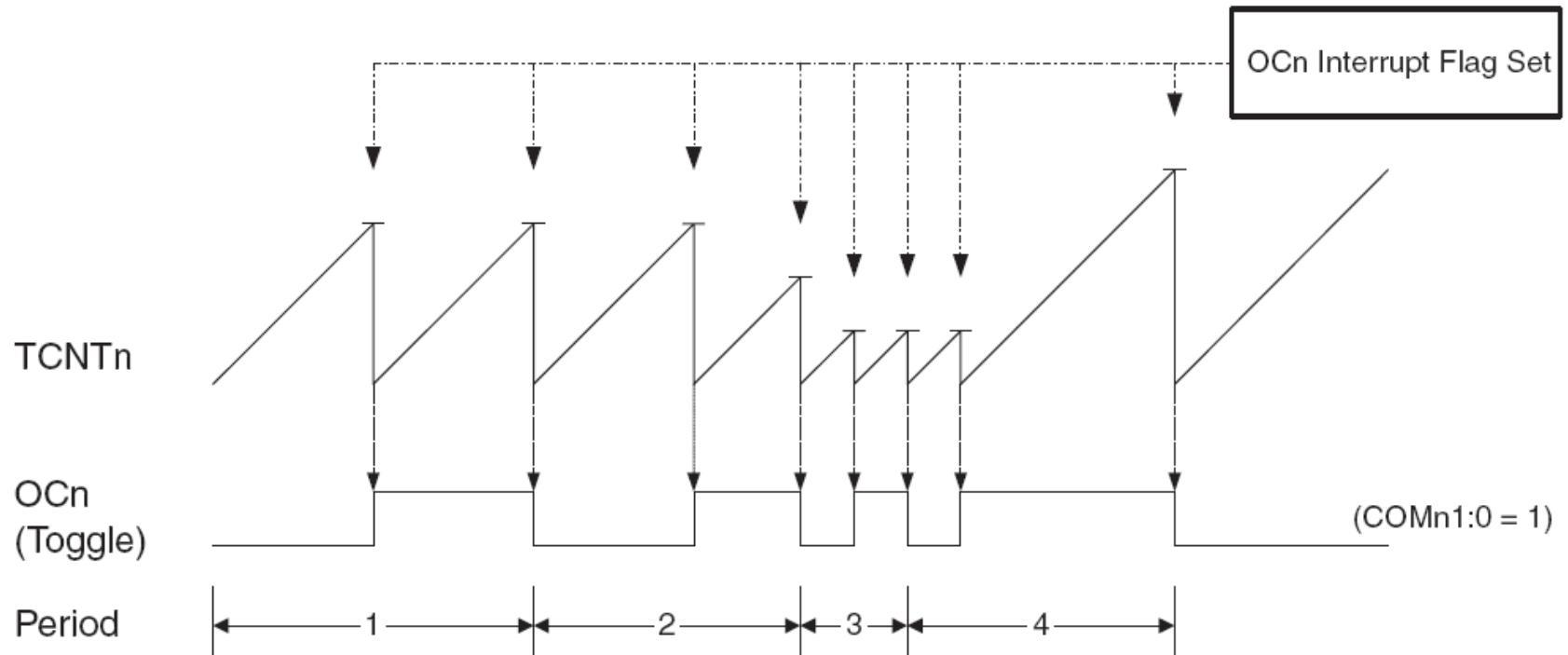


# Fast PWM



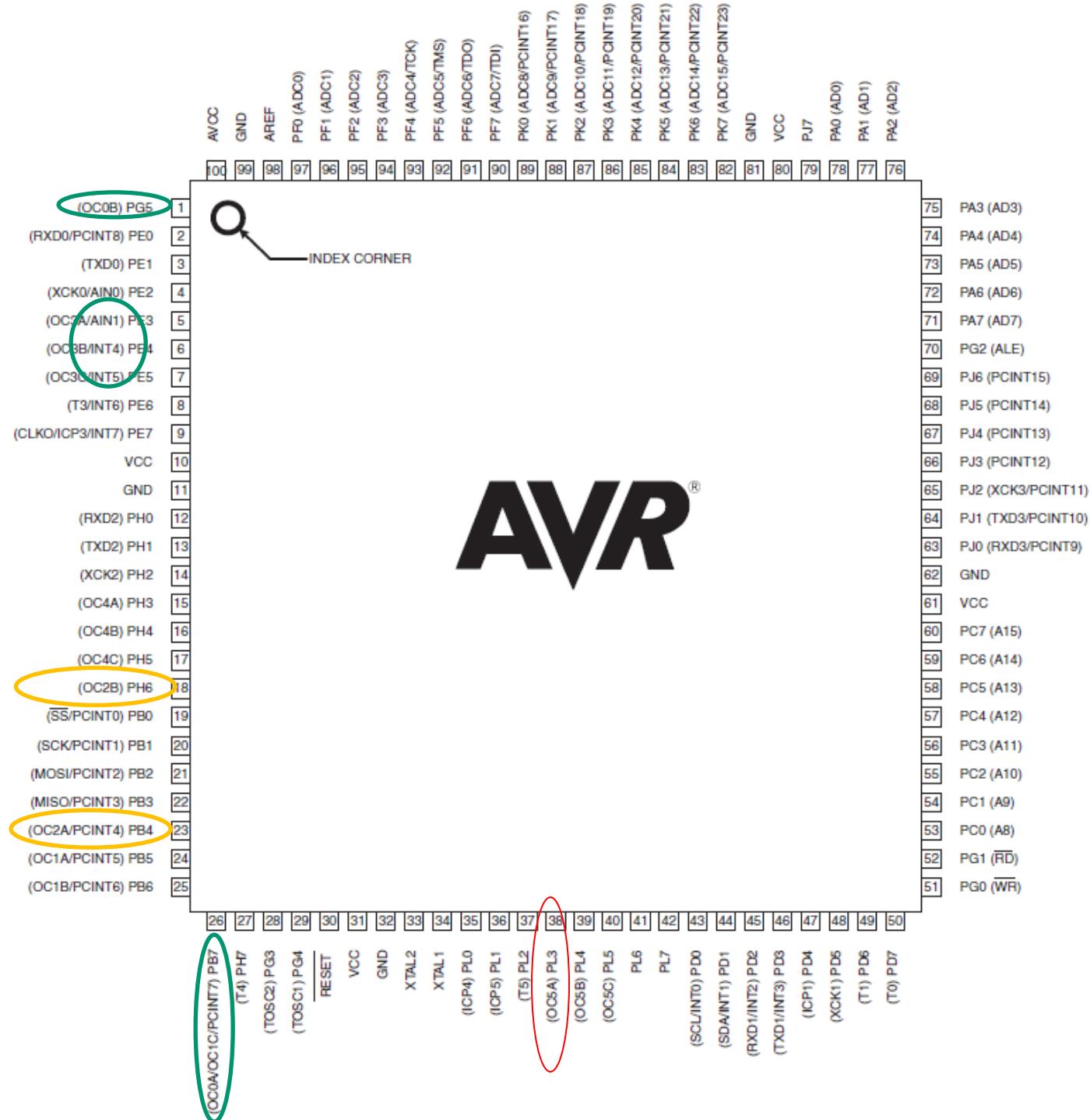
# CTC\*

- Clear Timer on Compare Match



# Example

- Generate a PWM waveform.



# Example (solution)

- Use Timer5
  - Set OC5A as output
  - Set the Timer5 operation mode as Phase Correct PWM mode
  - Set the timer clock

# 16-bit Timer Block Diagram\*

16-bit Timer/Counter Block Diagram<sup>(1)</sup>

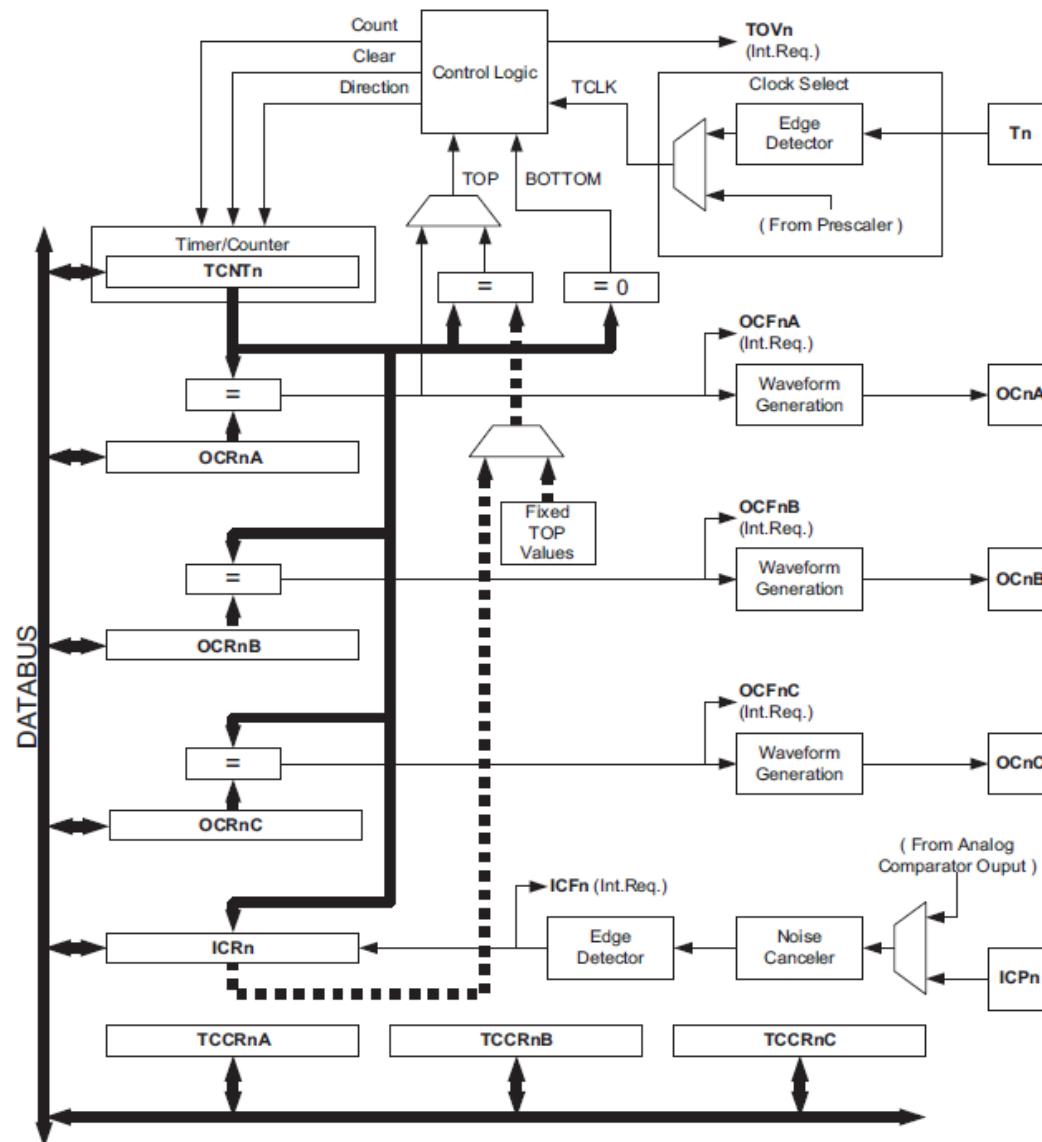
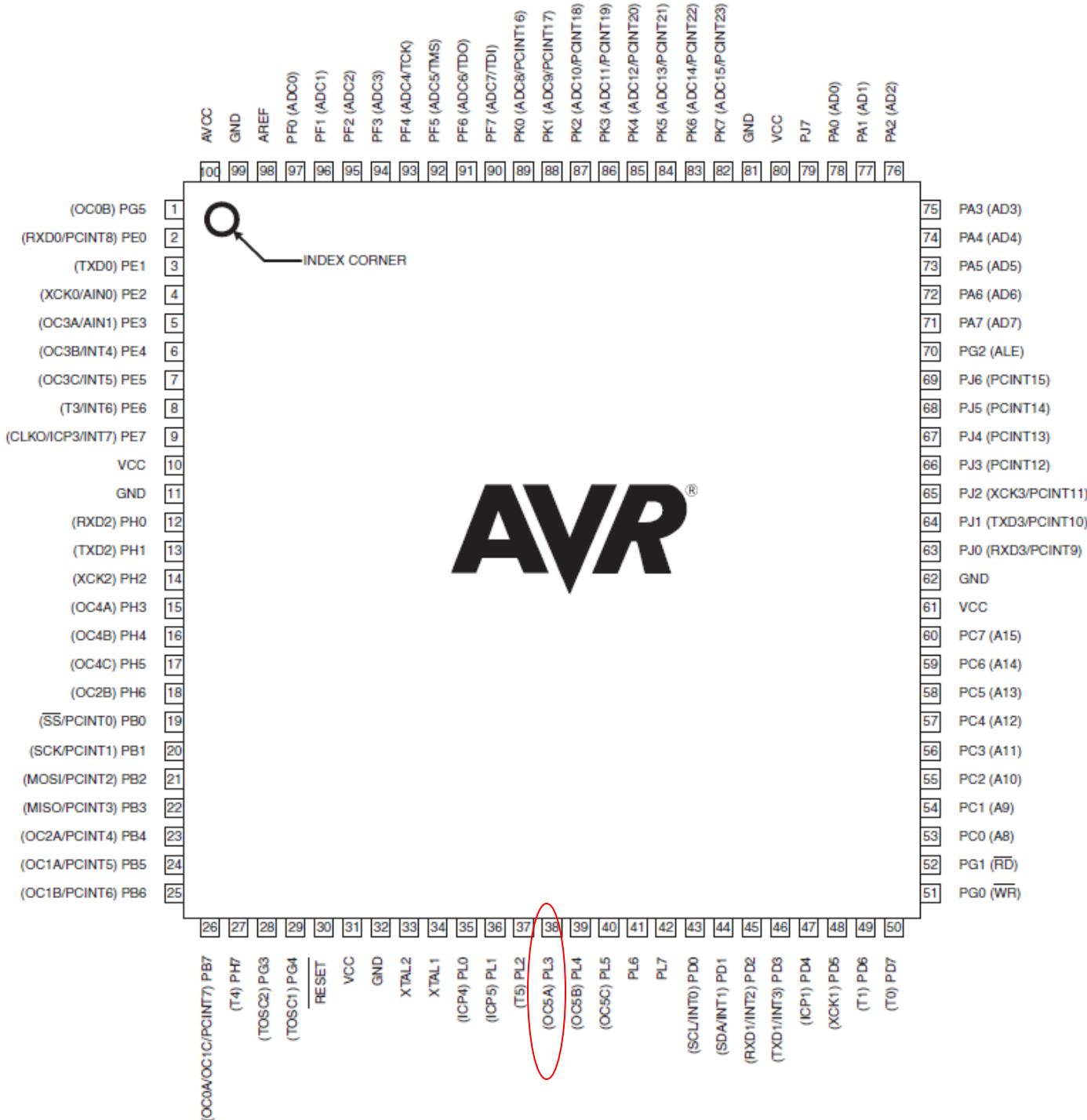


Table 17-2. Waveform Generation Mode Bit Description<sup>(1)</sup>

Mode	WG Mn3	WG Mn2 (CTCn)	WG Mn1 (PWMn1)	WG Mn0 (PWMn0)	Timer/Counter Mode of Operation	TOP	Update of OCRnx at	TOVn Flag Set on
0	0	0	0	0	Normal	0xFFFF	Immediate	MAX
1	0	0	0	1	PWM, Phase Correct, 8-bit	0x00FF	TOP	BOTTOM
2	0	0	1	0	PWM, Phase Correct, 9-bit	0x01FF	TOP	BOTTOM
3	0	0	1	1	PWM, Phase Correct, 10-bit	0x03FF	TOP	BOTTOM
4	0	1	0	0	CTC	OCRnA	Immediate	MAX
5	0	1	0	1	Fast PWM, 8-bit	0x00FF	BOTTOM	TOP
6	0	1	1	0	Fast PWM, 9-bit	0x01FF	BOTTOM	TOP
7	0	1	1	1	Fast PWM, 10-bit	0x03FF	BOTTOM	TOP
8	1	0	0	0	PWM, Phase and Frequency Correct	ICRn	BOTTOM	BOTTOM
9	1	0	0	1	PWM, Phase and Frequency Correct	OCRnA	BOTTOM	BOTTOM
10	1	0	1	0	PWM, Phase Correct	ICRn	TOP	BOTTOM
11	1	0	1	1	PWM, Phase Correct	OCRnA	TOP	BOTTOM
12	1	1	0	0	CTC	ICRn	Immediate	MAX
13	1	1	0	1	(Reserved)	-	-	-
14	1	1	1	0	Fast PWM	ICRn	BOTTOM	TOP
15	1	1	1	1	Fast PWM	OCRnA	BOTTOM	TOP



# Example Code

```
.include "m2560def.inc"
.def temp=r16

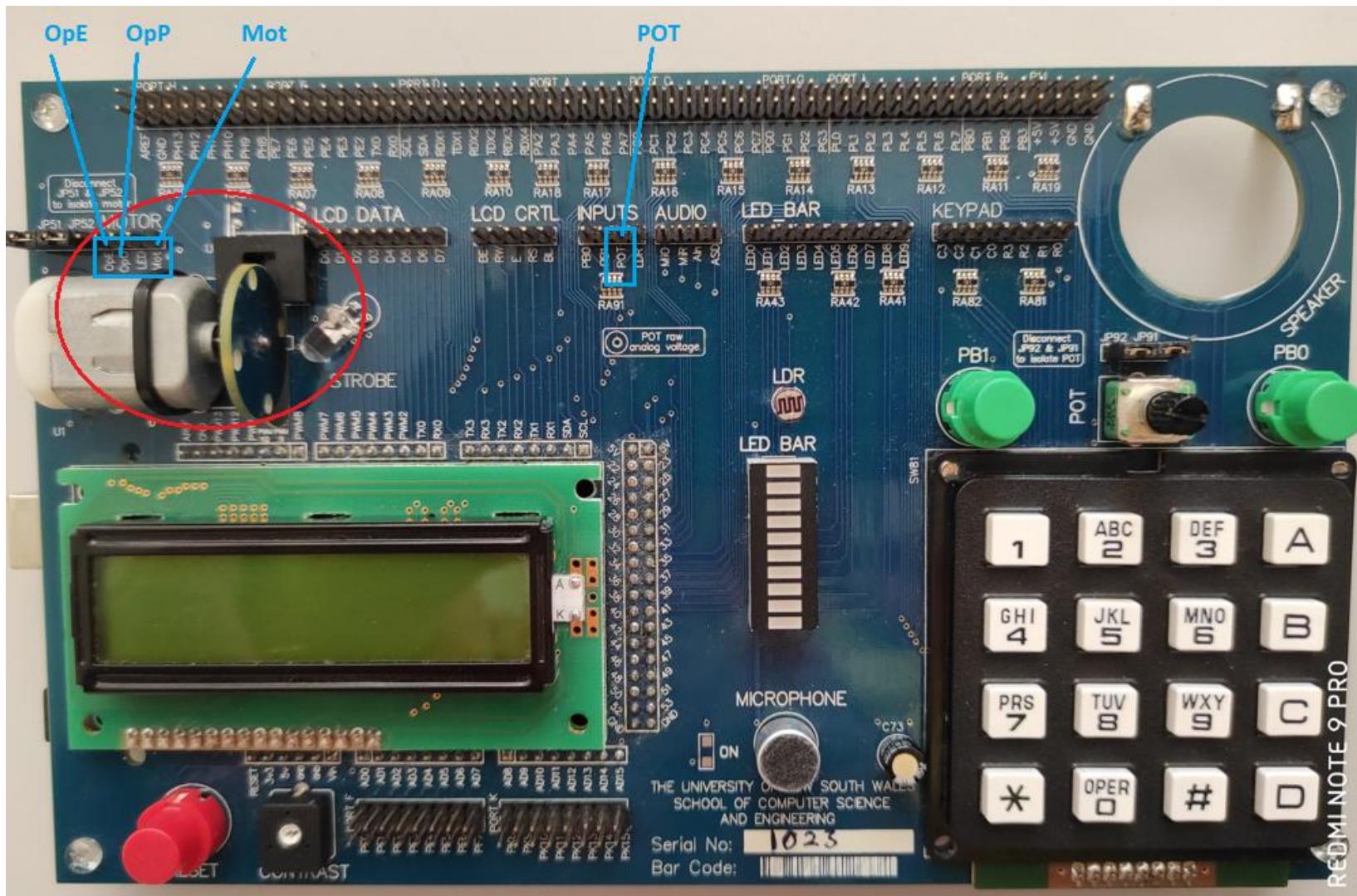
    ldi temp, 0b00001000
    sts DDRL, temp          ; Bit 3 will function as OC5A.

    clr temp                ; the value controls the PWM duty cycle
    sts OCR5AH, temp
    ldi temp, 0x4A
    sts OCR5AL, temp
                                ; Set Timer5 to Phase Correct PWM mode.
    ldi temp, (1 << CS50)   ; Set Timer clock frequency
    sts TCCR5B, temp.
    ldi temp, (1<< WGM50)|(1<<COM5A1)
    sts TCCR5A, temp
end:   rjmp end
```

# Exercise

- The motor on the lab board is a DC motor that is driven by an input signal. The higher the input voltage, the faster the motor spins. How to use the PWM signal generated from the code shown in the previous slide to drive the motor?
  - See Lab 4 spec for some explanation about DC motor

# DC Motor



# Reading Material

- Mega2560 Data Sheet.
  - PWM

# **Project Development & Software Design**

Lecturer: Annie Guo

# Lecture Overview

- Basic project development steps
- Some software design techniques
- Example

# Project Development Steps

- A project development basically consists of several stages:
  - Project definition
  - Design
  - Hardware implementation and Software development
  - Test

# Project Definition

- Includes:
  - Project requirement analysis
  - Development of a complete operational specification and coarse or macro-level block diagram
  - Possible project proposal
- Time spent: 10~15% of the total project time.

# Design

- Determines
  - Hardware design
    - Hardware components required and their connections
  - Overall system structure
    - Hardware+software
  - Software design
    - Software data structures and flowchart
      - Hardware and software interaction
      - Basic tasks and their interactions
- Takes 40-50% project time

# Software Design

- Consists of a number of steps:
  - Listing tasks to be performed
  - Prioritizing the tasks and scheduling tasks if required
    - often it is the case in the embedded system design
  - Designing interrupts for task scheduling if required
  - Creating a software flowchart
    - Key components and tasks
  - Writing code

# Test

- The system developed in the project is tested in accordance with the operational specification developed in the project definition stage.

# Example

- Lift Control Project
  - Refer to “Project Example” on the course website

# Example (cont.)

- Project definition
  - It basically has been done (in the course of this kind)
    - Hardware resources: AVR lab board
    - Operational specification
      - But you are allowed to add more, especially for enhancement

# Example (cont.)

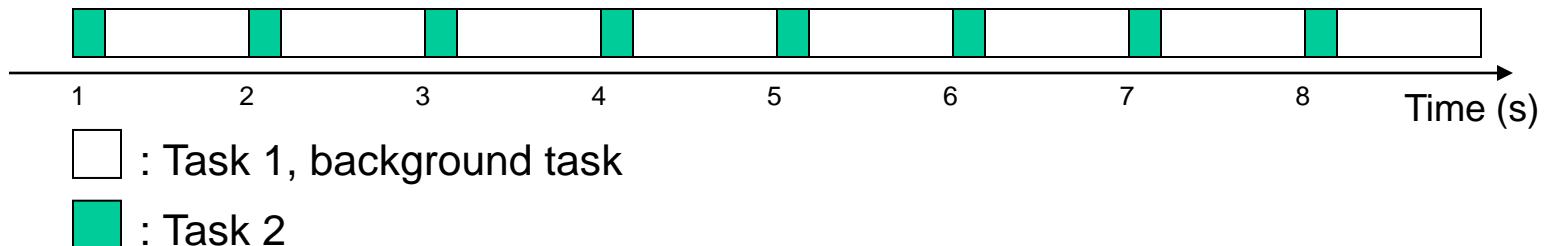
- Hardware design and implementation
  - The lab board is given, and the basic hardware design and implementation have been done for you.
  - What you need to do is to design how to use the available hardware resources
    - As specified
      - Using LED and LCD to display the status of the lift operation
      - Encoding input keys of the keyboard and push buttons to represent all possible input requests
    - Additionally
      - Using LCD and LED bars to display values for the debugging purpose

# Example (cont.)

- Software design
  - How many tasks are there?
    - Depending on your design and how you like your code organized.
      - One basic function → one task
      - A group of related functions → one task
    - E.g: two basic tasks (could be more)
      - Task 1: Getting the service requests
      - Task 2: Controlling the lift operations: moving up, moving down, stop

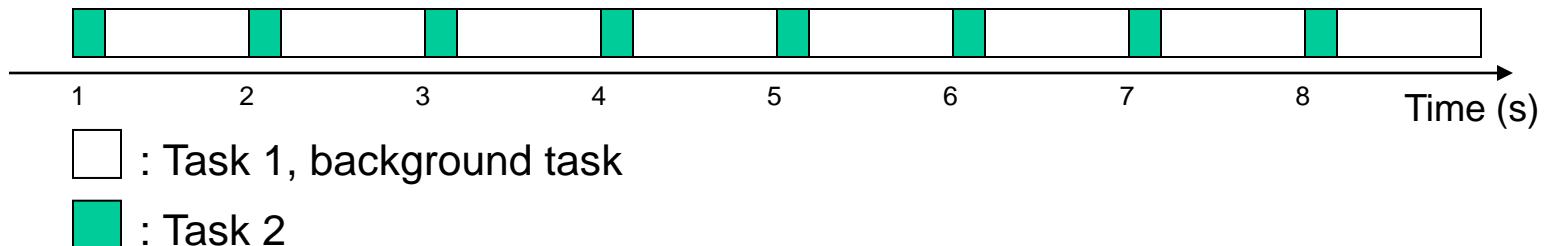
# Example (cont.)

- Software design
  - Are there any task priority issues?
    - Yes, Task 2 has a higher priority as specified in the project description – real time operation
      - If there are requests, the lift going from one floor to the next floor takes **two seconds**;
      - If there is a stop, during traveling, the stop takes **one second**.
    - a timing diagram for the task schedule



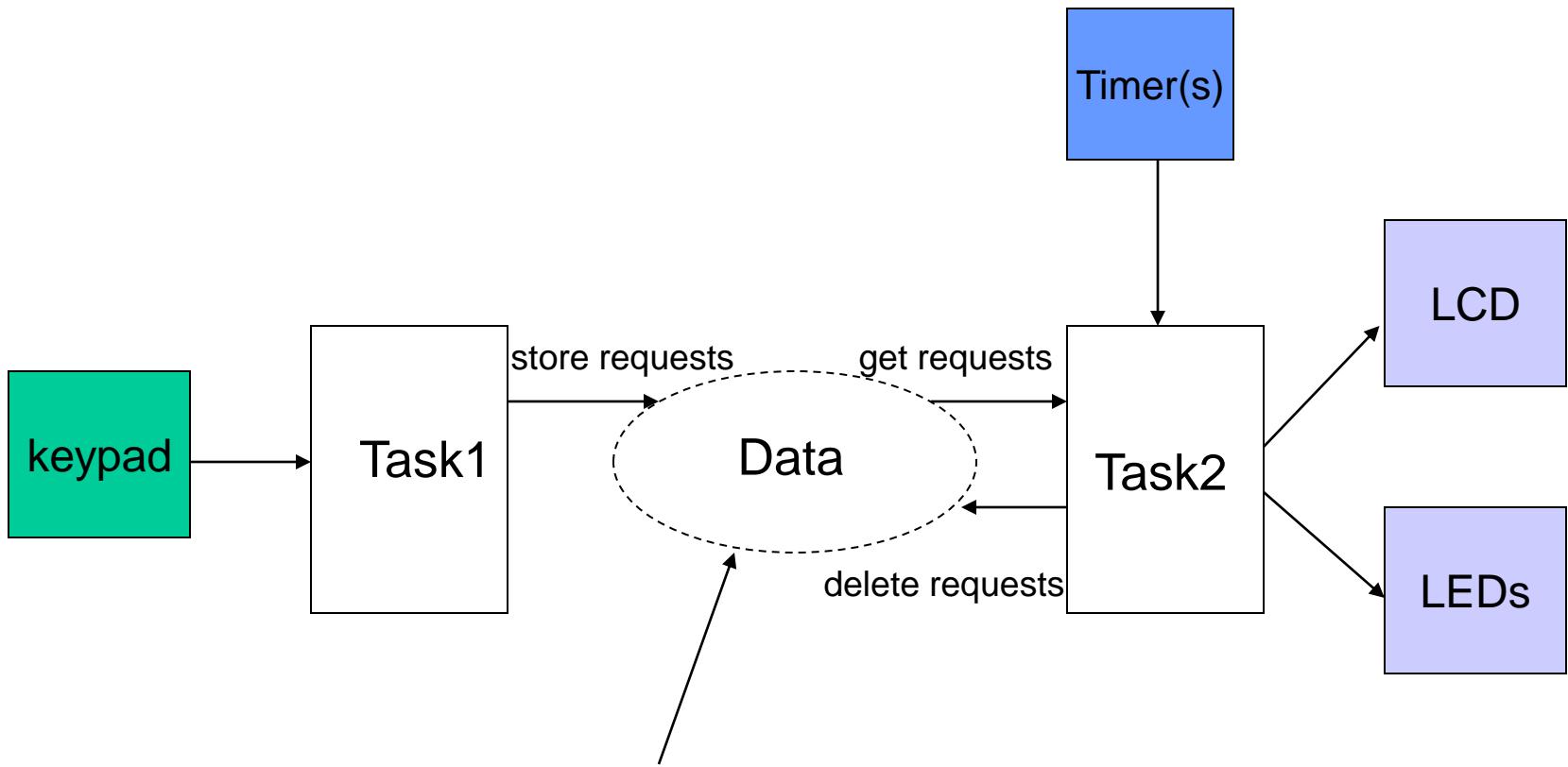
# Example (cont.)

- Software design
  - How to meet the timing requirement?
    - By software approach
      - Execution of a certain number of instructions to form a delay period
    - By the timer interrupts
      - Often preferred
        - » reliable, accurate, easy, and
        - » CPU efficient



# Example (cont.)

- Software design
  - How do the two tasks interact with each other?
    - Task 2 needs to know the requests from Task 1
      - Task 2 needs to deletes requests that have been processed.
  - How do the two tasks interact with input/output devices?
    - Task 1 keeps polling input requests from the keypad
    - Task 2 outputs the control for lift status display.
  - An overview block diagram may be helpful for a complicated design.
    - An example is given in the next slide



**How is data structured and stored?**

**How is the lift status represented?**

# Example (cont.)

- Software design
  - Data structure:
    - How is the information represented?
    - Is important
      - Can simplify your algorithms
      - Can affect the efficiency of your code
    - Should be carefully designed
  - Algorithms and flow diagrams
    - The high-level template of your code
    - A control flow diagram of the code is helpful
    - Verifying the control flow
      - Try to run some requests to see whether the control flow design can lead to the expected operations.

# Example (cont.)

- Software design
  - Set up the code template

```
; description ...

;include files
.include "m2560def.inc"

;define constants and variables here
; .equ directives
; .def directives
...
; set up interrupt vector table
jmp RESET
...
; the reset and other interrupt subroutines
RESET:

; main
; perform basic work including testing each function
;end_of_main

; normal functions
; func1
...

```

# Example (cont.)

- Software design
  - Write code – complete each section
    - Start from the definitions
    - Develop and test each module individually
      - Not necessarily implemented in “called functions”
      - Can be in other formats such as macros
    - The “display function” may be the first function to be completed since it may be used for testing other functions

# Example (cont.)

- Testing
  - Does it really work?
    - Compile and run your code on the lab board
      - Check for different operations
    - If it is not working ...
      - Check the wiring
      - Debug the code
      - Analyze the bugs
      - Work out the solutions
    - Until it is working
      - 

# **Microprocessors & Interfacing**

*Interrupt (II)*

Lecturer : Annie Guo

# Lecture Overview

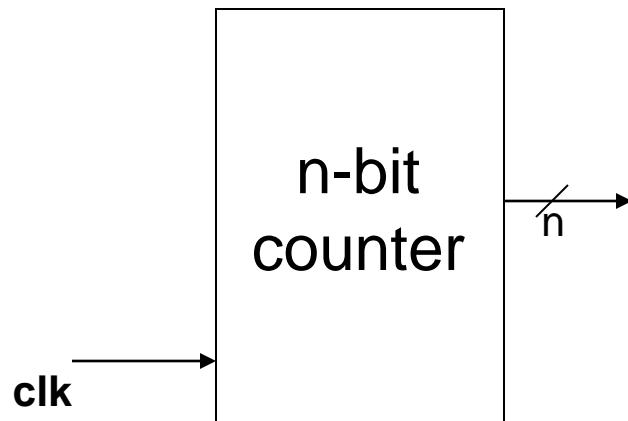
- Interrupts in AVR
  - Internal interrupt
    - Timer and timer generated interrupt

# Timer

- A timer is simply a binary counter
- Can be used to
  - Measure time duration
    - Determined by the count value and clock cycle time
  - Generate PWM signals
    - PWM: Pulse-Width Modulation
      - To be covered later
  - Schedule real-time tasks
    - Based on generated interrupts
  - Etc.

# Reference: Counter\* (1/2)

- A counter increases/decrease its value every clock cycle.
- Symbol



The slide and the next one were copied from  
<https://webcms3.cse.unsw.edu.au/COMP9032/22T3/resources/81819>

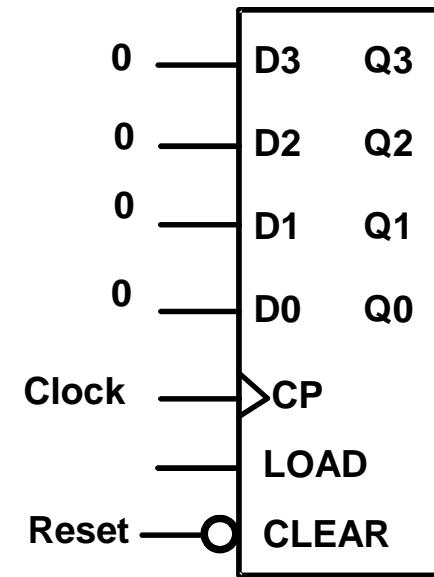
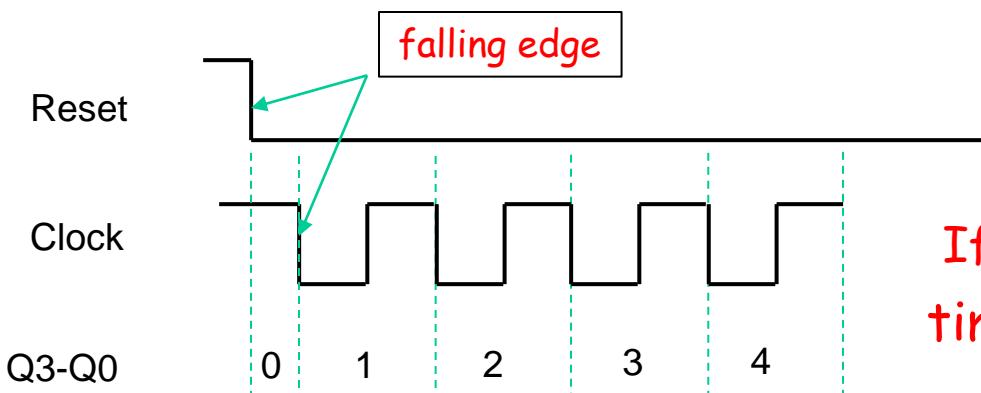
# Reference: Counter\* (2/2)

- 4-bit counter

The counter has

- a **synchronous load**
- an **asynchronous clear**

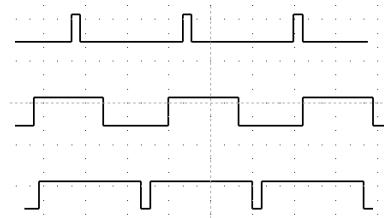
The counter counts through 0,  
1, 2, ..., 15, 0



If the counter size is large enough,  
 $\text{time} = (\text{countstop}-\text{countstart}) * \text{Tclk}$

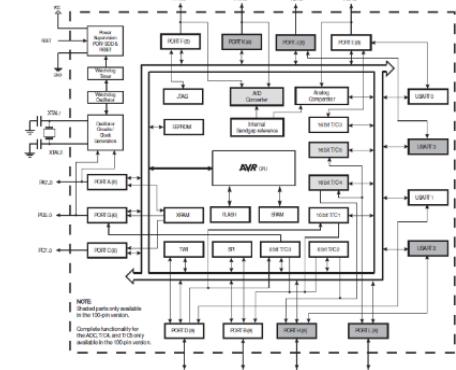
# Timer

- A timer is simply a binary counter
- Can be used to
  - Measure time duration
    - Determined by the count value and clock cycle time
  - Generate PWM signals
    - PWM: Pulse-Width Modulation
      - To be covered later
  - Schedule real-time tasks
    - Based on generated interrupts
  - Etc.

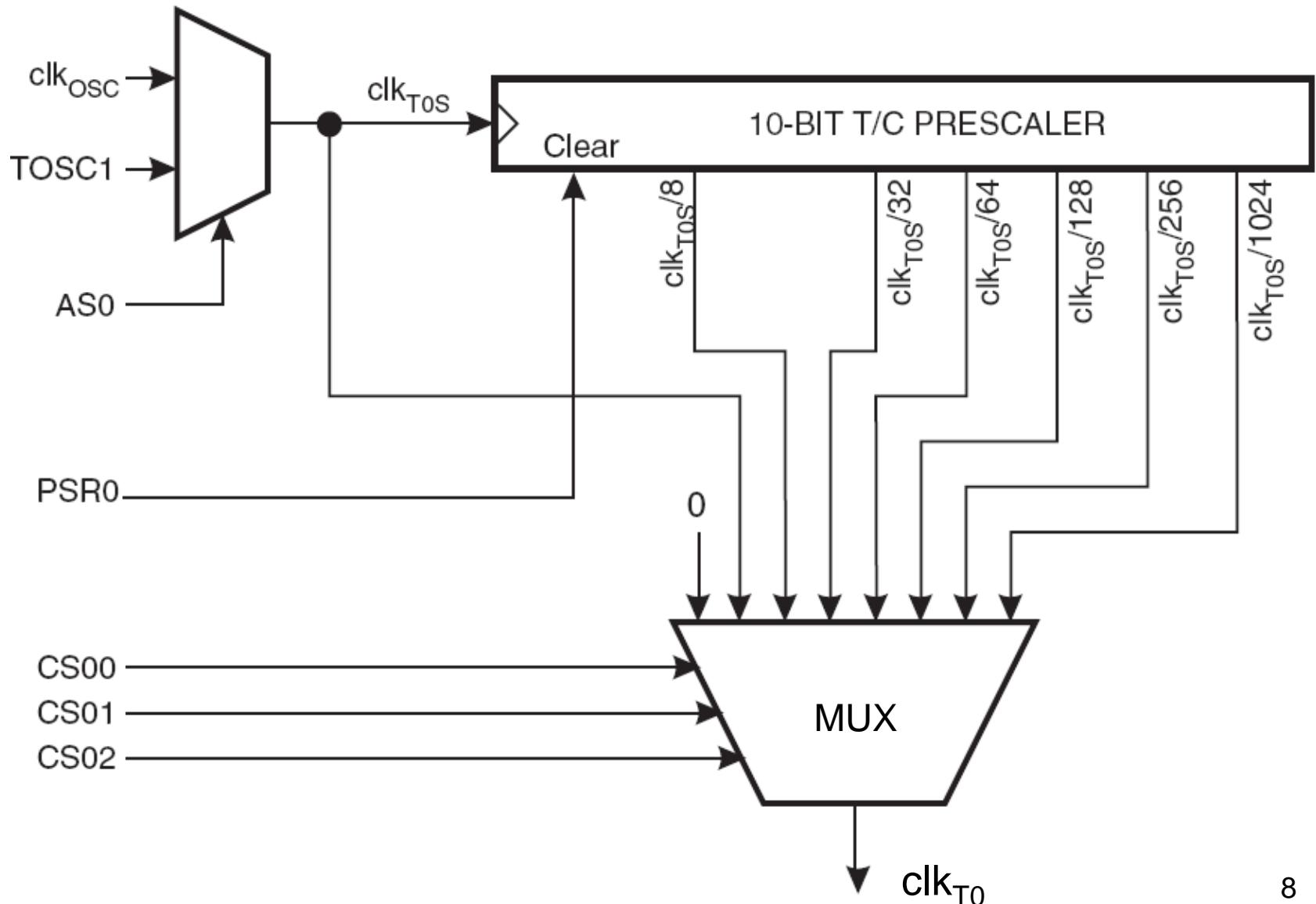


# Timers in AVR

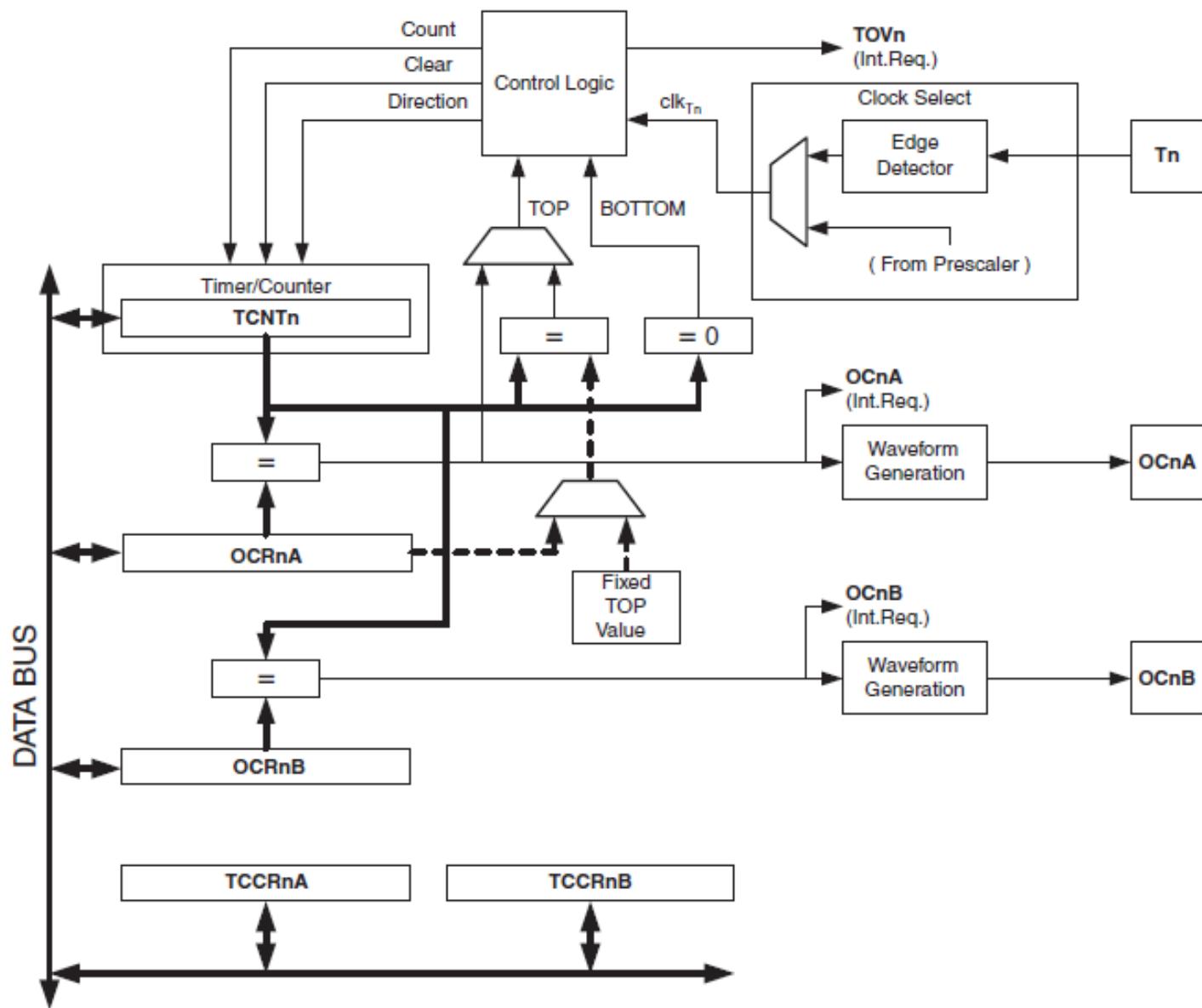
- In AVR, there are 8-bit and 16-bit timers.
  - Timer 0 and Timer 2
    - 8-bit counters
  - Timers 1, 3-5
    - 16-bit counters
- Timer 0 (named Timer0) is covered in the next slides
  - Similar designs can be found for other timers
    - See the Atmega2560 data sheet



# Timer Clock Source\*



# 8-bit Timer Block Diagram\*



# 8-bit Timer

- The counter can be initialized to
  - 0 (controlled by *clear*)
  - a number (controlled by *count signal*)
- Can count-up or count-down
  - controlled by *direction signal*
- Those control signals are generated by hardware control logic
  - The control logic is further controlled by programmer by
    - Writing control bits into TCCRnA/TCCRnB
      - E.g. TCCR0A/TCCR0B for Timer 0

# 8-bit Timer (cont.)

- Outputs from the timer
  - Overflow interrupt request bit
  - Output Compare interrupt request bits,
  - OCn bits: Output Compare bit for waveform generation
- The TIMSK register is used to enable the interrupts from the timer

# TIMSK0

- Timer/Counter Interrupt Mask Register for Timer0
  - Set TOIE0 (and I-bit in SREG) to enable the Overflow Interrupt
  - Set OCIE0A/B (and I bit in SREG) to enable Compare Match Interrupts

Bit (0x6E)	7	6	5	4	3	2	1	0
Read/Write	-	-	-	-	-	OCIE0B	OCIE0A	TOIE0
Initial Value	0	0	0	0	0	0	0	0

Control bits for Timer0

# TCCR0A/B

- Timer Counter Control Register

Bit	7	6	5	4	3	2	1	0	
0x24 (0x44)	COM0A1	COM0A0	COM0B1	COM0B0	-	-	WGM01	WGM00	TCCR0A
Read/Write	R/W	R/W	R/W	R/W	R	R	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Bit	7	6	5	4	3	2	1	0	
0x25 (0x45)	FOC0A	FOC0B	-	-	WGM02	CS02	CS01	CS00	TCCR0B
Read/Write	W	W	R	R	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

# TCCR0 Bit Description

- COM0xn/WGM0n:
  - Control the mode of the timer operation
    - The behavior of the Timer/Counter and the output is defined by the combination of the Waveform Generation mode (WGM02:00) and Compare Output mode (COM0x1:0) bits.
    - The simplest mode of operation is the **Normal Mode** (WGM02:00 =000). In this mode the counting direction is up. The counter rolls over when it passes its maximum 8-bit value (TOP = 0xFF) and then restarts from the bottom (0x00).
  - Refer to Mega2560 Data Sheet (pages 118~194) for details.

# TCCR0 Bit Description (cont.)

- Bit 2:0 in TCCR0B
  - Control the clock selection

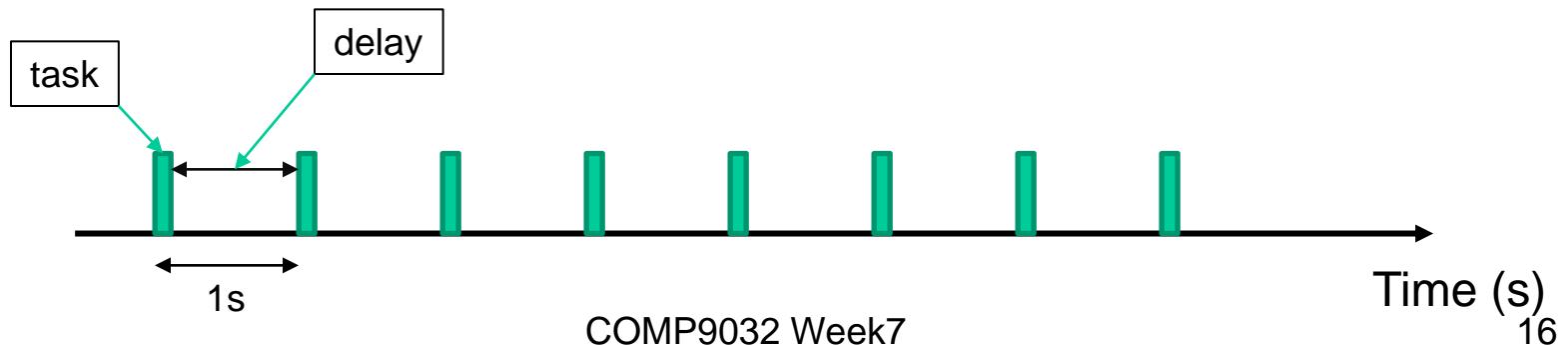
CS02	CS01	CS00	Description
0	0	0	No clock source (Timer/Counter stopped)
0	0	1	$\text{clk}_{\text{I/O}}$ (No prescaling)
0	1	0	$\text{clk}_{\text{I/O}}/8$ (From prescaler)
0	1	1	$\text{clk}_{\text{I/O}}/64$ (From prescaler)
1	0	0	$\text{clk}_{\text{I/O}}/256$ (From prescaler)
1	0	1	$\text{clk}_{\text{I/O}}/1024$ (From prescaler)
1	1	0	External clock source on T0 pin. Clock on falling edge
1	1	1	External clock source on T0 pin. Clock on rising edge

$T_{\text{clk}}$

Bit	7	6	5	4	3	2	1	0	
0x25 (0x45)	FOC0A	FOC0B	-	-	WGM02	CS02	CS01	CS00	TCCR0B
Read/Write	W	W	R	R	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

# Example 1

- Implement a scheduler that can execute a task every one second (for example display a value on LED).
- Generally, the function can be realized with
  - software design,
    - Software generates the delay
      - » With nop instructions
      - » With other tasks of known execution time
  - hardware design
    - Used here and solution is given in the next slides



# Example 1 Solution

- Use **8-bit** Timer0 to “count” the time
  - Let’s set Timer0 prescaler to 64 (i.e. the system source clock frequency is divided by 64)
    - The full counting duration (time-out) for the setting should therefore be
      - $256 \times (\text{clock period}) = 256 \times 64 / (16 \text{ MHz})$   
= 1024 us
        - » Namely, we set the Timer0 overflow interrupt that is to occur every 1024 us.
      - Note: clock period = 1/16 MHz (obtained from the data sheet) and the 8-bit counter can count 256 clock cycles.
    - For one second, there are
      - $1000000 / 1024 \approx 1000$  interrupts

# Example 1 Solution (cont.)

- In the assembly code,
  - Set Timer0 interrupt to occur every 1024 microseconds
    - as explained in the previous slide
  - Use a counter to count to 1000 interrupts for 1 second duration
  - To observe the 1 second time period, use the LED display that toggles every 1000 interrupts (i.e. one second)
    - a dummy task that flips display pattern
  - The code is given in the next slides

# Example 1

```
; This program uses Timer0 to schedule a task that occurs every second.  
; Every one second is generated by Timer0 interrupts.
```

```
.include "m2560def.inc"  
  
.equ PATTERN=0b11110000  
.def temp=r16  
.def leds = r17
```

```
; The macro clears a word (2 bytes) in the data memory for the counter stored in the memory  
; The parameter @0 is the memory address for that word
```

```
.macro Clear  
    ldi YL, low(@0)          ; load the memory address to Y  
    ldi YH, high(@0)  
    clr temp  
    st Y+, temp             ; clear the two bytes at @0 in SRAM  
    st Y, temp  
.endmacro
```

; continued

# Example 1

```
; continued
.dseg
SecondCounter:
    .byte 2                                ; Two-byte counter for counting the number of seconds.

TempCounter:
    .byte 2                                ; Temporary counter. Used to determine
                                                ; if one second has passed (i.e. when TempCounter=1000)

.cseg
.org 0x0000
    jmp RESET
    jmp DEFAULT                            ; No handling for IRQ0.
    jmp DEFAULT                            ; No handling for IRQ1.
    ;...
    ; insert other interrupt vectors

.org OVF0addr
    jmp Timer0OVF                         ; Jump to the interrupt handler for Timer0 overflow.
    ;...
    jmp DEFAULT                            ; other default service
                                                ; default service for all other interrupts.

DEFAULT: reti                            ; no service

                                                ; continued
```

# Example 1

; continued

RESET:

```
ser temp ; set Port C as output  
out DDRC, temp
```

```
rjmp main
```

; continued

# Example 1

; continued

```
Timer0OVF:          ; interrupt subroutine for Timer0
    ;in temp, SREG
    push temp           ; Prologue starts.
    push Yh             ; Save all conflict registers in the prologue.
    push YL
    push r25
    push r24           ; Prologue ends.
    ldi YL, low(TempCounter) ; Load the address of the temporary
    ldi YH, high(TempCounter) ; counter.
    ld r24, Y+          ; Load the value of the temporary counter.
    ld r25, Y
    adiw r25:r24, 1      ; Increase the temporary counter by one.

```

; continued

# Example 1

; continued

```
Timer0OVF:          ; interrupt subroutine for Timer0
    ; ... save SREG
    push temp           ; Prologue starts.
    push Yh             ; Save all conflict registers in the prologue.
    push YL
    push r25
    push r24           ; Prologue ends.
    ldi YL, low(TempCounter) ; Load the address of the temporary
    ldi YH, high(TempCounter) ; counter.
    ld r24, Y+          ; Load the value of the temporary counter.
    ld r25, Y
    adiw r25:r24, 1     ; Increase the temporary counter by one.
                        ; continued
```

# Example 1

; continued

```
cpi r24, low(1000)          ; Check if (r25:r24)=1000
brne NotSecond
cpi r25, high(1000)
brne NotSecond
com leds
out PORTC, leds
Clear TempCounter           ; Reset the temporary counter.

ldi YL, low(SecondCounter) ; Load the address of the second
ldi YH, high(SecondCounter); counter.
ld r24, Y+                  ; Load the value of the second counter.
ld r25, Y
adiw r25:r24, 1             ; Increase the second counter by one.
```

; continued

# Example 1

; continued

st Y, r25

; Store the value of the second counter.

st -Y, r24

rjmp endif

NotSecond:

st Y, r25

; Store the value of the temporary counter.

st -Y, r24

endif:

pop r24

; Epilogue starts;

pop r25

; Restore all conflict registers from the stack.

pop YL

pop YH

pop temp

;out SREG, temp

reti

; Return from the interrupt.

; continued

# Example 1

; continued

main:

```
ldi leds, 0xff          ; Init pattern displayed
out PORTC, leds
ldi leds, PATTERN
Clear TempCounter       ; Initialize the temporary counter to 0
Clear SecondCounter    ; Initialize the second counter to 0
ldi temp, 0b00000000
out TCCR0A, temp
ldi temp, 0b00000011
out TCCR0B, temp        ; Prescaler value=64, counting 1024 us
ldi temp, 1<<TOIE0
sts TIMSK0, temp        ; T/C0 interrupt enable
sei                      ; Enable global interrupt
```

loop:

```
rjmp loop              ; loop forever
```

# Reading Material

- Chapter 10: Interrupts and Real-Time Events.  
Microcontrollers and Microcomputers by  
Fredrick M. Cady.
- Mega2560 Data Sheet.
  - External Interrupts.
  - Timer0

# Homework

1. An underground oil tank monitor system has the following functions:
  1. `read()`: to read the tank oil level
  2. `display()`: to display the oil level
  3. `main()`: to process a few of basic tasks: if the oil level is below the low limit, do something; if oil level is over the high limit, do something else; and other routine work.

It is required that the display should be updated every 1 minute, reading should be done every 10 seconds. Assume `read()` and `display()` take 1 ms and 5 ms, respectively. Design a scheduling controller for those functions so that the above requirements can be met, and the design leads to an easy assembly code implementation.

# Microprocessors & Interfacing

*Interrupt (I)*

Lecturer : Annie Guo

# Lecture Overview

- Introduction to Interrupt
- Interrupts in AVR
  - Interrupt vector table
  - Interrupt service routine
  - System reset
  - External interrupt

# CPU Interaction with I/O

Two typical approaches:

- Polling
  - Software queries I/O devices
  - No extra hardware needed
  - Not efficient
    - It takes processor cycles to query a device even if it does not need any service.
- Interrupt
  - I/O devices generate signals to request services of CPU
  - Special hardware is required to implement interrupt
  - Efficient
    - A signal is generated only if the I/O device needs services from CPU.

# Interrupt System

- An interrupt system implements interrupt services
- It basically performs three tasks:
  - Detecting interrupt event
  - Responding to interrupt
  - Resuming normal programmed task

# Detect Interrupt Event

- Interrupt event
  - Associated with interrupt signal
    - In different forms, including signal levels and edges.
  - Can be multiple and simultaneous
    - There may be many sources to generate an interrupt;
    - A number of interrupts can be generated at the same time.
- Approaches are required to
  - Identify an interrupt event among multiple sources
  - Determine which interrupt to serve if there are multiple simultaneous interrupts

# Respond to Interrupt

- Handling interrupt
  - Wait for the current instruction to finish
  - Acknowledge the interrupting device
  - Branch to the correct ***interrupt service routine*** (interrupt handler) to service the interrupting device.

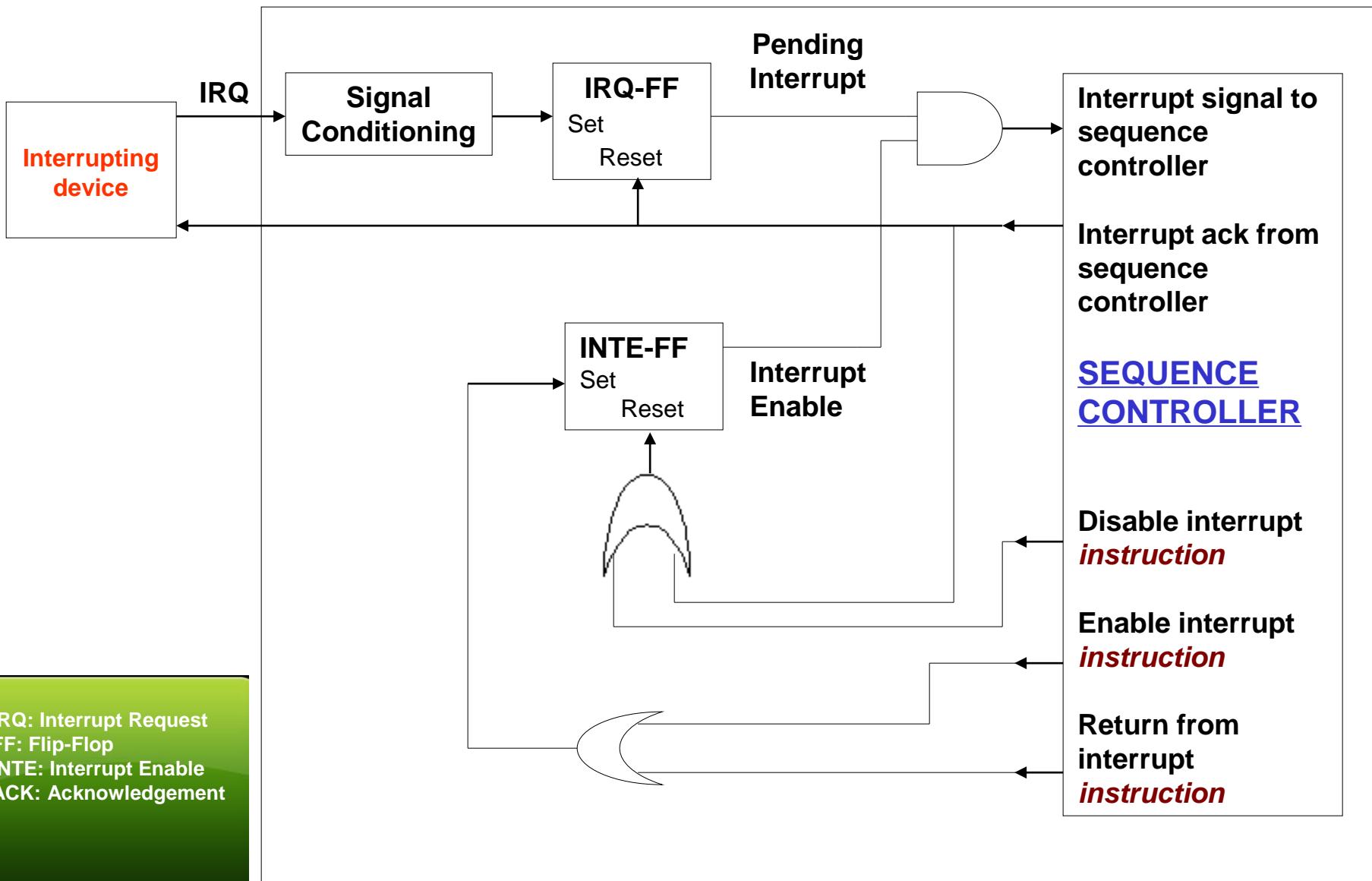
# Resume Normal Task

- Return to the interrupted program at the point it was interrupted.

# Interrupt Process Control

- Interrupts can be enabled or disabled
  - Special cases exist, for example *reset*
    - Will be covered later
- Can be controlled in two ways:
  - Software control
    - Allows code to enable and disable selected or all interrupts.
  - Hardware control
    - E.g. disable further interrupts while an interrupt is being serviced

# Interrupt Detection and Acknowledgement Hardware\*



# Interrupt Detection and Acknowledgement\*

- An interrupt request (IRQ) may occur at any time.
  - It can be represented by signal's rising or falling edges, or high or low levels.
- Signal Conditioning circuit detects the request.
- Interrupt Request Flip-Flop (IRQ-FF) holds the interrupt request until it is acknowledged.
  - When IRQ-FF is set, it generates a pending interrupt signal that goes towards the Sequence Controller.
  - IRQ-FF is reset when the controller acknowledges the interrupt with INTA signal.

# Interrupt Detection and Acknowledgement (cont.)\*

- An interrupt can be enabled and disabled by software instructions, which is supported by the hardware Interrupt Enable Flip-Flop (INTE-FF).
- When INTE-FF is set, the interrupt is enabled and the pending interrupt is passed through the AND gate to the sequence controller.
- INTE-FF is reset in the following cases:
  - the controller acknowledges the interrupt.
  - the controller is reset.
  - the Disable Interrupt instruction is executed.

# Interrupt Detection and Acknowledgement (cont.)\*

- An interrupt acknowledge signal is generated by the controller when execution of the current instruction finished, and the controller has detected the IRQ.
  - This resets IRQ-FF and INTE-FF and signals the interrupting device that CPU is ready to execute the interrupting device routine.
- At the end of the interrupt service routine, CPU executes a return-from-interrupt instruction.
  - Part of this instruction's job is to set INTE-FF to re-enable interrupts.
- Nested interrupts can happen if the INTE-FF is set during an interrupt service routine
  - An interrupt can therefore interrupt an existing interrupt.

# Multiple Sources of Interrupt

- To handle multiple sources of an interrupt, the interrupt system must
  - Identify which device has generated the IRQ.
    - Using polling approach
    - Using vectoring approach
  - Resolve simultaneous interrupt requests
    - using prioritization schemes

# Multiple Interrupt Masking

- Masking enables some interrupts and disables others
- Individual disable/enable bit is assigned to each interrupting source.

# Transferring Control to Interrupt Service Routine

- **Hardware** needs to save the return address.
  - Most processors save the return address on the stack
- Hardware may also save some registers such as program status register.
  - AVR does not save any register. It is the programmer's responsibility to save program status register and conflict registers.
- The delay from the time the pending IRQ is generated to the time the Interrupt Service Routine (ISR) starts to execute is called *interrupt latency*.

# Interrupt Service Routine

- A section of code to be executed when the corresponding interrupt is responded by CPU.
- Interrupt service routine is a special function, therefore can be constructed with three parts:
  - Prologue:
    - Code mainly for saving conflict registers
  - Body:
    - Code for doing the required task
  - Epilogue:
    - Code for restoring conflict registers
    - The last instruction is the return-from-interrupt instruction.

# Software Interrupt

- Software interrupt is the interrupt generated by software without a hardware-generated-IRQ.
- Software interrupt is typically used to implement system calls in OS.
- Some processors have a special machine instruction to generate software interrupt.
  - E.g. SWI in ARM.
- AVR does NOT provide a software interrupt instruction.
  - Programmers can use External Interrupts to implement software interrupts.

# Reset Interrupt

- Reset is a special interrupt available in most processors (including AVR).
- **It is not maskable.**
- Its service function mainly sets the processor system to the initial state (hence called reset interrupt).
  - No need to deal with conflict registers.

# AVR Interrupt

- Interrupts in AVR basically can be divided into
  - internal interrupts
  - external interrupts
- Each interrupt has a dedicated interrupt vector
  - To be discussed
- Hardware is used to detect interrupt

# AVR Interrupt (cont.)

- To enable an interrupt, two control bits must be set
  - the Global Interrupt Enable bit (I bit) in the Status Register, SREG
    - Using *sei* instruction
  - the enable bit for that interrupt
- To disable all **maskable** interrupts, reset the I bit in SREG
  - Using *cli* instruction
- Priority of interrupts is used to handle multiple simultaneous interrupts
  - To be discussed

# Set Global Interrupt Flag

- Syntax:            ***sei***
- Operands:        none
- Operation:       $I \leftarrow 1$ 
  - Set the global interrupt flag ( $I$ ) in SREG. The instruction following ***sei*** will be executed before any pending interrupts.
- Words:            1
- Cycles:           1
- Example:
  - sei***       ; set global interrupt enable
  - sleep***      ; enter sleep state, waiting for an interrupt

# Clear Global Interrupt Flag

- Syntax:            ***cli***
- Operands:        none
- Operation:       $I \leftarrow 0$ 
  - Clear the Global interrupt flag in SREG. Interrupts will be immediately disabled.
- Words:            1
- Cycles:           1
- Example:

```
in r18, SREG          ; store SREG value
cli                  ; disable interrupts
; do something very important here
out SREG, r18        ; restore SREG value
```

# Interrupt Response Time

- Minimum 4 clock cycles
  - For saving the Program Counter (2 clock cycles)
  - For jumping to the interrupt routine (2 clock cycles)

# Interrupt Vectors

- Each interrupt has a 4-byte (2-word) **interrupt vector**, containing an instruction to be executed after CPU has accepted the interrupt.
- The lowest address space in the program memory is, by default, defined as the section for Interrupt Vectors.
- The priority of an interrupt is based on the position of its vector in the program memory
  - The lower the address, the higher the priority level
- RESET has the highest priority

# Interrupt Vectors in Mega2560

Vector No.	Program Address <sup>(2)</sup>	Source	Interrupt Definition
1	\$0000 <sup>(1)</sup>	RESET	External Pin, Power-on Reset, Brown-out Reset, Watchdog Reset, and JTAG AVR Reset
2	\$0002	INT0	External Interrupt Request 0
3	\$0004	INT1	External Interrupt Request 1
4	\$0006	INT2	External Interrupt Request 2
5	\$0008	INT3	External Interrupt Request 3
6	\$000A	INT4	External Interrupt Request 4
7	\$000C	INT5	External Interrupt Request 5
8	\$000E	INT6	External Interrupt Request 6
9	\$0010	INT7	External Interrupt Request 7
10	\$0012	PCINT0	Pin Change Interrupt Request 0
11	\$0014	PCINT1	Pin Change Interrupt Request 1
12	\$0016 <sup>(3)</sup>	PCINT2	Pin Change Interrupt Request 2
13	\$0018	WDT	Watchdog Time-out Interrupt
14	\$001A	TIMER2 COMPA	Timer/Counter2 Compare Match A
15	\$001C	TIMER2 COMPB	Timer/Counter2 Compare Match B

# Interrupt Vectors in Mega2560

16	\$001E	TIMER2 OVF	Timer/Counter2 Overflow
17	\$0020	TIMER1 CAPT	Timer/Counter1 Capture Event
18	\$0022	TIMER1 COMPA	Timer/Counter1 Compare Match A
19	\$0024	TIMER1 COMPB	Timer/Counter1 Compare Match B
20	\$0026	TIMER1 COMPC	Timer/Counter1 Compare Match C
21	\$0028	TIMER1 OVF	Timer/Counter1 Overflow
22	\$002A	TIMER0 COMPA	Timer/Counter0 Compare Match A
23	\$002C	TIMER0 COMPB	Timer/Counter0 Compare match B
24	\$002E	TIMER0 OVF	Timer/Counter0 Overflow
25	\$0030	SPI, STC	SPI Serial Transfer Complete
26	\$0032	USART0 RX	USART0 Rx Complete
27	\$0034	USART0 UDRE	USART0 Data Register Empty
28	\$0036	USART0 TX	USART0 Tx Complete
29	\$0038	ANALOG COMP	Analog Comparator

# Interrupt Vectors in Mega2560

30	\$003A	ADC	ADC Conversion Complete
31	\$003C	EE READY	EEPROM Ready
32	\$003E	TIMER3 CAPT	Timer/Counter3 Capture Event
33	\$0040	TIMER3 COMPA	Timer/Counter3 Compare Match A
34	\$0042	TIMER3 COMPB	Timer/Counter3 Compare Match B
35	\$0044	TIMER3 COMPC	Timer/Counter3 Compare Match C
36	\$0046	TIMER3 OVF	Timer/Counter3 Overflow
37	\$0048	USART1 RX	USART1 Rx Complete
38	\$004A	USART1 UDRE	USART1 Data Register Empty
39	\$004C	USART1 TX	USART1 Tx Complete
40	\$004E	TWI	2-wire Serial Interface
41	\$0050	SPM READY	Store Program Memory Ready
42	\$0052 <sup>(3)</sup>	TIMER4 CAPT	Timer/Counter4 Capture Event
43	\$0054	TIMER4 COMPA	Timer/Counter4 Compare Match A
44	\$0056	TIMER4 COMPB	Timer/Counter4 Compare Match B
45	\$0058	TIMER4 COMPC	Timer/Counter4 Compare Match C

# Interrupt Vectors in Mega2560

46	\$005A	TIMER4 OVF	Timer/Counter4 Overflow
47	\$005C <sup>(3)</sup>	TIMER5 CAPT	Timer/Counter5 Capture Event
48	\$005E	TIMER5 COMPA	Timer/Counter5 Compare Match A
49	\$0060	TIMER5 COMPB	Timer/Counter5 Compare Match B
50	\$0062	TIMER5 COMPC	Timer/Counter5 Compare Match C
51	\$0064	TIMER5 OVF	Timer/Counter5 Overflow
52	\$0066 <sup>(3)</sup>	USART2 RX	USART2 Rx Complete
53	\$0068 <sup>(3)</sup>	USART2 UDRE	USART2 Data Register Empty
54	\$006A <sup>(3)</sup>	USART2 TX	USART2 Tx Complete
55	\$006C <sup>(3)</sup>	USART3 RX	USART3 Rx Complete
56	\$006E <sup>(3))</sup>	USART3 UDRE	USART3 Data Register Empty
57	\$0070 <sup>(3)</sup>	USART3 TX	USART3 Tx Complete

# Interrupt Process

- When an interrupt service starts,
  - the Global Interrupt Enable I-bit is cleared (0) and all interrupts are disabled.
    - The user **Software** can set the I-bit in SREG to allow nested interrupts
- When the processor exits from an interrupt,
  - the Global Interrupt Enable I-bit is set (1)
    - via execution of the *reti* instruction
  - the execution returns to the main program and executes one more instruction before any pending interrupt is served.
    - The Reset interrupt is an exception

# Initialization of Interrupt Vector Table (IVT) in Mega2560

- Typically, an **interrupt vector** contains
  - a branch instruction (*jmp* or *rjmp*) that branches to the first instruction of the interrupt service routine, or
  - simply *reti* (return-from-interrupt) if you don't need to handle this interrupt.

# Example of IVT Initialization in Mega2560

```
.include "m2560def.inc"

.cseg

.org 0x0000

;first vector ----

    rjmp RESET           ; Jump to the start of the Reset interrupt service routine
                          ; Relative jump is used if RESET is not far

    nop                 ; to make the vector 4 bytes.

;second vector ----

    jmp IRQ0            ; Long jump is used assuming IRQ0 is very far away

;third vector ----

    reti                ; Return to the interrupted point (no handling for this interrupt).

; ... some code here

RESET:                  ; The interrupt service routine for RESET starts here.

; ... some code here

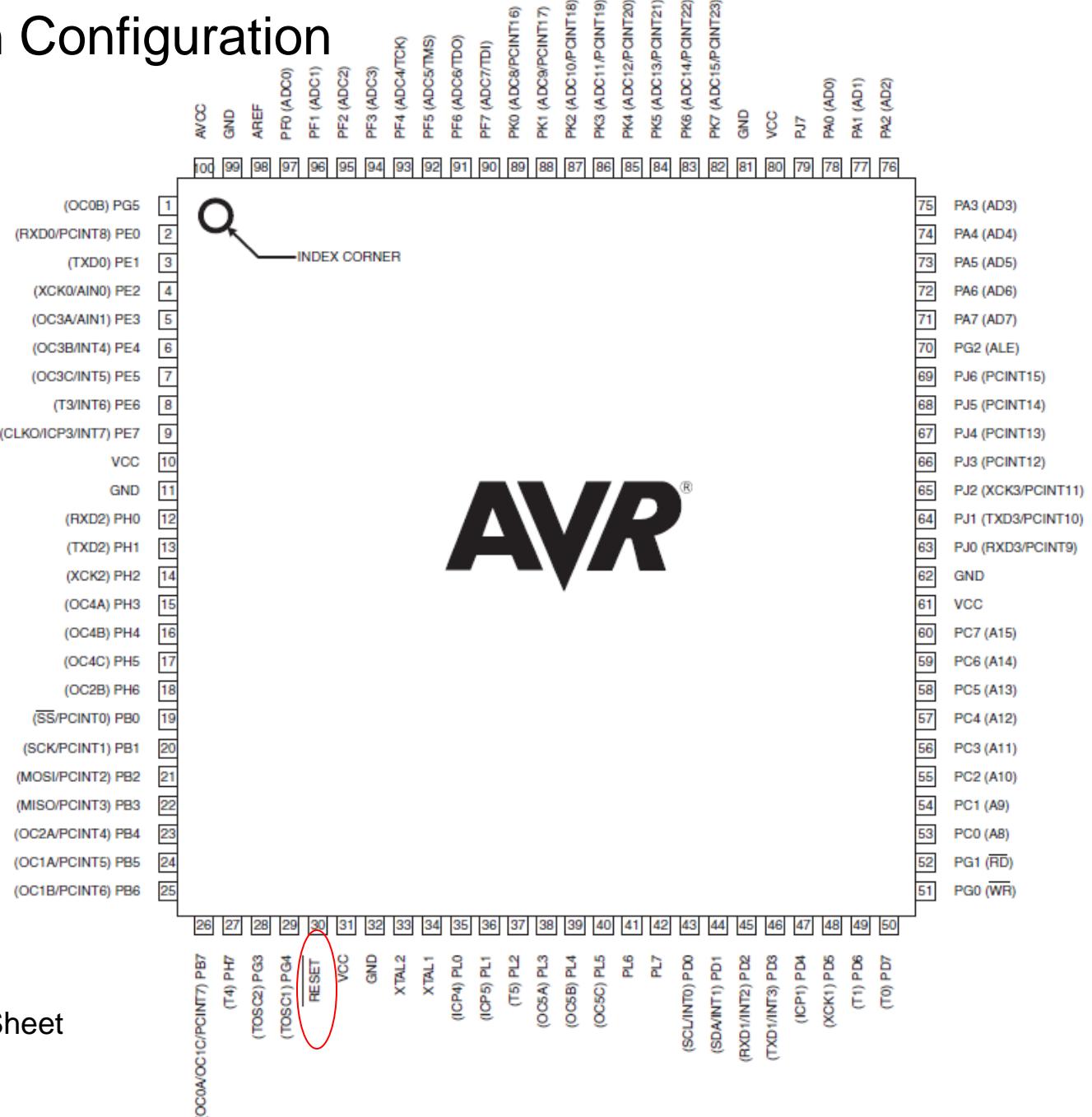
IRQ0:                  ; The interrupt service routine for IRQ0 starts here.

; ... some code here
```

# RESET in Mega2560

- ATmega2560 has five sources of reset:
  - Power-on Reset.
    - The MCU is reset when the supply voltage is below the Power-on Reset threshold ( $V_{POT}$ ).
  - External Reset.
    - The MCU is reset when a low level is present on the RESET pin for longer than the minimum pulse length.
  - Watchdog Reset.
    - The MCU is reset when the Watchdog Timer period expires, and the Watchdog is enabled.
  - Etc.

# Atmega2560 Pin Configuration



Source: Atmega2560 Data Sheet

# External Interrupts

- The external interrupts are triggered through the INT7:0 pins.
  - If enabled, the interrupts can be triggered even if the INT7:0 pins are configured as outputs
    - This feature provides a way of generating a software interrupt.
  - Can be triggered by a falling or rising edge or a logic level
    - Specified in External Interrupt Control Register
      - EICRA (for INT3:0)
      - EICRB (for INT7:4)

# External Interrupts (cont.)

- To enable an external interrupt, two bits must be set
  - 1 bit in SREG
  - INTx bit in register EIMSK
    - E.g. bit INT0 in EIMSK for interrupt INT0
- To generate an external interrupt, the following requirements must be met:
  - The interrupt must be enabled
  - The associated external pin must have a designed signal produced.

# EIMSK

- External Interrupt Mask Register
  - A bit is set to enable the related interrupt

Bit	7	6	5	4	3	2	1	0	
0x1D (0x3D)	INT7	INT6	INT5	INT4	INT3	INT2	INT1	INT0	EIMSK
Read/Write	R/W								
Initial Value	0	0	0	0	0	0	0	0	

NOTE: All tables in the notes are copied from the ATmega2560 data sheet

# EICRA

- External Interrupt Control Register A
  - For INT0-3
  - Defines the type of signal that activates the external interrupt
    - on the rising or falling edge or level sensed

Bit	7	6	5	4	3	2	1	0
(0x69)	INT3	INT2	INT1	INT0				
Read/Write	ISC31	ISC30	ISC21	ISC20	ISC11	ISC10	ISC01	ISC00
Initial Value	0	0	0	0	0	0	0	0

ISCn1	ISCn0	Description
0	0	The low level of INTn generates an interrupt request
0	1	Any edge of INTn generates asynchronously an interrupt request
1	0	The falling edge of INTn generates asynchronously an interrupt request
1	1	The rising edge of INTn generates asynchronously an interrupt request

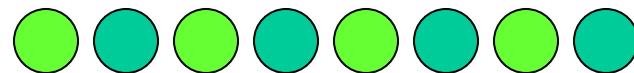
# EIFR

- Interrupt flag register
  - A bit in the register is set when an edge-triggered interrupt is enabled and an event on the related INT pin happens.

Bit	7	6	5	4	3	2	1	0
0x1D (0x3D)	INT7	INT6	INT5	INT4	INT3	INT2	INT1	INT0
Read/Write	R/W							
Initial Value	0	0	0	0	0	0	0	0

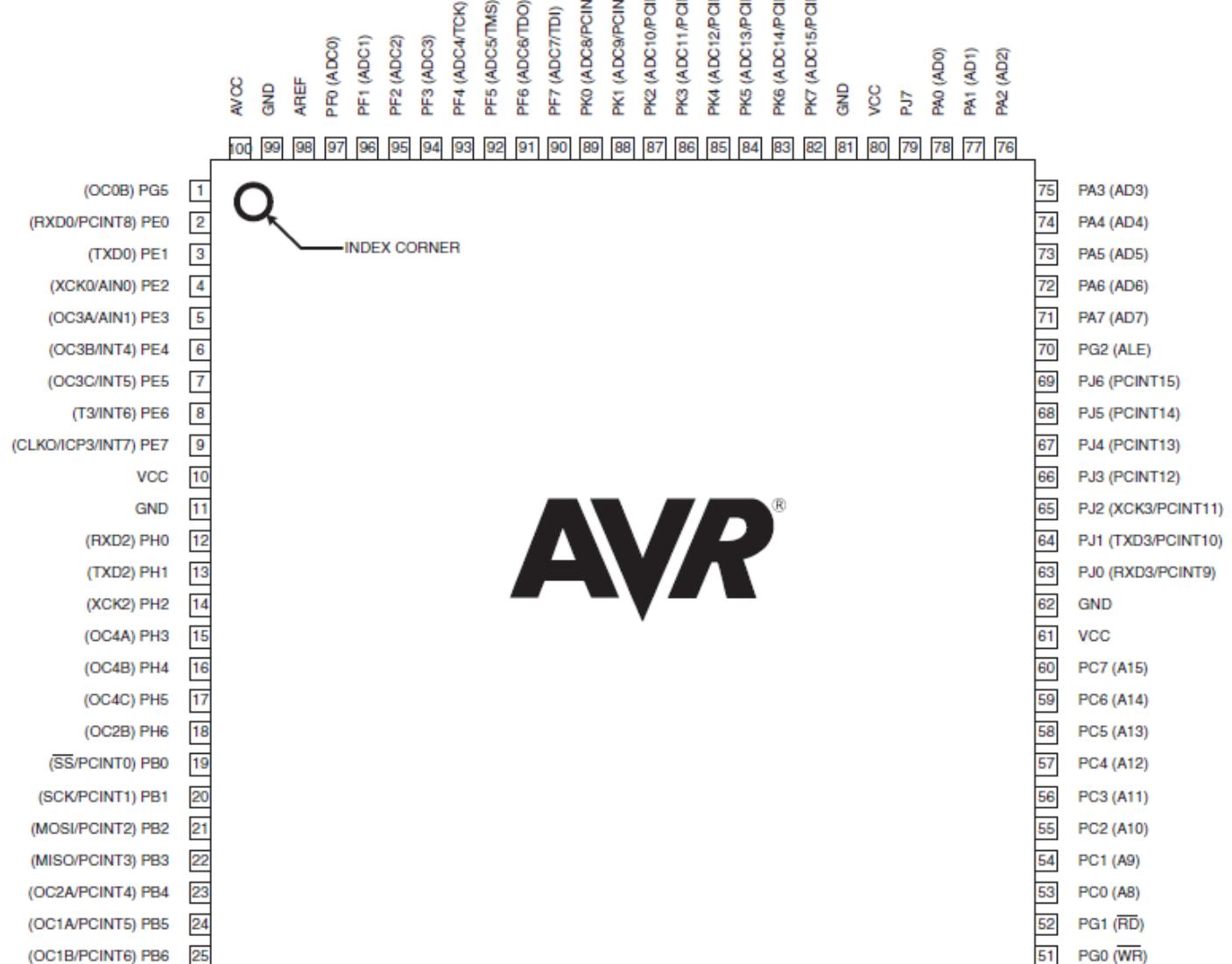
# Example 1

- Design a system, where the state of LEDs toggles under the control of the user, and the number of toggles is counted.



# Example 1 (solution)

- Use an external interrupt
  - Connect the external interrupt pin to a push button
    - INT0 to be used. The related pin is Port D Bit 0
  - When the button is pressed, an interrupt is generated
- In the assembly code
  - Set up the interrupt
    - Create the interrupt vector
    - Enable the interrupt
  - Write a service routine for this interrupt
    - Change the display pattern
    - Write the pattern to the port connected to LEDs
    - Increase the toggle count



**AVR®**

# Code for Example 1

```
.include "m2560def.inc"

.def    temp = r16
.def    output = r17
.def    count = r18          ; count the number of interrupts
.equ   PATTERN = 0b01010101

; set up interrupt vectors
.org   jmp RESET            ; vector for the RESET interrupt
      INT0addr           ; location for INT0 vector, defined in m2560def.inc
      jmp EXT_INT0         ; INT0 vector

; service routine for the reset interrupt
RESET:
      ser temp             ; set Port C as output
      out DDRC, temp
      out PORTC, temp
      ldi output, PATTERN
```

# Code for Example 1 (cont.)

```
ldi temp, (2 << ISC00) ; set INT0 as falling edge triggered interrupt
sts EICRA, temp

in temp, EIMSK           ; enable INT0
ori temp, (1<<INT0)
out EIMSK, temp

sei                      ; enable Global Interrupt
jmp main

EXT_INTERRUPT:
push temp                ; save register
in temp, SREG             ; save SREG
push temp

com output               ; flip the pattern
out PORTC, output
inc count

pop temp                 ; restore SREG
out SREG, temp            ; restore register
pop temp
reti
```

# Code for Example 1 (cont.)

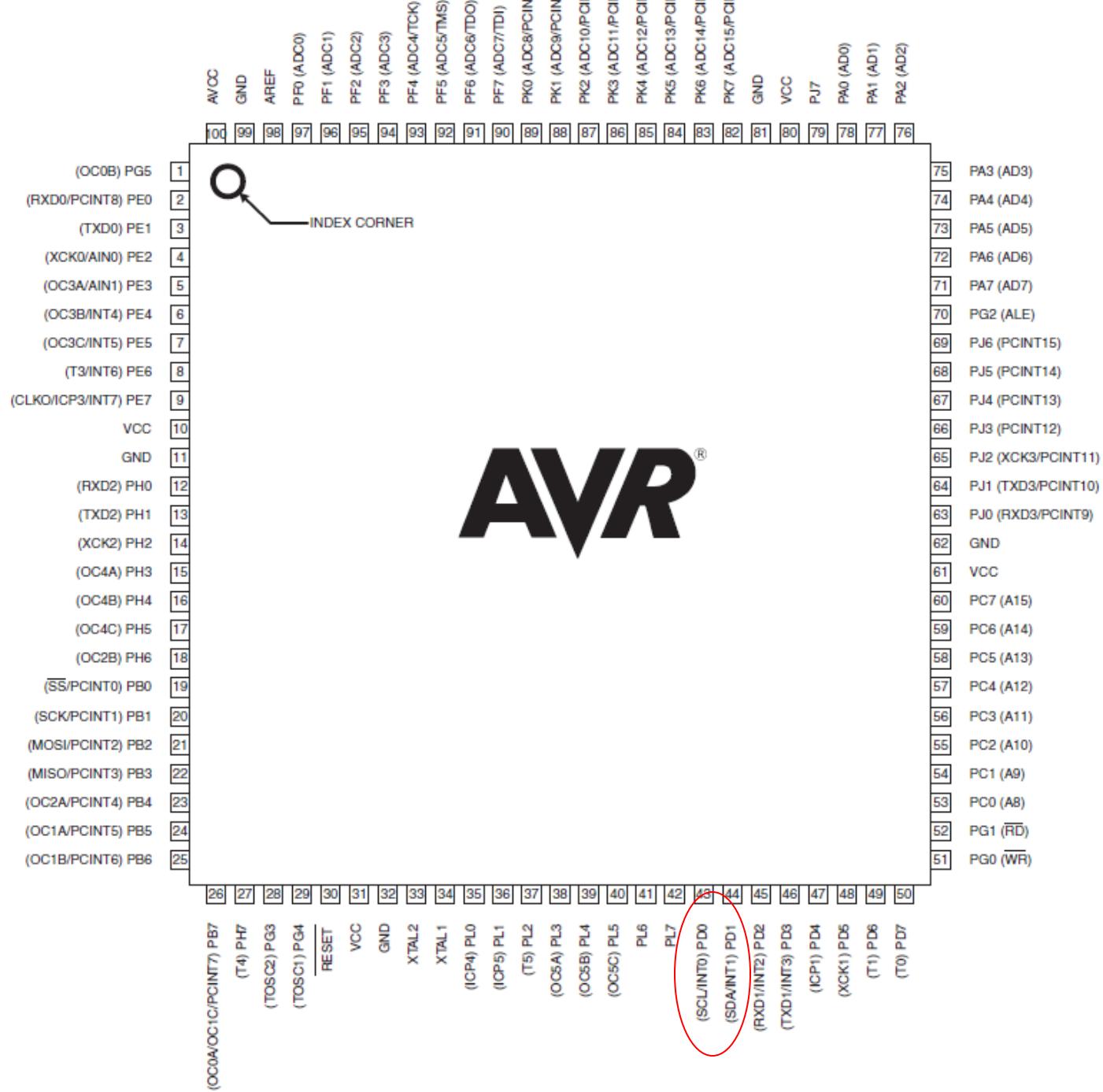
```
main:  
    clr count  
    clr temp  
  
loop:  
    inc temp          ; a dummy task in main  
  
    cpi temp, 0x1F    ; the following section in red  
    breq reset_temp  ; shows the need to save SREG  
    rjmp loop        ; in the interrupt service routine  
  
reset_temp:  
    clr temp  
    rjmp loop
```

# Example 2

- Based on Example 1, implement a software interrupt
  - When the counter that counts LED toggles reaches the maximum value 0xFF, all LEDs are turned on.

## Example 2 (solution)

- Use another external interrupt as software interrupt
  - INT1 is used (Port D bit 1)
  - Software generates the external interrupt request
- In the main program, test if counter=0xFF
  - If yes, write a value (based on the interrupt type chosen) to the pin to invoke the interrupt.



# Code for Example 2

```
.include "m2560def.inc"
.include "my_macros.inc" ; macros for oneSecondDelay

.def    temp =r16
.def    output = r17
.def    count = r18 ; counter the number of interrupts
.equ   PATTERN = 0b01010101
.equ   MAX = 0b11111111

; set up interrupt vectors
rjmp RESET ; vector for the RESET interrupt
.org   INT0addr
rjmp EXT_INT0 ; vector for INT0
.org   INT1addr
jmp EXT_INT1 ; vector for INT1

RESET: ; continued
```

# Code for Example 2 (cont.)

; continued

```
ser temp                                ; set Port C as output
out DDRC, temp
ldi output, PATTERN
out PORTC, temp
ldi temp, 0b00000010
out DDRD, temp                            ; set Port D bit 1 as output
out PORTD, temp

ldi temp, (2 << ISC00) | (2 << ISC10)    ; set INT0 and INT1 as
sts EICRA, temp                           ; falling edge sensed interrupts

in temp, EIMSK                            ; enable INT0 and INT1
ori temp, (1<<INT0) | (1<<INT1)
out EIMSK, temp

sei                                     ; enable Global interrupt
jmp main                                 ; continued
```

# Code for Example 2 (cont.)

```
; continued
EXT_INT0:
    push temp          ; save register
    in temp, SREG      ; save SREG
    push temp

    com output         ; flip the pattern
    out PORTC, output
    inc count

    pop temp           ; restore SREG
    out SREG, temp
    pop temp           ; restore register
    reti

; continued
```

# Code for Example 2 (cont.)

```
; continued
EXT_INT1:
    push temp
    in temp, SREG
    push temp

    ldi output, MAX
    out PORTC, output
    oneSecondDelay ; macro for one second delay
                    ; stored in "my_macros.inc"

    ldi output, PATTERN
    sbi PORTD, 1
    pop temp
    out SREG, temp
    pop temp
    reti

; set pattern for normal LED display
; set bit for INT1
; continued
```

# Code for Example 2 (cont.)

```
; continued

                                ; main - does nothing but increment a counter
main:
    clr count
    clr temp
loop:
    inc temp
    cpi count, 0xFF
    breq max_value      ; if reaches max value
    rjmp loop
max_value:
    cbi PORTD, 1        ; generate an INT1 request
    clr count           ; prepare for the next sw interrupt
    rjmp loop
```

# Reading Material

- Chapter 10: Interrupts and Real-Time Events.  
Microcontrollers and Microcomputers by  
Fredrick M. Cady.
- Mega2560 Data Sheet.
  - System Control and Reset.
  - Interrupts.

# Homework

1. Based your code for Task 3 in Lab 2, modify it by using an external interrupt to start and stop the LEDs' display.

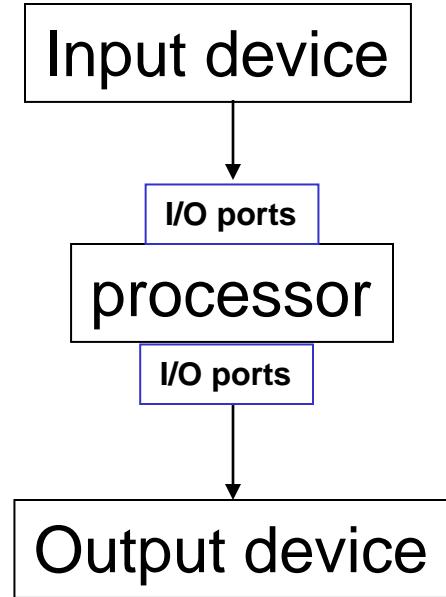
# **Microprocessors & Interfacing**

*Input/Output Devices (II)*

Lecturer : Annie Guo

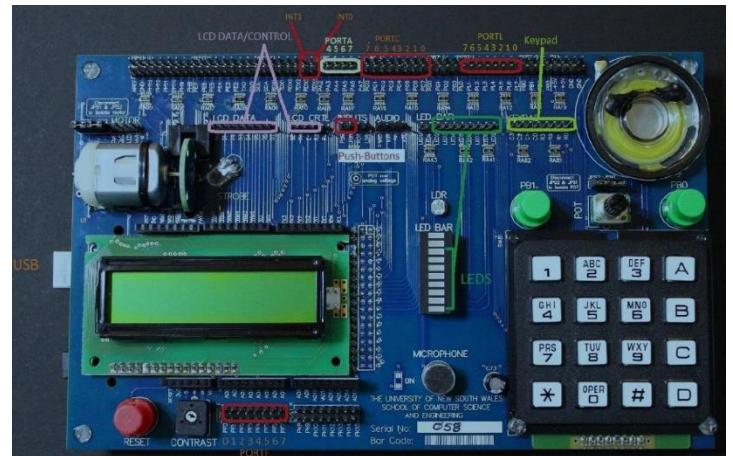
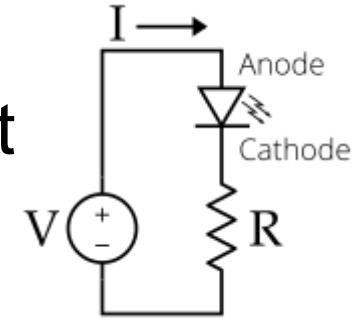
# Lecture Overview

- Output devices
  - LED
  - LCD



# LED

- Light-Emitting Diode
- Emit light when current flows through it
  - Its brightness increases with the current value
    - Within a limited range
- Can be used to indicate
  - a 1-bit digital output
    - LED on,  $V=1$
    - LED off,  $V=0$
  - an analog output value
    - To be covered later



# LCD

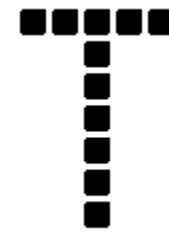
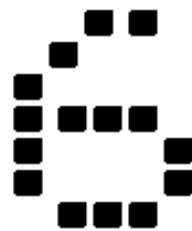
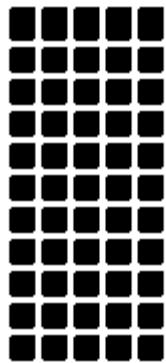
- Liquid Crystal Display
- Programmable output device



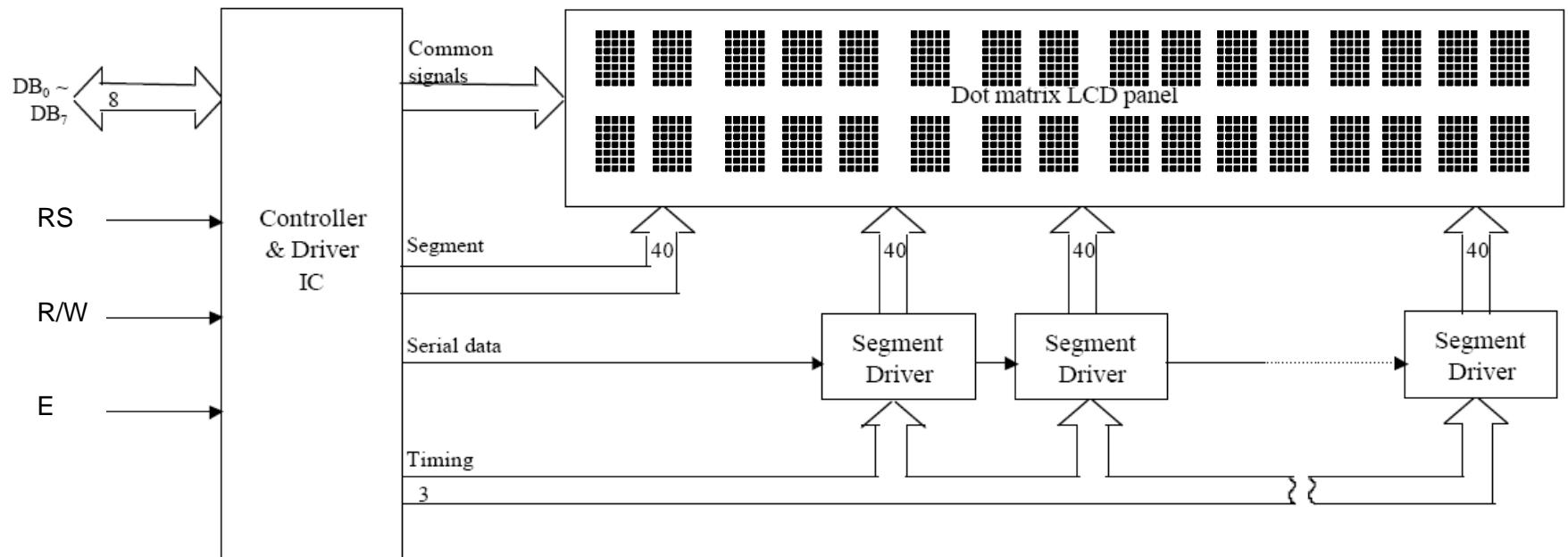
[How it works](#)

# Dot Matrix LCD

- Characters are displayed using a dot matrix.
  - 5x7, 5x8, and 5x11
- A controller is used for communication between the LCD and other components, e.g. microprocessor unit (MPU)
- The controller has an internal character generator ROM. All display functions are controllable by instructions.



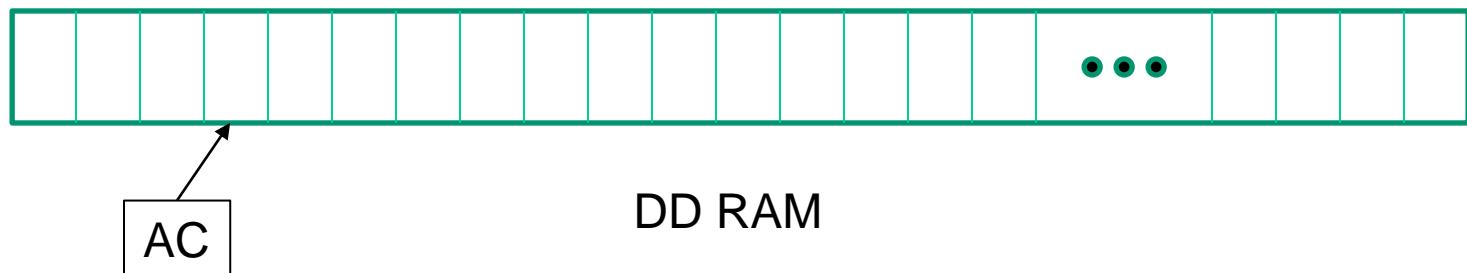
# Dot Matrix LCD Diagram\*



Note: The diagram and tables are extracted from the LCD Manual available on the course website

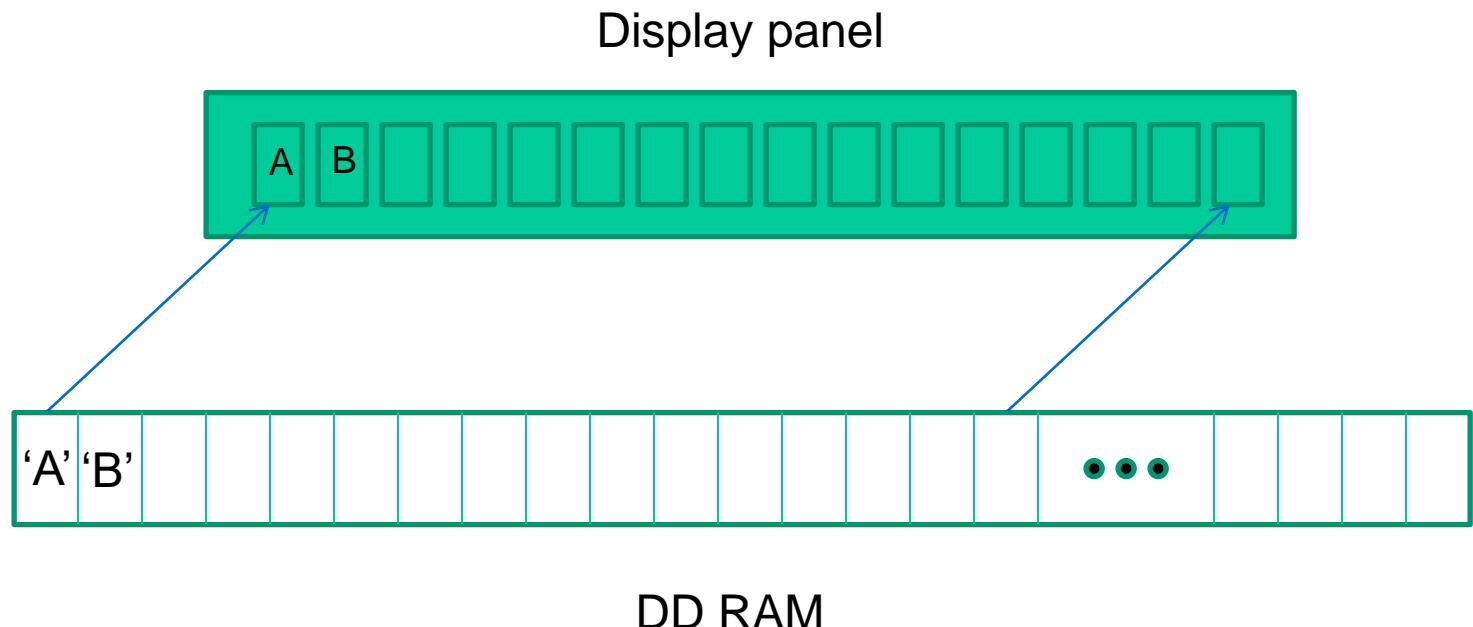
# General Overview

- The LCD has
  - Instruction register, IR
  - Data register, DR
  - Display Data RAM, DD RAM
    - To store the characters to be displayed
    - The address counter (AC) is used to control the location of the data to be read or written
      - AC can be automatically increased/decreased



# General Overview (cont.)

- A line on the LCD display panel can display a window of the characters in DD RAM



# Pin Assignments

Pin Number	Symbol
1	$V_{ss}$
2	$V_{cc}$
3	$V_{ee}$
4	RS
5	R/W
6	E
7	DB0
8	DB1
9	DB2
10	DB3
11	DB4
12	DB5
13	DB6
14	DB7

# Pin Description

Signal name	No. of Lines	Input/Output	Connected to	Function
DB4 ~ DB7	4	Input/Output	MPU	4 lines of high order data bus. Bi-directional transfer of data between MPU and module is done through these lines. Also DB <sub>7</sub> can be used as a busy flag. These lines are used as data in 4 bit operation.
DB0 ~ DB3	4	Input/Output	MPU	4 lines of low order data bus. Bi-directional transfer of data between MPU and module is done through these lines. In 4 bit operation, these are not used and should be grounded.
E	1	Input	MPU	Enable - Operation start signal for data read/write.
R/W	1	Input	MPU	Signal to select Read or Write “0”: Write “1”: Read
RS	1	Input	MPU	Register Select “0”: Instruction register (Write) : Busy flag; Address counter (Read) “1”: Data register (Write, Read)
Vee	1		Power Supply	Terminal for LCD drive power source.
Vcc	1		Power Supply	+5V
Vss	1		Power Supply	0V (GND)

# Operations

- MPU communicates with LCD through two registers
  - Instruction Register (IR)
    - To store
      - instruction code
        - » e.g Display Clear or Cursor Shift
      - address of DD RAM
      - etc.
    - Data Register (DR)
      - To store
        - data to be read/written to/from DD RAM in the display controller.

# Operations (cont.)

- The register select (RS) signal determines which of the two registers (IR and DR) is selected to use
- The table below shows the operations by the two control signals, RS and R/W

RS	R/W	Operation
0	0	IR write, internal operation (Display Clear etc.)
0	1	Busy flag (DB <sub>7</sub> ) and Address Counter (DB <sub>0</sub> ~ DB <sub>6</sub> ) read
1	0	DR Write, Internal Operation (DR ~ DD RAM or CG RAM)
1	1	DR Read, Internal Operation (DD RAM or CG RAM)

# Operations (cont.)

- When the busy flag is high or “1”, the LCD is busy with the internal operation.
- The next instruction **must not be** written/sent to LCD until the busy flag is low or “0”.
- For details, refer to the LCD USER’S MANUAL.

# LCD Instructions

- A list of binary instructions are available for LCD operations
- Some typical ones are explained in the next slides.
  - More information and examples can be found in the LCD user's manual

# Instructions

- Function Set

	RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
Code	0	0	0	0	1	DL	N	F	X	X

- Set the interface data length (DL), the number of lines (N), and character font size (F).
  - DL = 1, data sent to LCD in 8-bit (bus) size, otherwise 4-bit size
  - N = 1, 2-line display, otherwise 1-line display
  - F = 1, 5 x 10 dots, otherwise 5 x 7 dots
- e.g. 0b00111000
  - set LCD for 2-line 5x7 display and the 8-bit data bus is used



# Instructions

- Entry Mode Set

	RS	R/W	DB7	DB6	DB5	BD4	DB3	DB2	DB1	DB0
Code	0	0	0	0	0	0	0	1	I/D	S

- Set the Increment/Decrement and Shift modes
  - I/D = 1: increment the address counter by 1 for each DD RAM access (read or write); I/D = 0: decrement the address counter
  - S=0, no shift
  - S=1, shift the entire display
    - Shift to the left when I/D = 1
    - Shift to the right when I/D = 0
- e.g. 00000110
  - LCD\_display will go from left to right

# Instructions

- Write Data to DD RAM

	RS	R/W	DB7	DB6	DB5	BD4	DB3	DB2	DB1	DB0
Code	1	0	D	D	D	D	D	D	D	D

- Write binary 8-bit data DDDDDDDDD to DD RAM pointed by the address counter (AC)
  - e.g. data = 0100 0001 for 'A'
- After a write, AC will be automatically increased or decreased, based on the entry mode setting.
  - I/D = 1, AC is increased; I/D = 0, AC is decreased.

# Instructions

- Set DD RAM Address

	RS	R/W	DB7	DB6	DB5	BD4	DB3	DB2	DB1	DB0
Code	0	0	1	A	A	A	A	A	A	A

- Set the address counter (pointing to DD RAM).
- The address range:
  - For 1-line display, 0x00-0x4F
  - For 2-line display,
    - 0x00-0x27 for the first line
    - 0x40-0x67 for the second line

# Instructions

- Display ON/OFF Control

	RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
Code	0	0	0	0	0	0	1	D	C	B

- Control to turn on/off the display, cursor and cursor blink functions
  - D: The display is on when D = 1, off when D = 0.
  - C: The cursor is on when C = 1, off when C = 0.
  - B: The character indicated by the cursor blinks when B = 1.

# Instructions

- Clear Display

	RS	R/W	DB7	DB6	DB5	BD4	DB3	DB2	DB1	DB0
Code	0	0	0	0	0	0	0	0	0	1

- Clear the display and the cursor moves to the upper left corner of the display.

# Instructions

- Return Home

RS	R/W	DB7	DB6	DB5	BD4	DB3	DB2	DB1	DB0
Code	0	0	0	0	0	0	0	1	x

- The cursor moves to the upper left corner of the display. Text on the display remains unchanged.

# Instructions

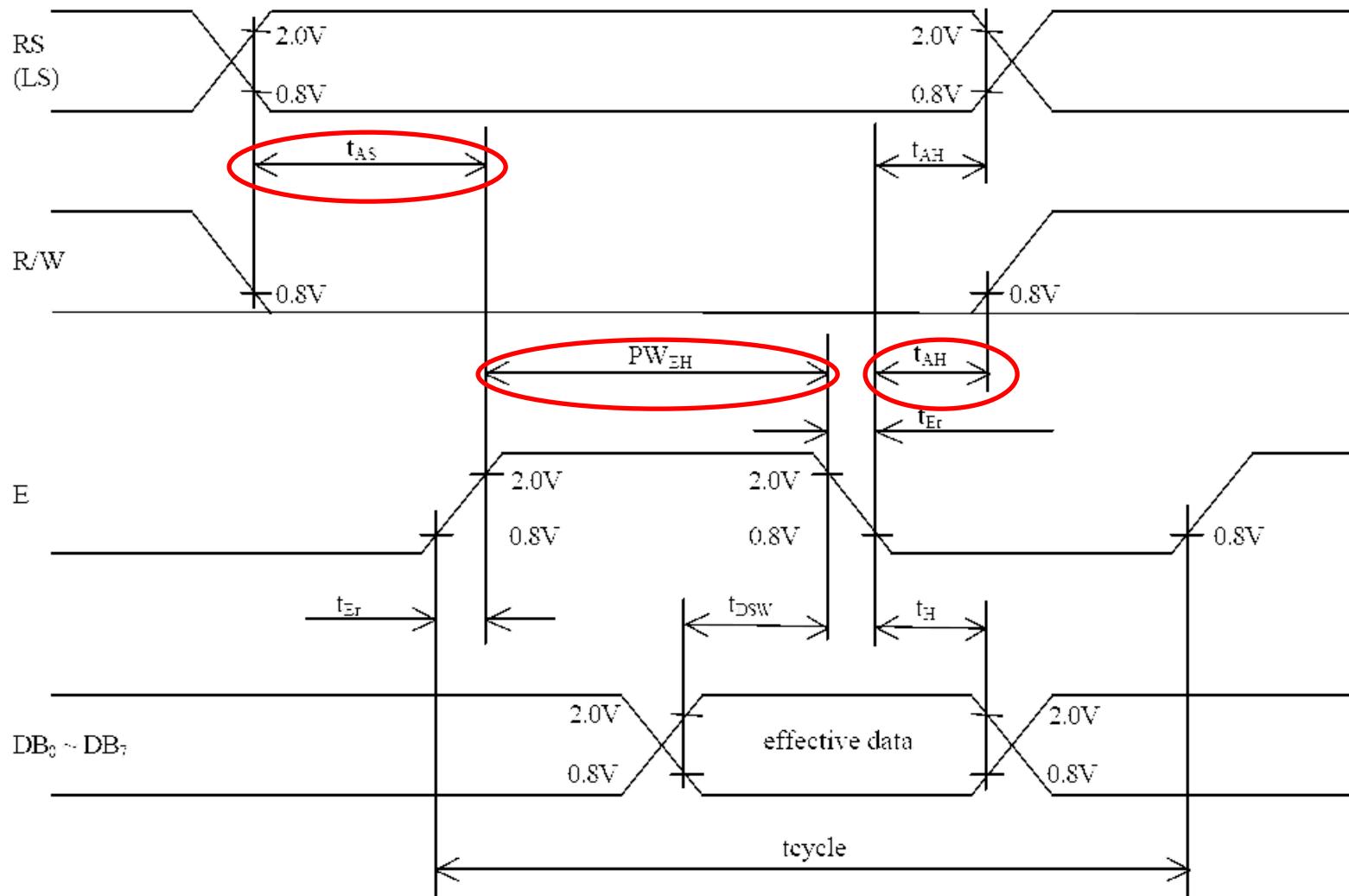
- Read Busy Flag and Address

	RS	R/W	DB7	DB6	DB5	BD4	DB3	DB2	DB1	DB0
Code	0	1	BF	A	A	A	A	A	A	A

- Read the busy flag (BF) and value of the address counter (AC). BF = 1 indicates that an internal operation is in progress and the next instruction will not be accepted until BF is set to “0”.
  - If the display is written (a write operation is performed) while BF = 1, abnormal operation will occur.

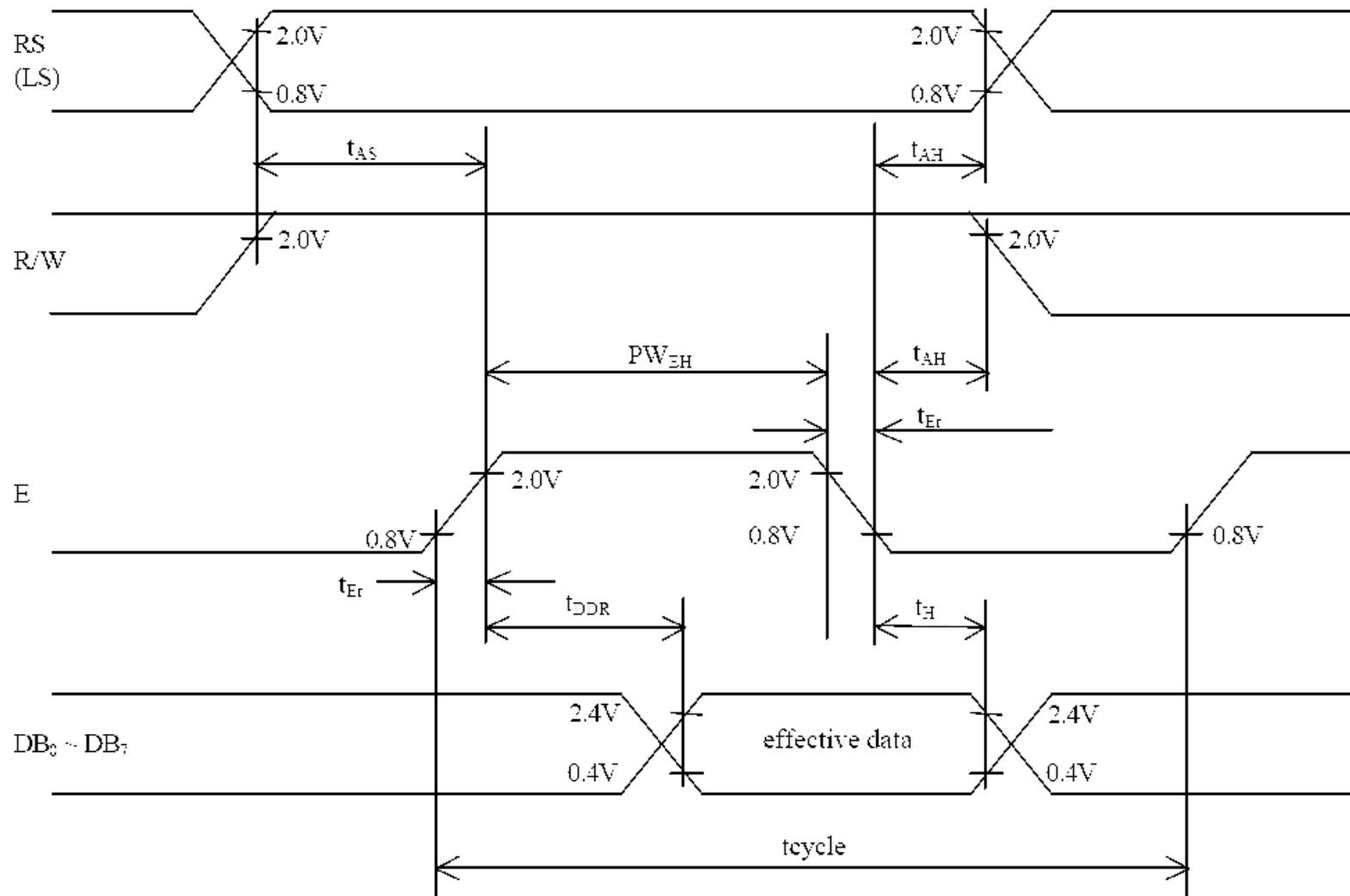
# Timing Requirement

- For write operation



# Timing Requirement

- For read operation



# Examples

- Send a command to LCD



- LCD\_RS: pin position for setting RS
- LCD\_E: pin position for setting E
- LCD\_RW: pin position for setting RW

; General purpose register **data** stores value to be written to the LCD  
; Port F is **output port** and connects to LCD data port; Port A controls the LCD (Bit LCD\_RS for RS and bit LCD\_RW for RW, LCD\_E for E). The character to be displayed is stored in register **data**  
; Assume all labels are pre-defined.

```
.macro lcd_write_com
    out PORTF, data          ; set the LCD data port's value up
    ldi temp, (0<<LCD_RS)|(0<<LCD_RW)
    out PORTA, temp          ; RS = 0, RW = 0 for a command write
    nop                      ; delay to meet timing (Set up time)
    sbi PORTA, LCD_E         ; turn on the enable pin
    nop                      ; delay to meet timing (Enable pulse width)
    nop
    nop
    cbi PORTA, LCD_E         ; turn off the enable pin
    nop                      ; delay to meet timing (Enable cycle time)
    nop
    nop
    nop
.endmacro
```

# Examples

- Send data to display

; comments are same as in the previous slide.

```
.macro lcd_write_data
    out PORTF, data          ; set the data port's value up
    ldi temp, (1 << LCD_RS)|(0<<LCD_RW)
    out PORTA, temp          ; RS = 1, RW = 0 for a data write
    nop                      ; delay to meet timing (Set up time)
    sbi PORTA, LCD_E         ; turn on the enable pin
    nop                      ; delay to meet timing (Enable pulse width)
    nop
    nop
    cbi PORTA, LCD_E         ; turn off the enable pin
    nop                      ; delay to meet timing (Enable cycle time)
    nop
    nop
    nop
.endmacro
```

# Examples

- Check LCD and wait until LCD is not busy

```
; comments are same as in the previous slide
.macro lcd_wait_busy
    clr temp
    out DDRF, temp
    ldi temp, 1 << LCD_RW
    out PORTA, temp
    busy_loop:
        nop
        sbi PORTA, LCD_E
        nop
        nop
        in temp, PINF
        cbi PORTA, LCD_E
        sbrc temp, LCD_BF
        rjmp busy_loop

        clr temp
        out PORTA, temp
        ser temp
        out DDRF, temp
.endmacro
```

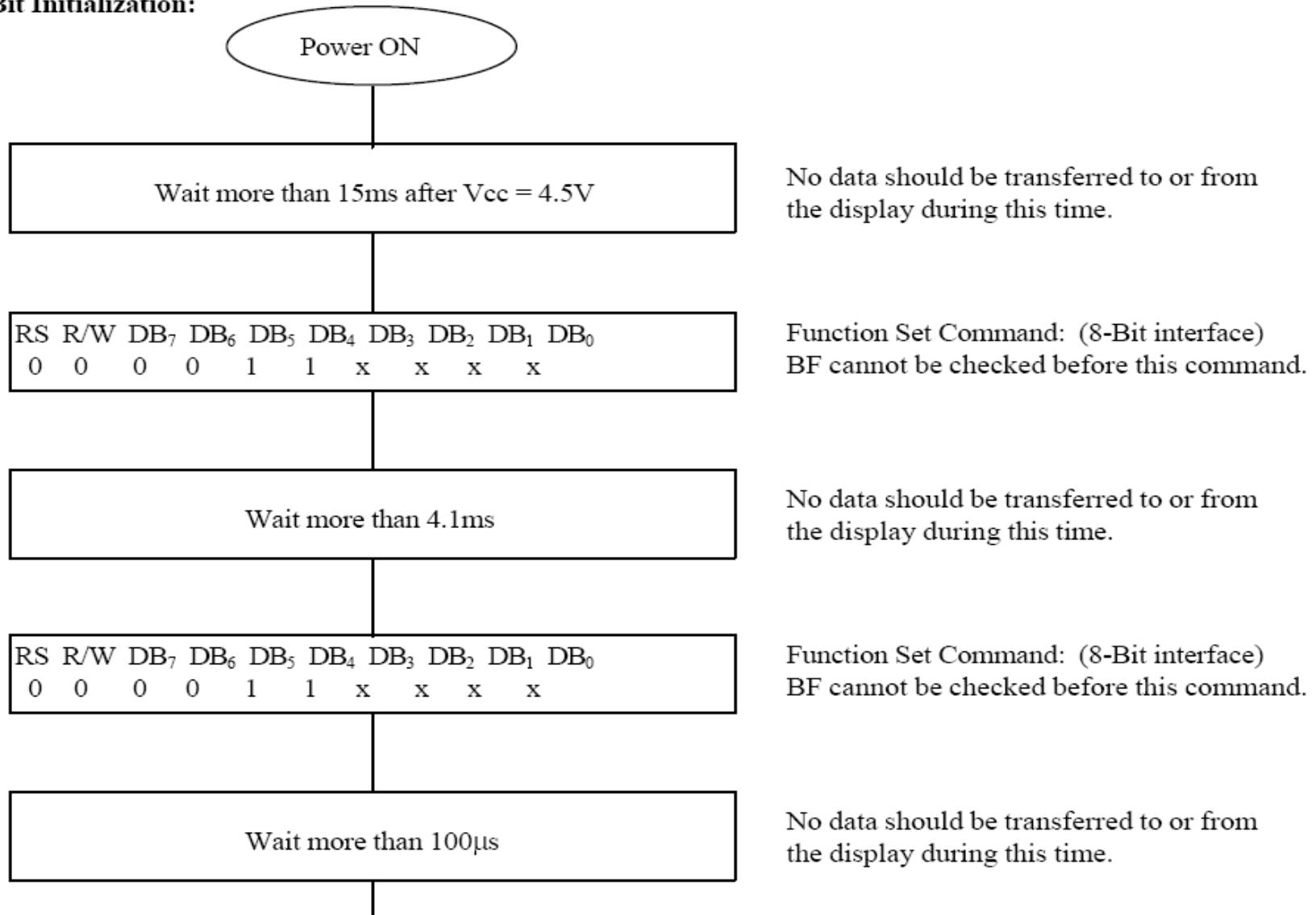
; Make port F as an input port for now  
; RS = 0, RW = 1 for a command port read  
; delay to meet set-up time  
; turn on the enable pin  
; delay to meet timing (Data delay time)  
; read value from LCD  
; turn off the enable pin  
; if the busy flag is set  
; repeat command read  
; else  
; turn off read mode,  
;  
; make port F an output port again

# LCD Initialization

- LCD should be initialized before use
- Internal Reset circuit can be used, but it is related to power supply load, may not work properly.
- Therefore, software initialization is recommended.

# Software Initialization

## 8 - Bit Initialization:



# Software Initialization

Wait more than 100µs

No data should be transferred to or from the display during this time.

RS	R/W	DB <sub>7</sub>	DB <sub>6</sub>	DB <sub>5</sub>	DB <sub>4</sub>	DB <sub>3</sub>	DB <sub>2</sub>	DB <sub>1</sub>	DB <sub>0</sub>
0	0	0	0	1	1	x	x	x	x

Function Set Command: (8-Bit interface)  
After this command is written, BF can be checked.

RS	R/W	DB <sub>7</sub>	DB <sub>6</sub>	DB <sub>5</sub>	DB <sub>4</sub>	DB <sub>3</sub>	DB <sub>2</sub>	DB <sub>1</sub>	DB <sub>0</sub>
0	0	0	0	1	1	N	F	x	x

Function Set      (Interface = 8 bits, Set No. of lines and display font)

0	0	0	0	0	0	1	0	0	0
---	---	---	---	---	---	---	---	---	---

Display OFF

0	0	0	0	0	0	0	0	0	1
---	---	---	---	---	---	---	---	---	---

Clear Display

0	0	0	0	0	0	0	1	I/D	S
---	---	---	---	---	---	---	---	-----	---

Entry Mode Set:

0	0	0	0	0	0	1	1	C	B
---	---	---	---	---	---	---	---	---	---

Display ON      (Set C and B for cursor/Blink options.)

Initialization Complete,  
Display Ready.

Note:      BF should be checked before each of the instructions starting with Display OFF.

# Example of Initialization Code

```
.include "m2560def.inc"

; The del_hi:del_lo register pair store the loop count
; each iteration of loop1 generates 1 us delay
; if the clock frequency is 16 MHz
.macro delay
loop1:
    ldi r16, 0x3
loop2: dec r16
        nop
        brne loop2

        subi del_lo, 1
        sbci del_hi, 0
        brne loop1      ; taken branch takes two cycles.

.endmacro
; continued
```

# Example of Initialization Code

```
ldi del_lo, low(15000) ;delay (15ms)
ldi del_hi, high(15000)
delay

; Function set command with N = 1 and F = 0
; for 2 line display and 5*7 font. The 1st command
ldi data, LCD_FUNC_SET | (1 << LCD_N)
lcd_write_com

ldi del_lo, low(4100) ; delay (4.1 ms)
ldi del_hi, high(4100)
delay

lcd_write_com ; 2nd Function set command
; continued
```

# Example of Initialization Code

```
ldi del_lo, low(100) ; delay (100 ns)
ldi del_hi, high(100)
delay

lcd_write_com ; 3rd Function set command
lcd_write_com ; Final Function set command

lcd_wait_busy ; Wait until the LCD is ready
ldi data, LCD_DISP_OFF
lcd_write_com ; Turn Display off

lcd_wait_busy ; Wait until the LCD is ready
ldi data, LCD_DISP_CLR
lcd_write_com ; Clear Display

; continued
```

# Example of Initialization Code

```
lcd_wait_busy ; Wait until the LCD is ready
; Entry set command with I/D = 1 and S = 0
; Set Entry mode: Increment = yes and Shift = no
ldi data, LCD_ENTRY_SET | (1 << LCD_ID)
lcd_write_com

lcd_wait_busy ; Wait until the LCD is ready
; Display On command with C = 1 and B = 0
ldi data, LCD_DISP_ON | (1 << LCD_C)
lcd_write_com

; ...
```

A working sample code is available on the course website

# Reading Material

- DOT Matrix LCD User's Manual
  - Available on the course website
    - On the Resources page
  - The useful examples of instructions can be found on pages 41-46.

# Homework

Write an assembly program for LCD to

1. display “COMP9032” from left to right on the first line
2. display “ComArch” from right to left on the second line

# **Microprocessors & Interfacing**

*Input/Output Devices (I)*

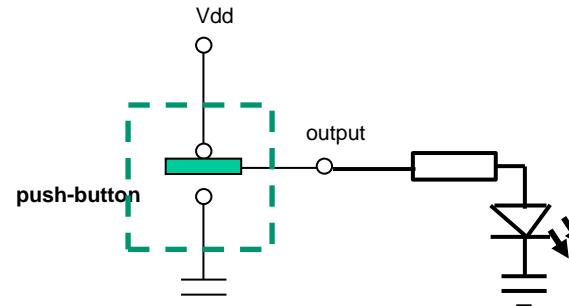
Lecturer : Annie Guo

# Lecture Overview

- Input devices
  - Push button
  - Input switch
  - Keypad

# Push Button

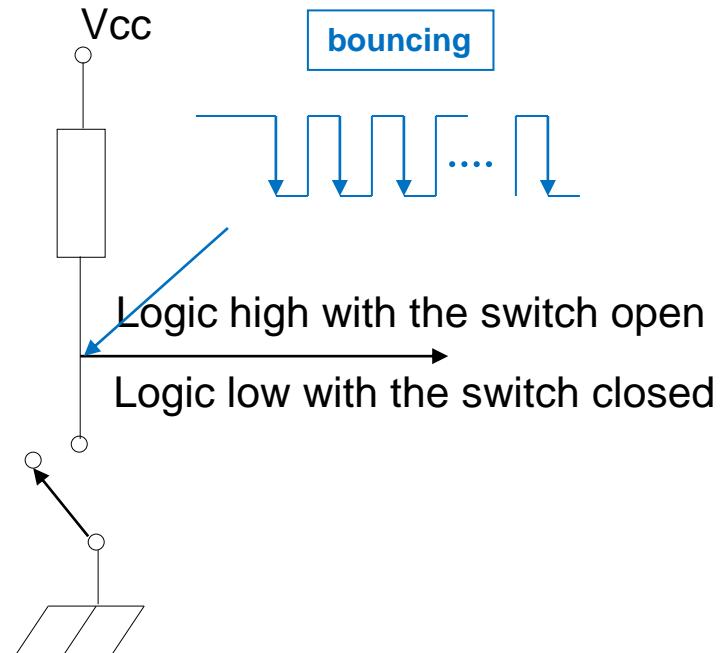
- A small mechanical device that can control the connection of two electric nodes (wires).
  - When it is pushed, the small metal inside the button connects two wires.
- Can be used as a 1-bit input device, as used in our lab board
  - Not pushed: 1
  - Pushed down: 0



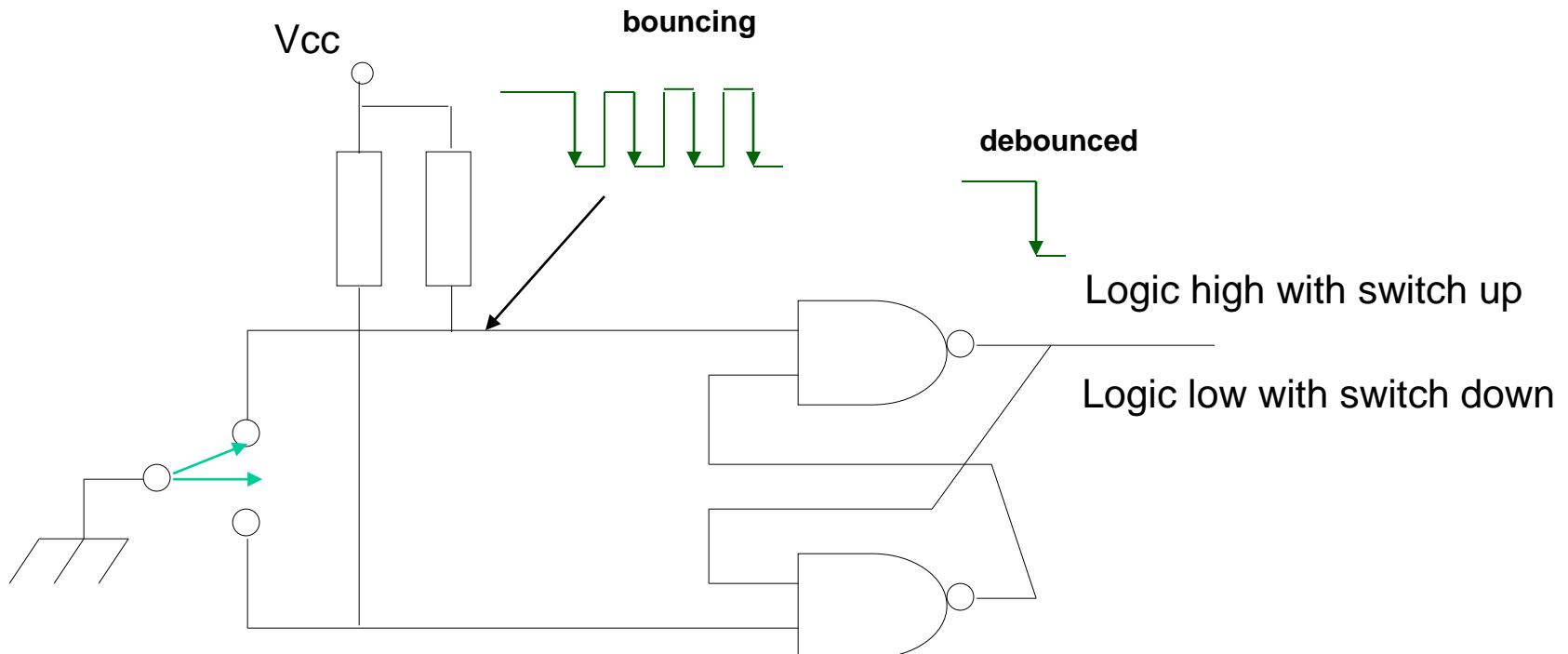
# Input Switch

- Like the push button, a switch provides two different values, depending on the switch position.
- Pull-up resistor/circuit may be needed for the switch to provide a high logic level when the switch is open.
- Problem with switch (also push button):
  - **Switch bouncing**
    - When a switch makes contact, its mechanical springiness will cause the contact to bounce, namely contact and break, for a few milliseconds (typically 5 to 10 ms).

# Switch Bouncing Example



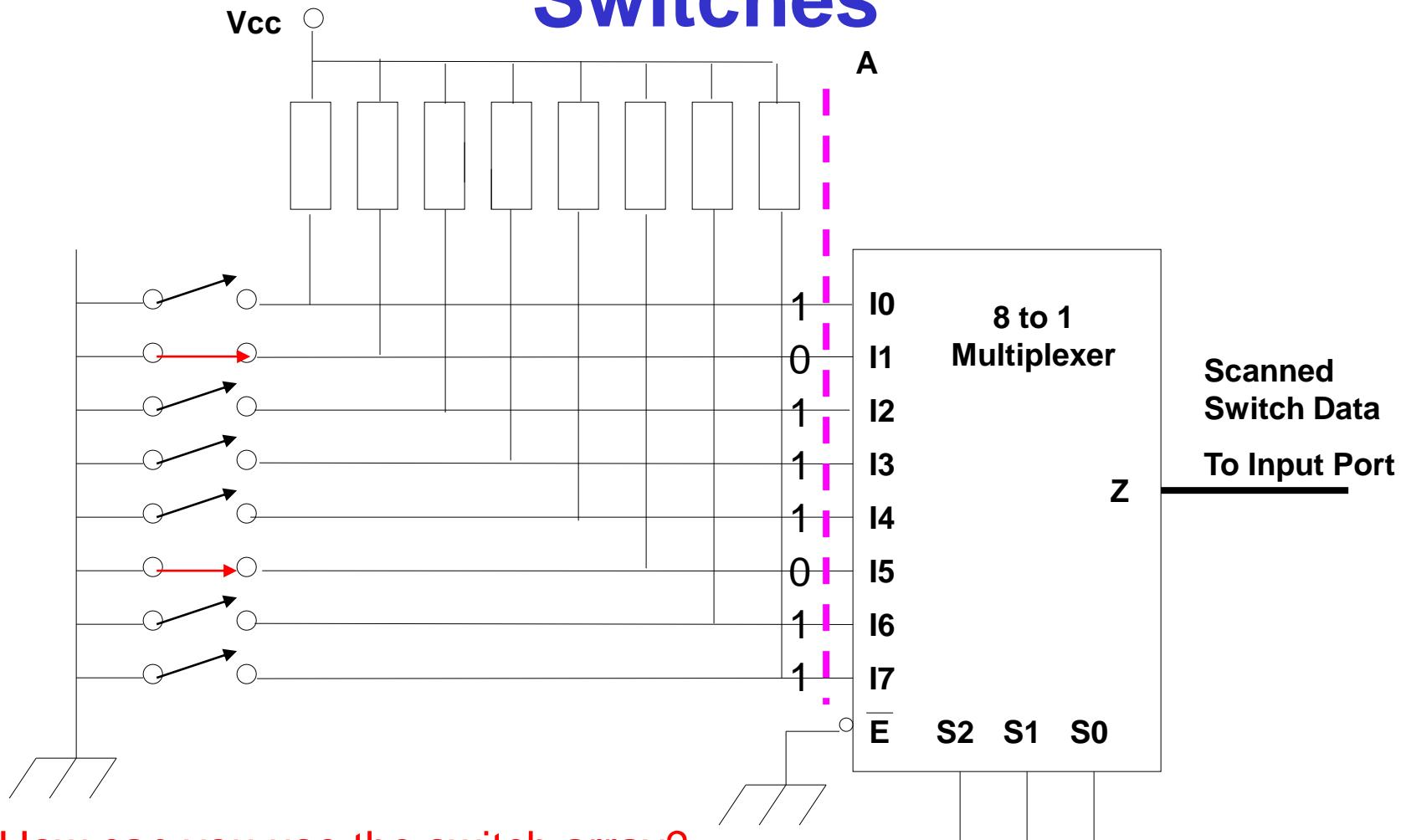
# NAND Latch Debouncer\*



# Software Debouncing

- Basic idea: wait until the switch is stable
- For example:
  - Wait and see:
    - If the software detects a low logic level, indicating that switch has closed, it simply waits for some time, say 20 to 100ms, and then tests if the switch is still low.
  - Counter-based approach:
    - Initialize a counter to 10.
    - Poll the switch every millisecond until the counter is either 0 or 20.
      - If the switch output is low, decrease the counter; otherwise, increment the counter.
    - If the counter is 0, we know that switch output has been low (closed) for at least 10 ms. If, on the other hand, the counter reaches 20, we know that the switch output has been high for at least 10 ms.

# One-Dimensional Array of Switches



How can you use the switch array?

- get all bits from each bit line
- scan each bit in a sequence

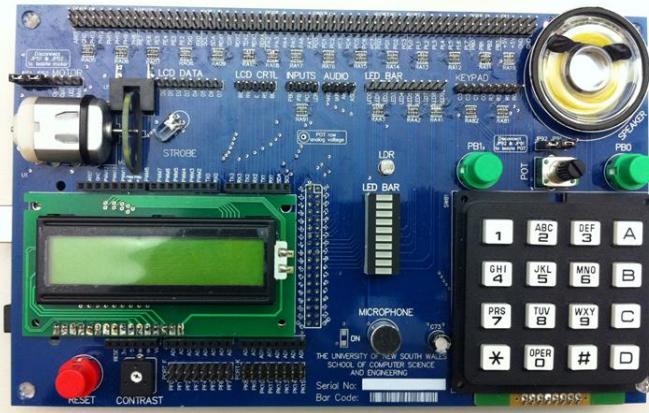
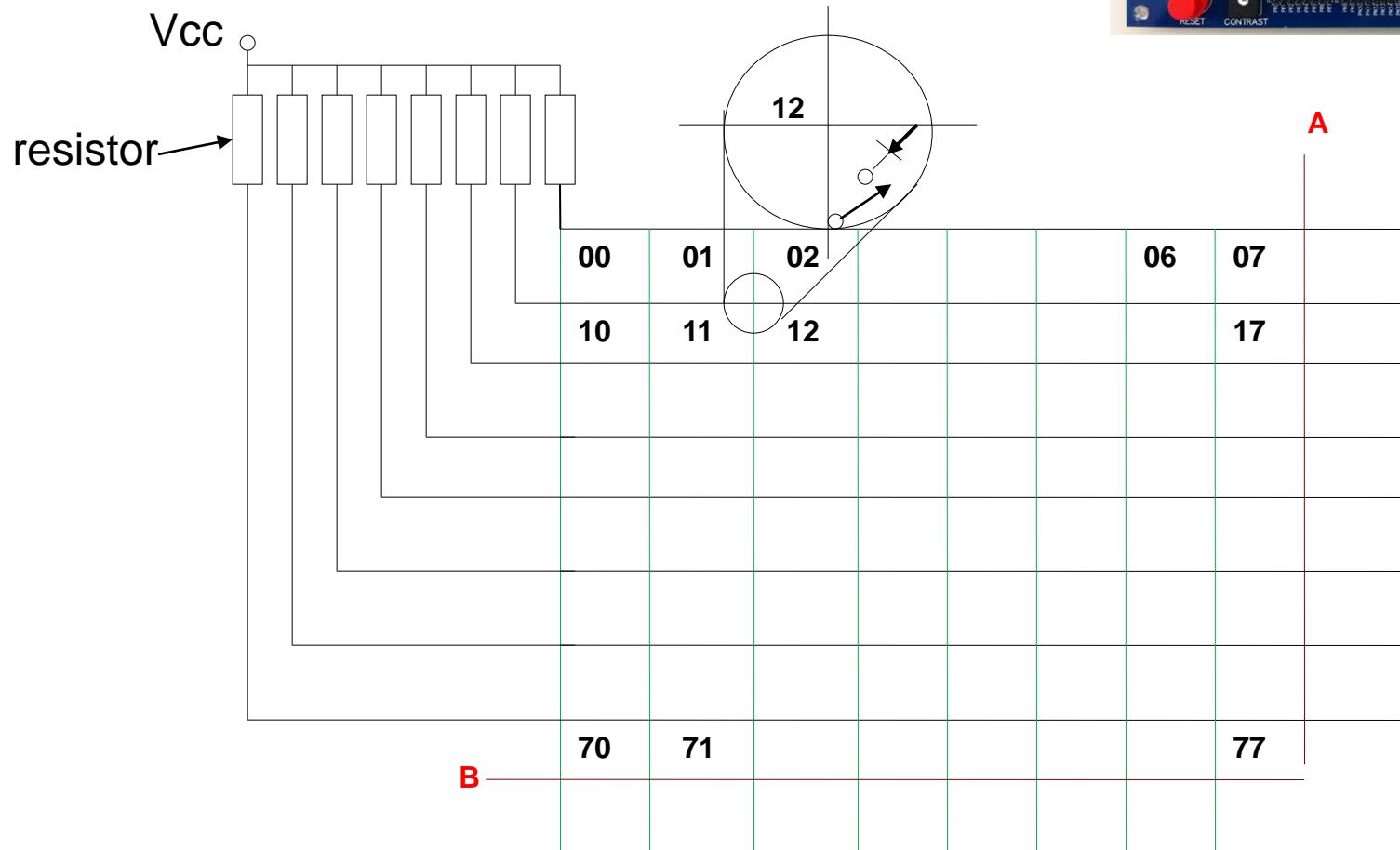
Select-Input From Output Port

# One-Dimensional Array of Switches (cont.)

- Switch bouncing problem must be solved
  - Either using software or hardware
- The output of switch array can be interfaced directly to an eight-bit port at point A.
- The array of switches can also be scanned by the software to find out which switches are closed or open.
  - The software outputs a 3-bit sequence from 000 to 111 and the multiplexer selects each of the switch inputs.

# Keypad

- Internal circuit diagram



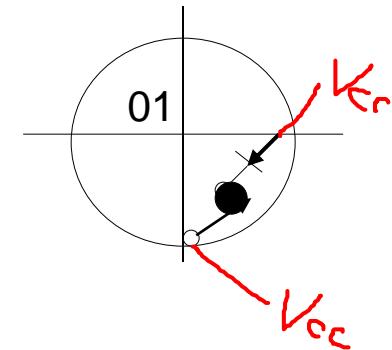
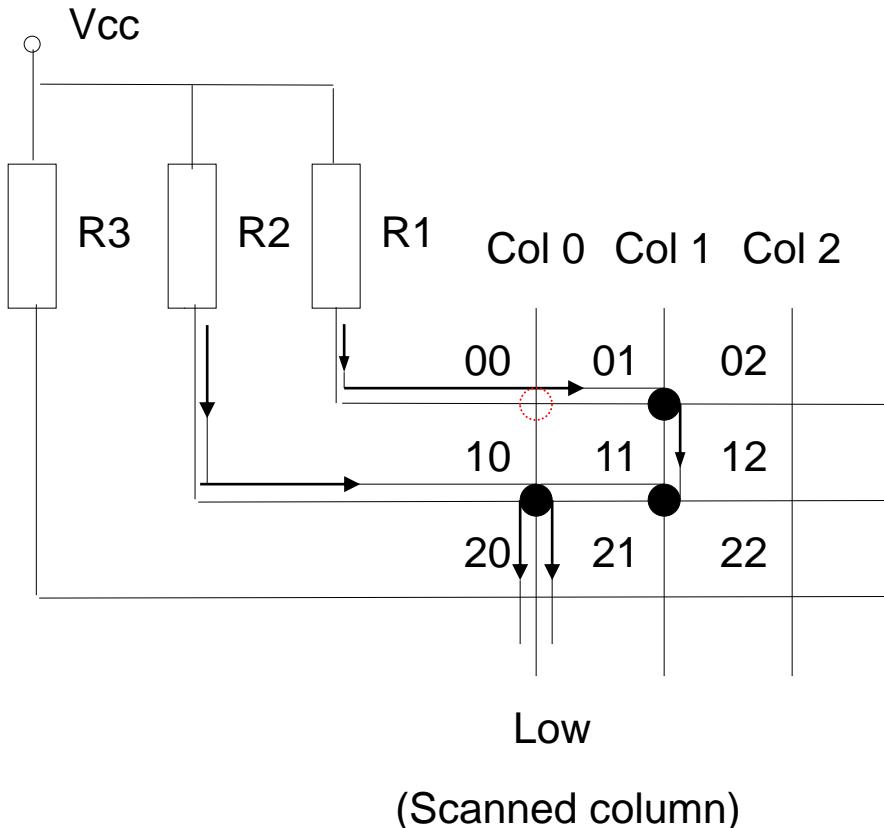
# Keypad (cont.)

- A keypad is a set of switches arranged in a two-dimensional matrix, consisting of two layers
  - A layer of the horizontal lines
    - connected to the power supply via resistors
  - A layer of the vertical lines
    - normally disconnected to the horizontal layer
- Each intersection of the vertical and horizontal lines forms a switch
  - The switch can be operated by a key button
  - When the key is pressed, the switch connects both horizontal and vertical lines.

# Keypad (cont.)

- The 8\*8 keypad can be interfaced directly to 8-bit output and input ports
  - at point *A* (*to input port*) and point *B* (*to output port*)
- The output from each horizontal line
  - normally is a logic high (1)
  - becomes logic low (0) when a key is pressed, **and** the related vertical line is set/connected to logic low (0)
- The diode prevents a problem called **ghosting**.

# Ghosting\*



Row 0 (Pulled low, error)

Row 1 (Pulled low, OK)

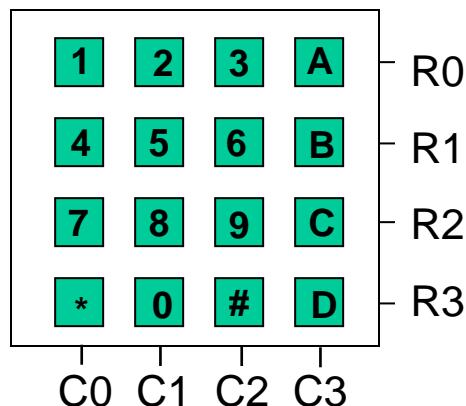
Row 2 (High, OK)

# Ghosting (cont.)\*

- Ghosting occurs when several keys are pushed at once.
- Consider the case shown in the figure in the previous slide, where three switches 01, 10 and 11 are all closed. Column 0 is selected with a logic low and assume that the circuit does not contain the diodes.  
As the rows are scanned, a low is sensed on Row 1, which is true because switch 10 is closed. But a low is also seen on Row 0, indicating switch 00 is closed, which is NOT true.
- The diodes in the switches eliminate this problem by preventing current flow from resistor R1 through switches 01 and 11. Thus Row 0 will not be low when it is scanned.

# Example

- Get an input from 4x4 keypad used in our lab board.



# Example (solution)

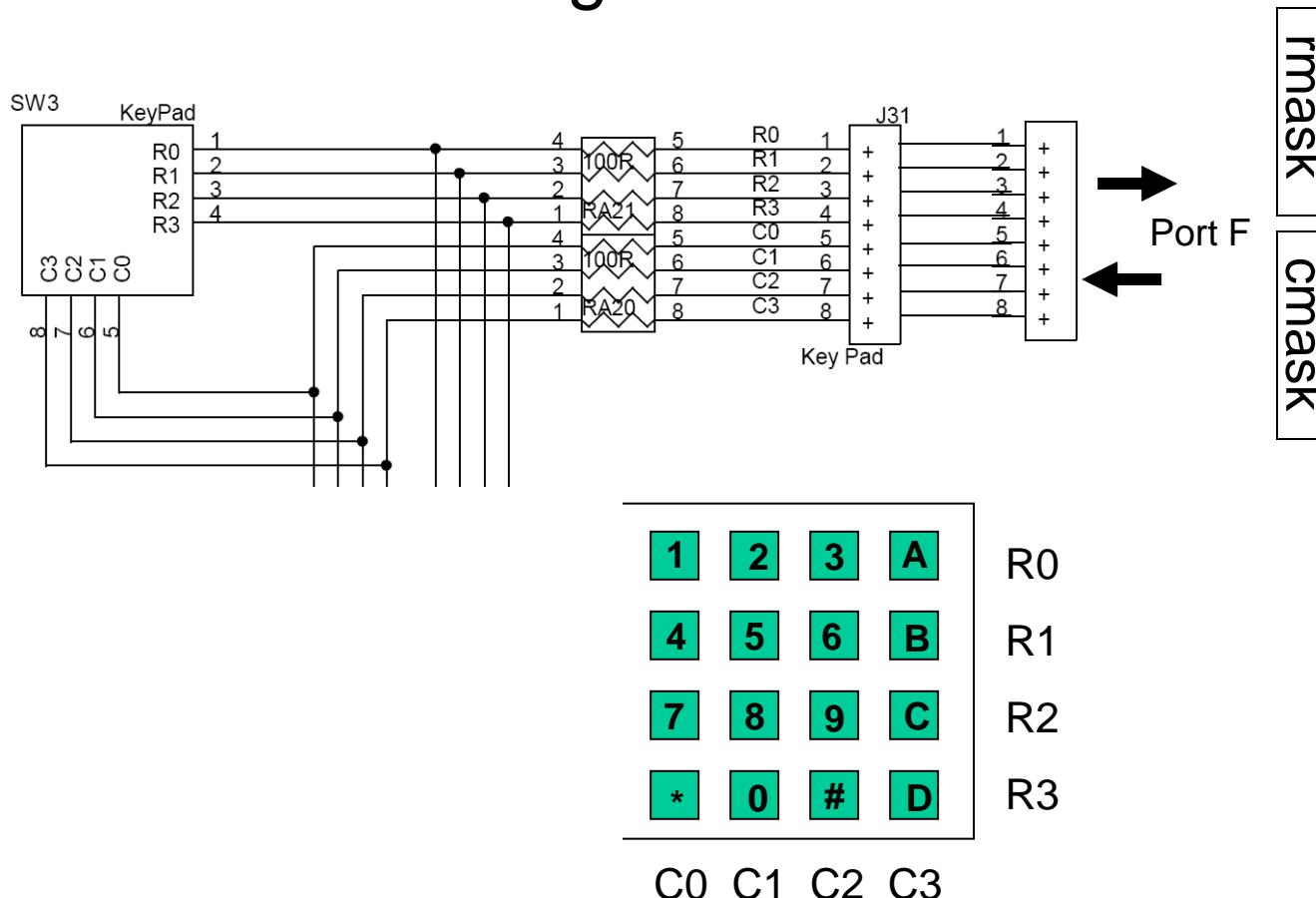
- Algorithm

```
scan columns from left to right
    for each column, scan rows from top to bottom
        for each key being scanned
            if it is pressed
                display
                wait
            endif
        endfor
    endfor
repeat the scan process
```

- To select a column, set the related C<sub>x</sub> value to 0
- A mask is used to read one row at a time.

# Example (solution)

- Hardware Interfacing



# Code Implementation

```
; The program gets input from keypad and displays its ascii value on the
; LED bar

.include "m2560def.inc"

.def row = r16           ; current row number
.def col = r17           ; current column number
.def rmask = r18          ; mask for current row during scan
.def cmask = r19          ; mask for current column during scan
.def temp1 = r20
.def temp2 = r21

.equ PORTFDIR = 0xF0      ; PF7-4: output, PF3-0, input
.equ ROWMASK = 0x0F        ; for obtaining input from Port F
.equ INITCOLMASK = 0xEF    ; scan from the leftmost column,
.equ INITROWMASK = 0x01    ; scan from the top row
```

# Code Implementation

```
; The program gets input from keypad and displays its ascii value on the
; LED bar

.include "m2560def.inc"

.def row = r16           ; current row number
.def col = r17           ; current column number
.def rmask = r18          ; mask for current row during scan
.def cmask = r19          ; mask for current column during scan
.def temp1 = r20
.def temp2 = r21

.equ PORTFDIR = 0xF0      ; PF7-4: output, PF3-0: input
.equ ROWMASK = 0x0F        ; for obtaining input from Port F
.equ INITCOLMASK = 0xEF    ; scan from Column 0 (C0)
.equ INITROWMASK = 0x01    ; scan from Row 0 (R0)
```

# Code Implementation

**RESET:**

ldi	temp1, PORTFDIR	; set Port F, PF7:4/PF3:0, out/in
out	DDRF, temp1	
ser	temp1	; PORTC is set for output
out	DDRC, temp1	; to display ASCII of pressed key.
out	PORTC, temp1	; Initially LEDs are turned on

**main:**

ldi	cmask, INITCOLMASK	; initial column mask
clr	col	; initial column

# Code Implementation

```
colloop:  
    cpi      col, 4  
    breq    main          ; if all keys are scanned, repeat.  
    out     PORTF, cmask  ; otherwise, scan the column  
  
delay:  ldi      temp1, 0xFF        ; slow down the scan operation.  
        dec      temp1  
        brne   delay  
  
        in       temp1, PINF        ; read PORTF  
        andi   temp1, ROWMASK     ; get the keypad output value  
        cpi    temp1, 0xF          ; check if any row is low  
        breq   nextcol  
  
        ldi      rmask, INITROWMASK ; if yes, find which row is low  
        clr      row              ; initialize for row check  
        ;
```

# Code Implementation

**rowloop:**

```
;cpi      row, 4
;breq    nextcol           ; the row scan is over.
mov      temp2, temp1
and      temp2, rmask      ; check un-masked bit
breq    convert            ; if bit is clear, the key is pressed
inc      row                ; else move to the next row
lsl      rmask
rjmp   rowloop
```

**nextcol:**

```
lsl cmask
inc col               ; increase column value
rjmp colloop          ; go to the next column
```

# Code Implementation

convert:

cpi	col, 3	; If the pressed key is in col. 3
breq	letters	; we have a letter
		; If the key is not in col. 3 and
cpi	row, 3	; if the key is in row3,
breq	symbols	; we have a symbol or 0
mov	temp1, row	; Otherwise we have a number in 1-9
lsl	temp1	
add	temp1, row	;
add	temp1, col	; temp1 = row*3 + col
subi	temp1, -'1'	; Add the value of character '1'
rjmp	convert_end	

# Code Implementation

letters:

```
ldi temp1, 'A'  
add temp1, row  
rjmp convert_end
```

; Get the ASCII value for the key

symbols:

```
cpi col, 0  
breq star  
cpi col, 1  
breq zero  
ldi temp1, '#'  
rjmp convert_end
```

; Check if we have a star  
; or if we have zero  
; if not we have hash

star:

```
ldi temp1, '*'  
rjmp convert_end
```

; Set to star

zero:

```
ldi temp1, '0'
```

; Set to zero

convert\_end:

```
out PORTC, temp1  
;delay  
rjmp main
```

; Write value to PORTC  
; Restart main loop

# Reading Material

- Chapter 9: Computer Buses and Parallel Input and Output. Microcontrollers and Microcomputers by Fredrick M. Cady.
  - Simple I/O Devices

# Homework

1. Refer to the AVR Instruction Set manual, study the following instructions:
  - Arithmetic and logic instructions
    - lsr, ror
    - lsl, rol
  - Data transfer instructions
    - sts, lds

# Homework

2. Write an AVR assembly program to map the number-keys on the keypad to the individual LEDs on the LED bar. For example, when key 0 is pressed, LED 0 is turned on. After all number keys are pressed, all LEDs are on.

# **Microprocessors & Interfacing**

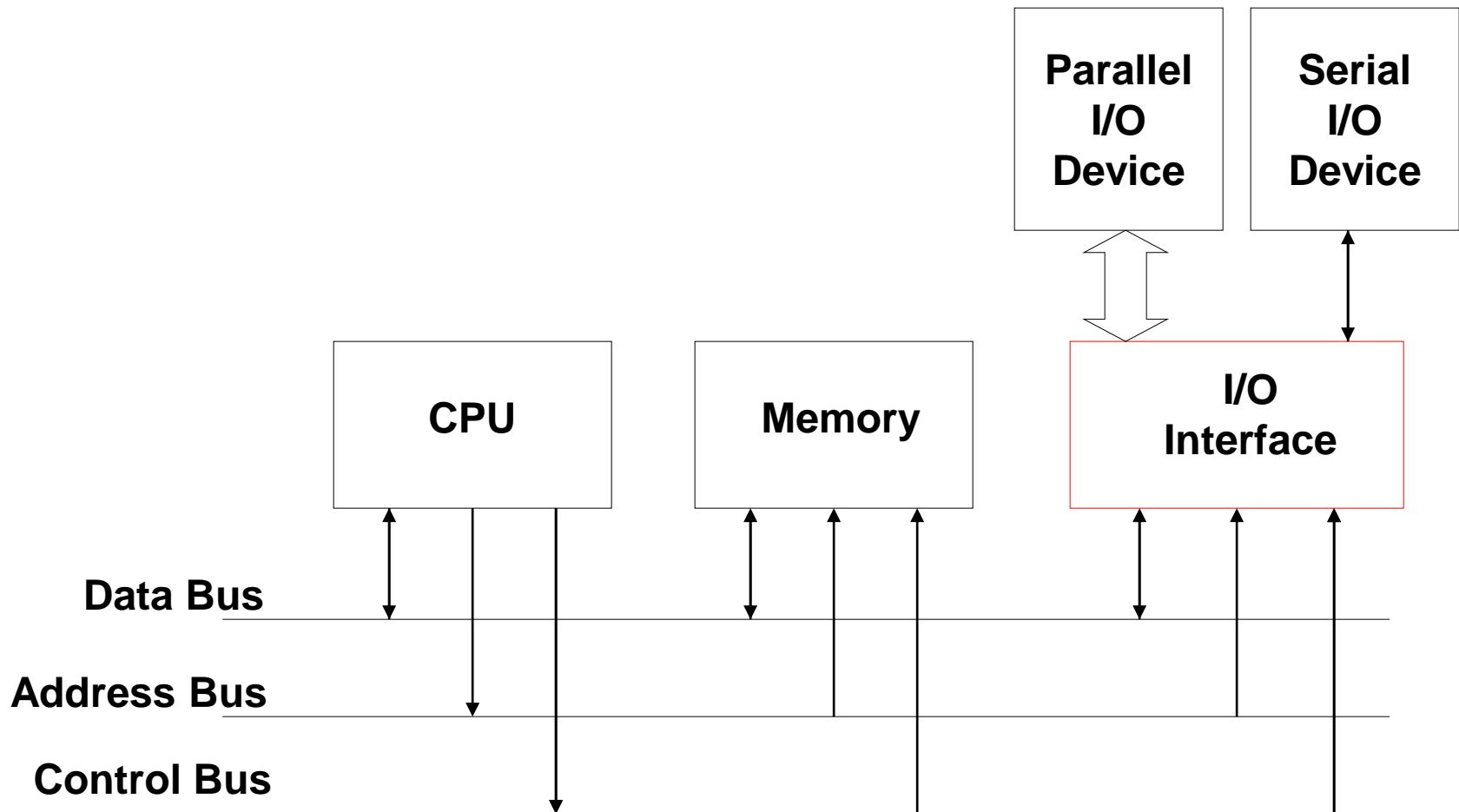
*Parallel Input/Output*

Lecturer : Annie Guo

# Lecture Overview

- I/O Addressing
  - Memory mapped I/O
  - Separate I/O
- Parallel Input/Output
  - AVR examples

# Typical Computer Structure

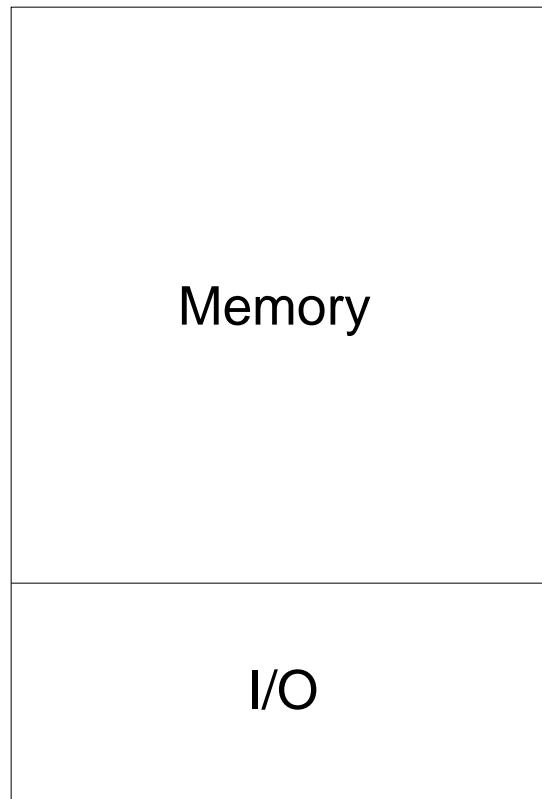


# I/O Addressing

- If the same address bus is used for both memory and I/O, how does hardware distinguish between memory reads/writes and I/O reads/writes?
  - Two approaches:
    - Memory-mapped I/O
    - Separate I/O
  - Both adopted in AVR

# Memory Mapped I/O

- The memory address space contains a section for I/O registers.



# AVR Memory Mapped I/O

- In AVR, 64+ I/O registers are mapped into memory space \$0020 ~ \$01FF
  - with 2-byte address
- With such memory addresses, the access to the I/O's registers uses memory-access type of instructions
  - E.g. *st* and *ld*

Address (HEX)	
32 Registers	0 - 1F
64 I/O Registers	20 - 5F
416 External I/O Registers	60 - 1FF
Internal SRAM (8192 × 8)	200
	21FF
External SRAM (0 - 64K × 8)	2200
	FFFF

# Memory Mapped I/O (cont.)\*

- Advantages:
  - Simple CPU design
  - No special instructions for I/O accesses
  - Scalable
- Disadvantages:
  - I/O devices reduce the amount of memory space available for application programs.
  - The address decoder needs to decode the full address bus to avoid conflict with memory addresses.

# Separate I/O

- Separate address space specifically for I/O.
  - Less expensive address decoders than those needed for memory-mapped I/O
- Special I/O instructions are required.

# Separate I/O (cont.)

- In AVR, the first 64 I/O registers can be addressed with the separate I/O addresses:  
\$00 ~ \$3F
  - 1-byte addresses
- With such separate addresses, the access to the I/O's registers uses I/O specific instructions.
  - *IN* and *OUT*

	Address (HEX)
32 Registers	0 - 1F
64 I/O Registers	20 - 5F
416 External I/O Registers	60 - 1FF
Internal SRAM (8192 × 8)	200
External SRAM (0 - 64K × 8)	21FF
	2200
	FFFF

# I/O Synchronization

- CPU is typically much faster than I/O devices.
- Therefore, synchronization between CPU and I/O devices is required.
- Two synchronization approaches:
  - Software
  - Hardware
    - To be covered later

# Software Synchronization

- Two basic methods:
  - Real-time synchronization
    - Uses a software delay to match CPU to the timing requirement of the I/O device.
      - The timing requirement must be known
      - Sensitive to CPU clock frequency
      - Consumes CPU time.
  - Polling I/O
    - A status register, with a DATA\_READY bit, is added to the device. The software keeps reading the status register until the DATA\_READY bit is set.
      - Not sensitive to CPU clock frequency
      - Still consumes CPU time, but CPU can do other tasks at the same time.
- Examples will be given later

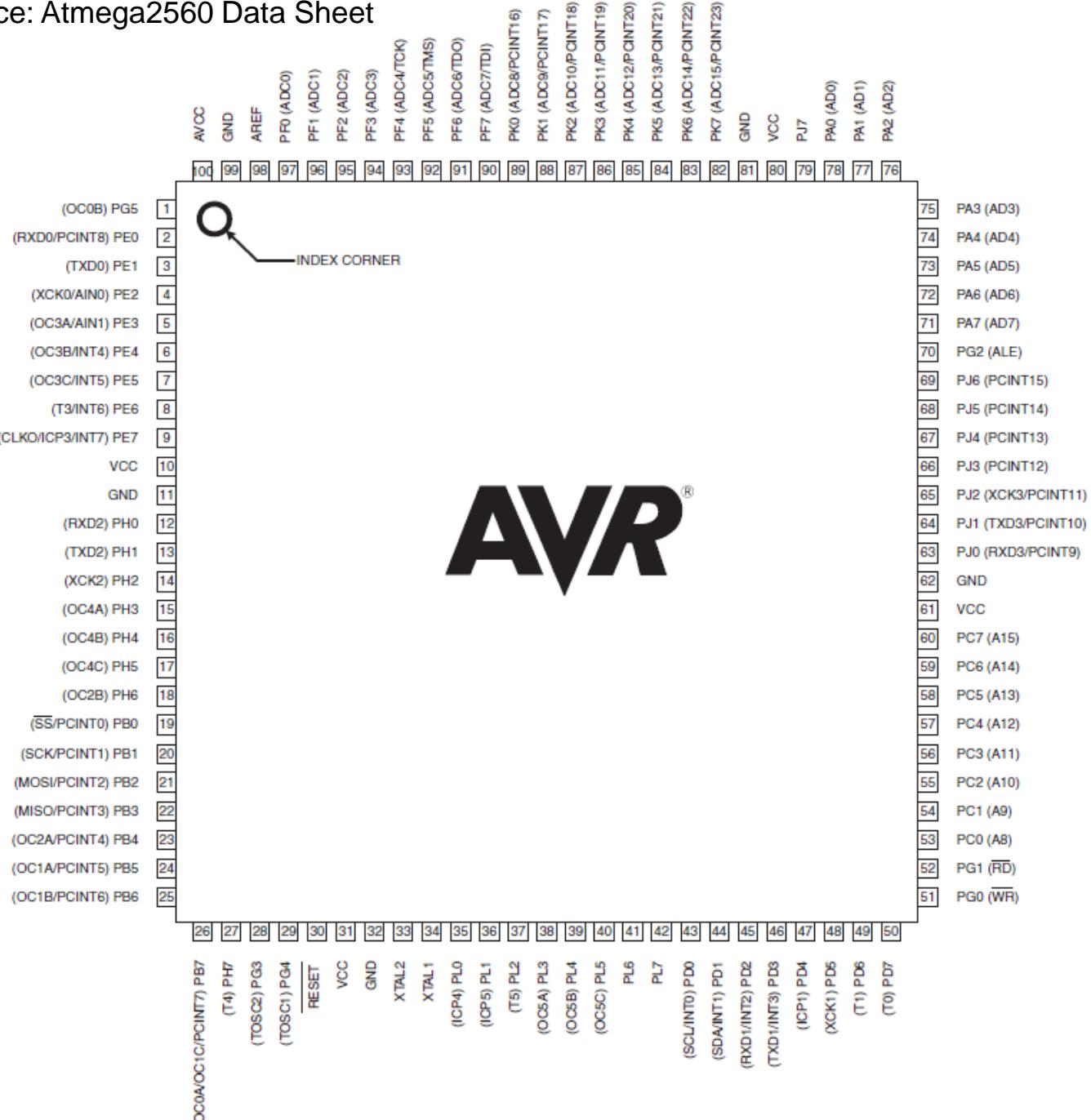
# Parallel Input/Output in AVR

- Communication through parallel port
- Two special instructions designed for parallel input/output operations
  - IN
  - OUT
- The port information is given in the next slides.

# Atmega2560

## Pin Configuration

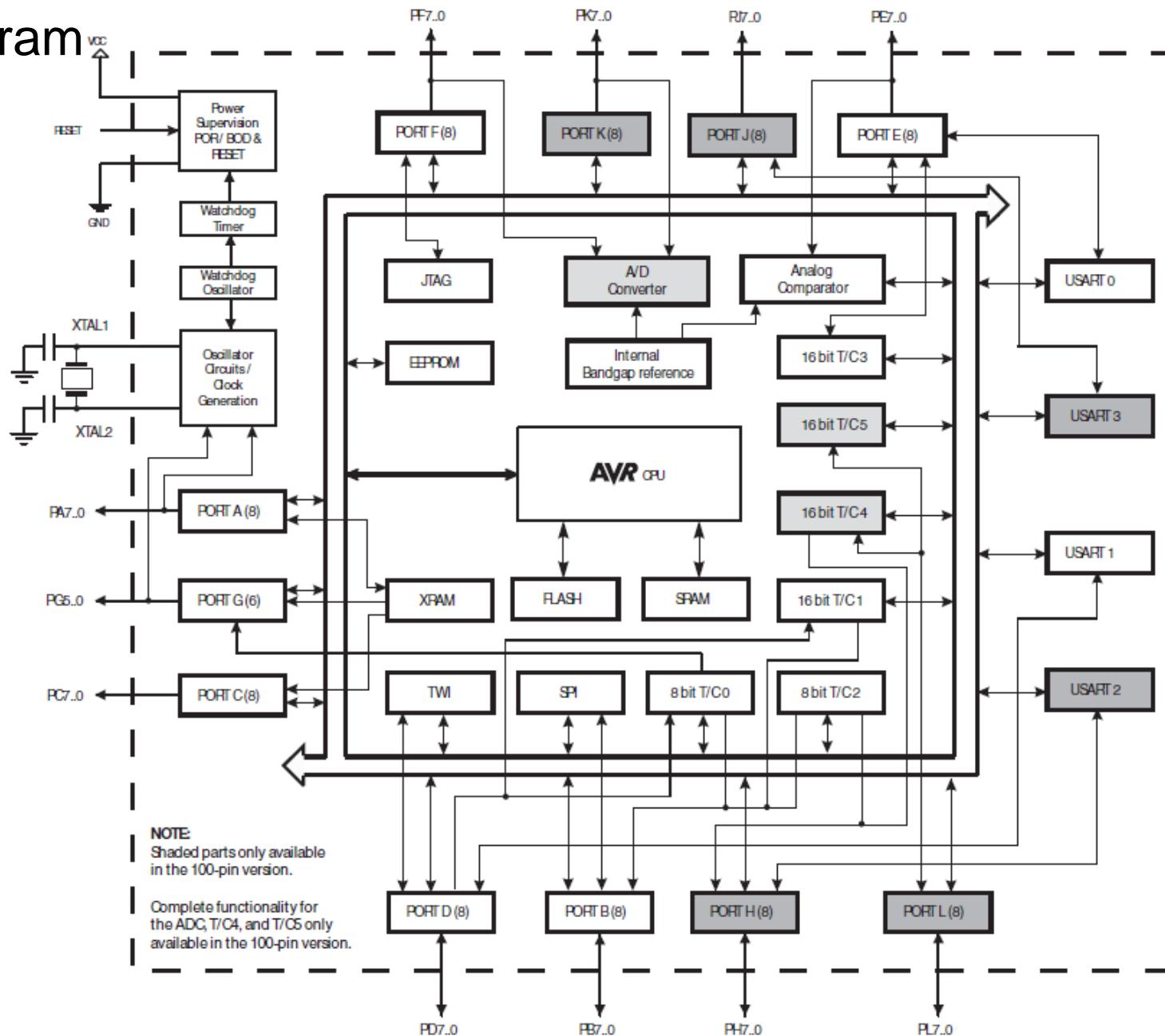
Source: Atmega2560 Data Sheet



# Atmega2560

## Block Diagram

Source: Atmega2560 Data Sheet



# AVR PORTs

- Can be configured to receive or send data
- Include physical pins and related circuitry to enable input/output operations.
- Different AVR microcontroller devices have different port design
  - ATmega2560 has 100 pins, most of them form ports for parallel input/output.
    - Port A to Port G (7 ports)
      - Having separate I/O addresses
        - » using *in* or *out* instructions
      - Port H to Port L (5 ports)
        - Only having memory-mapped addresses
      - Three I/O addresses are allocated for each port. For example, for Port x, the related three registers are:
        - PORTx: data register
        - DDRx: data direction register
        - PINx: input pin register

# Load I/O Data to Register

- Syntax:  $\text{in } Rd, A$
- Operands:  $0 \leq d \leq 31, 0 \leq A \leq 63$
- Operation:  $Rd \leftarrow \text{I/O}(A)$

- Words: 1
- Cycles: 1
- Example:

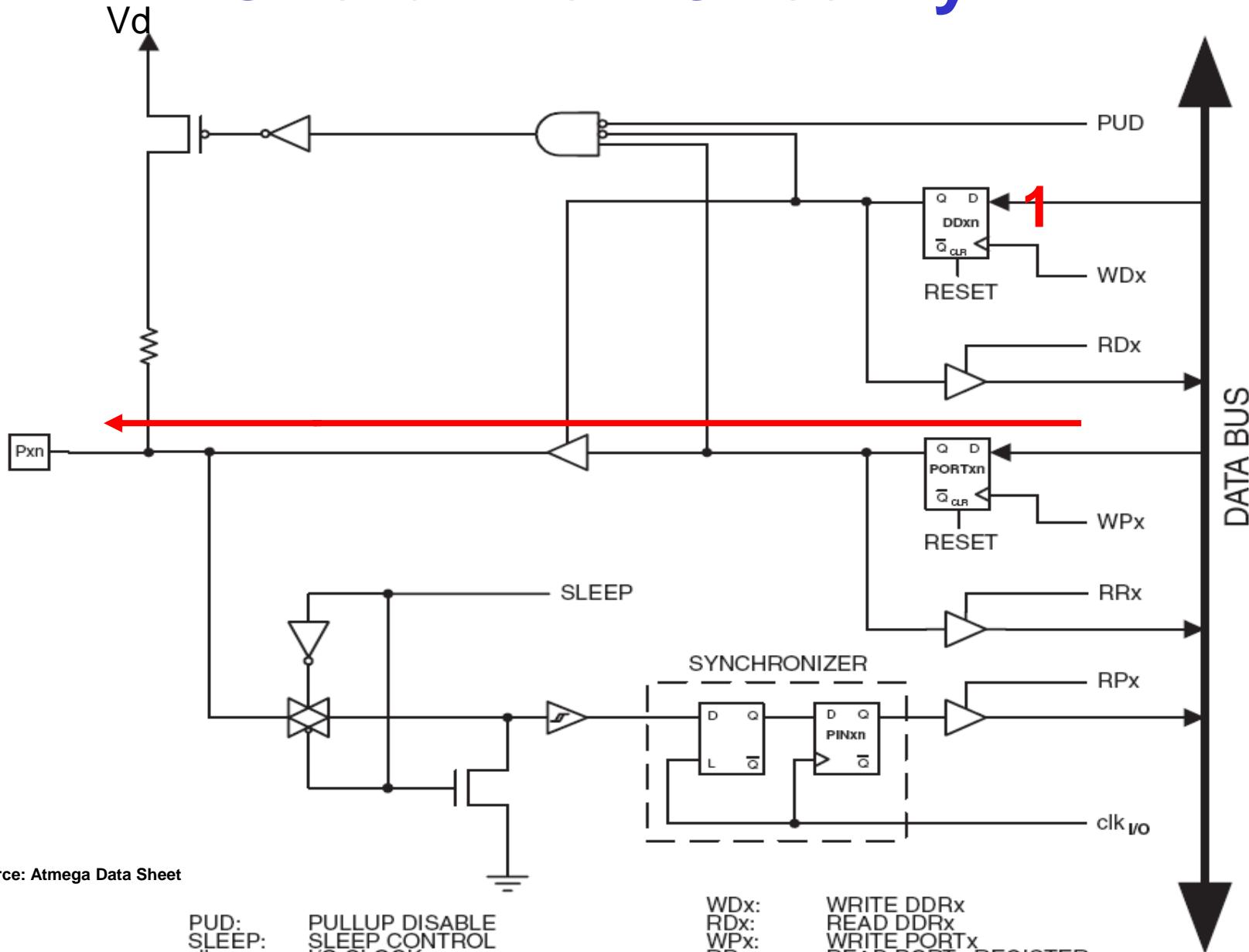
`in r25, 0x03 ; read port B`

- The names of the I/O ports are given in the device definition file, [m2560def.inc](#).
- 0x03 is an I/O register address of port B

# Store Register Data to I/O Location

- Syntax:  $\text{out } A, Rr$
  - Operands:  $0 \leq r \leq 31, 0 \leq A \leq 63$
  - Operation:  $I/O(A) \leftarrow Rr$
  - Words: 1
  - Cycles: 1
  - Example:  
`out 0x05, r16 ; write to port B`

# One-bit Port Circuitry\*



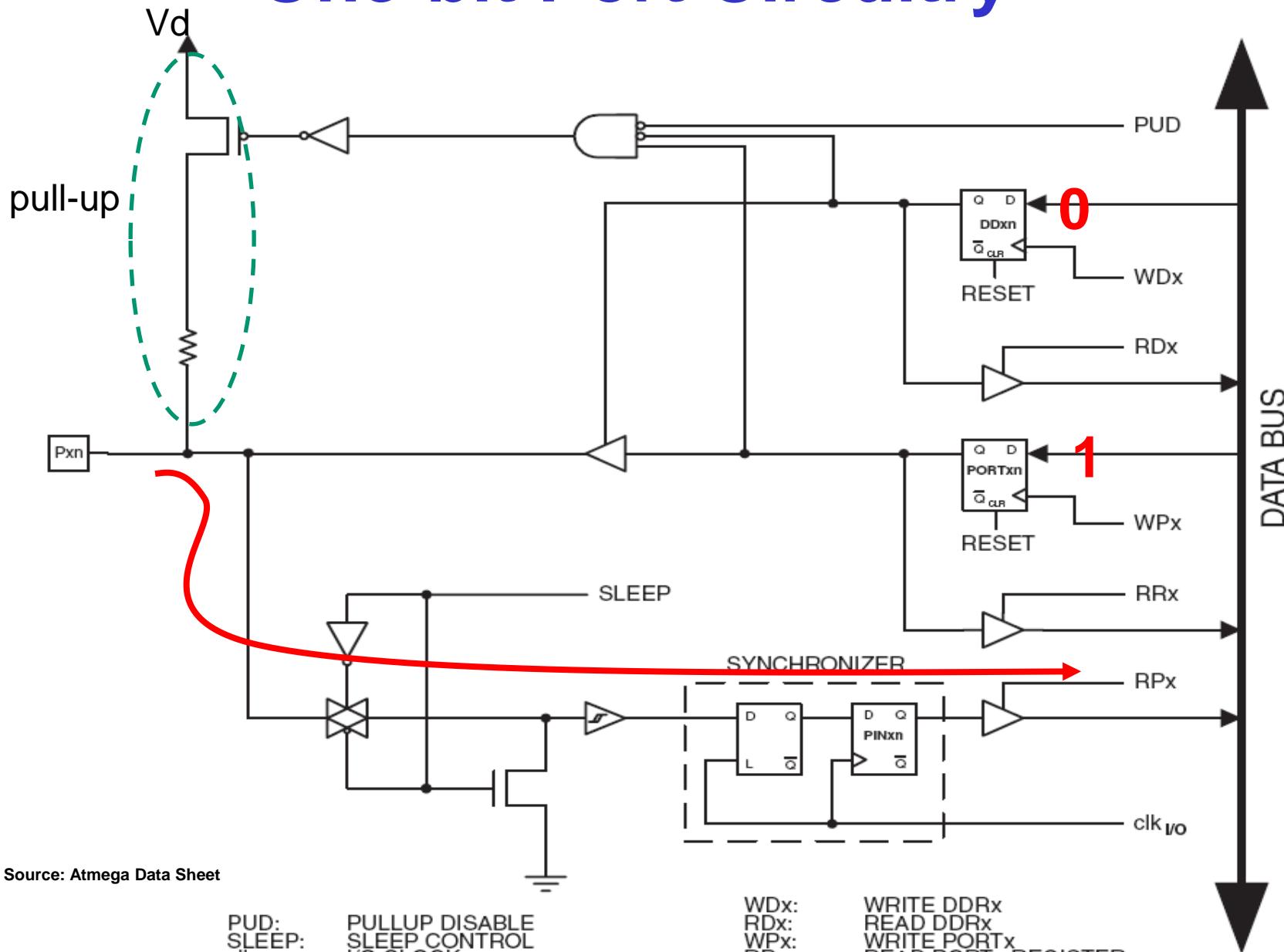
Source: Atmega Data Sheet

PUD:  
SLEEP:  
clk<sub>I/O</sub>:

PULLUP DISABLE  
SLEEP CONTROL  
I/O CLOCK

WDx:	WRITE DDRx
RDx:	READ DDRx
WPx:	WRITE PORTx
RRx:	READ PORTx REGISTER
RPx:	READ PORTx PIN

# One-bit Port Circuitry\*



Source: Atmega Data Sheet

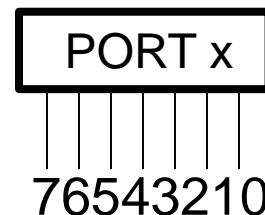
PUD:  
SLEEP:  
clk<sub>I/O</sub>:

PULLUP DISABLE  
SLEEP CONTROL  
I/O CLOCK

WDx:	WRITE DDRx
RDx:	READ DDRx
WPx:	WRITE PORTx
RRx:	READ PORTx REGISTER
RPx:	READ PORTx PIN

# How does it work?

- The circuit for each bit input/output operation in the I/O port consists of three register bits.  
E.g. for pin  $n$  of port  $x$ , we have
  - DDR $xn$ , PORT $xn$ , and PIN $xn$ .
- The DDR $xn$  bit in the DDR $x$  Register selects the direction of this pin.
  - If DD $xn$  is set to 1, the pin is configured as an output pin. If DD $xn$  is set to 0, the pin is configured as an input pin.



# How does it work?\* (cont.)

- When the pin is configured as an input pin, the pull-up resistor can be activated/deactivated.
- To active pull-up resistor for input pin, PORTxn needs to be written to 1.

# Sample Code for Output

```
.include "m2560def.inc"

clr      r16          ; clear r16
ser      r17          ; set r17
out      DDRA, r17    ; set Port A for output operation

out      PORTA, r16   ; write zeros to Port A
nop
out      PORTA, r17   ; write ones to Port A

...
```

# Sample Code for Input

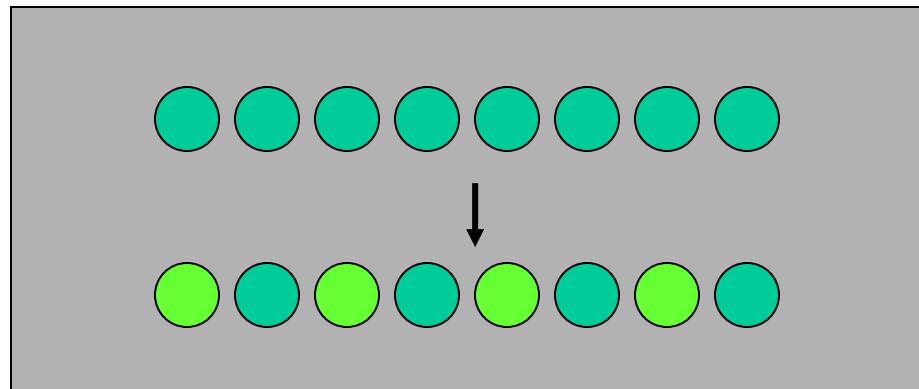
```
.include "m2560def.inc"

    clr      r15
    out      DDRA, r15      ; set Port A for input operation

    in       r25, PINA      ; read Port A
    cpi      r25, 4         ; compare read value with constant
    breq    exit            ; branch if r25=4
    ...
exit:   nop              ; branch destination (do nothing)
```

# Example 1

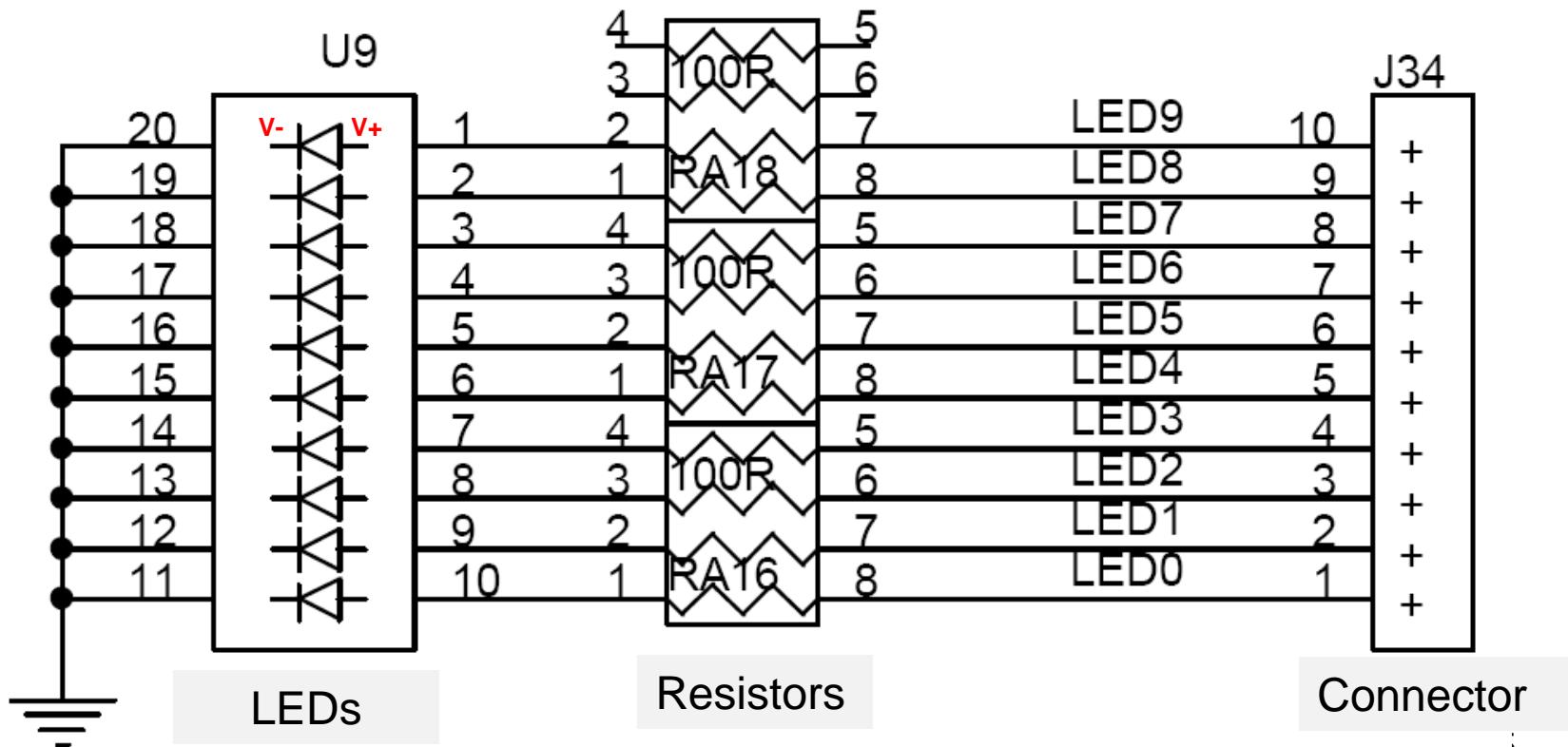
- Design a simple control system that can control a set of LEDs to display a fixed pattern.



• LED: light-emitting diode

# LED and Its Operation

- For each LED, when its  $V_+ > V_-$ , it will emit light.



# Example 1 (solution)

- The design consists of several steps:
  - 1) Set a port for the output operation; One pin of the port is connected to one LED.
  - 2) Write the pattern value to the port so that it can drive the LEDs to display the related pattern.

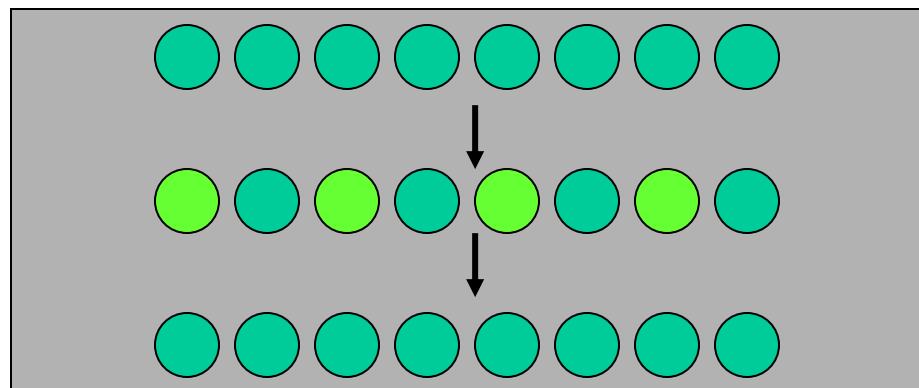
```
.include "m2560def.inc"
    ser r16                      ; set all bits in r16 to 1
    out DDRB, r16                 ; set Port B for output

    ldi r16, 0xAA                ; write the pattern
    out PORTB, r16

end:
    rjmp end
```

## Example 2

- Design a simple control system that can control a set of LEDs to display a fixed pattern for *one second and then turn the LEDs off.*



# Example 2 (solution)

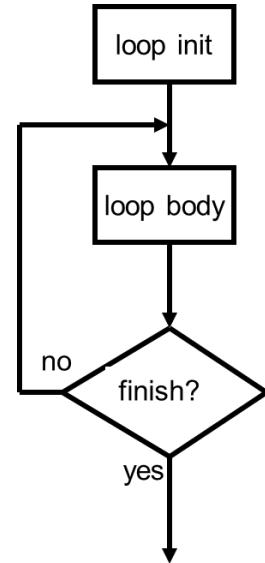
- The design consists of several steps:
  - 1) Set a port for the output operation; One pin of the port is connected to one LED
  - 2) Write the pattern value to the port so that it can drive the display of LEDs
  - 3) *Wait for one second*
    - *To be discussed in the following slides*
  - 4) *Write pattern, 00000000, to set all LEDs off.*

# Counting One Second

- Basic idea:
  - **Assume** the clock cycle period is 1ms (very very slow, not a real value). Then we can write a program that executes  $(\frac{1}{10^{-3}} = 10^3)$  cycles.
    - Namely, execution of the code will take 1 second
    - If each instruction takes one clock cycle, 1000 instructions will be executed
- An AVR implementation example is given in the next slide, **where the 1ms clock cycle time is assumed.**

# Code for One Second Delay ( $T_{clock}=1\text{ms}$ )

```
.include "m2560def.inc"
.equ loop_count = 124 ?
.def iH = r25
.def iL = r24
.def countH = r17
.def countL = r16
.macro oneSecondDelay
    ldi countL, low(loop_count)          ; 1 cycle
    ldi countH, high(loop_count)
    clr iH                             ; 1
    clr iL
loop:   cp iL, countL                ; 1
        cpc iH, countH
        brsh done                      ; 1, 2 (if branch)
        adiw iH:iL, 1                  ; 2
        nop                            ; 1
        rjmp loop                      ; 2
done:
.endmacro
```



# Code for Example 2

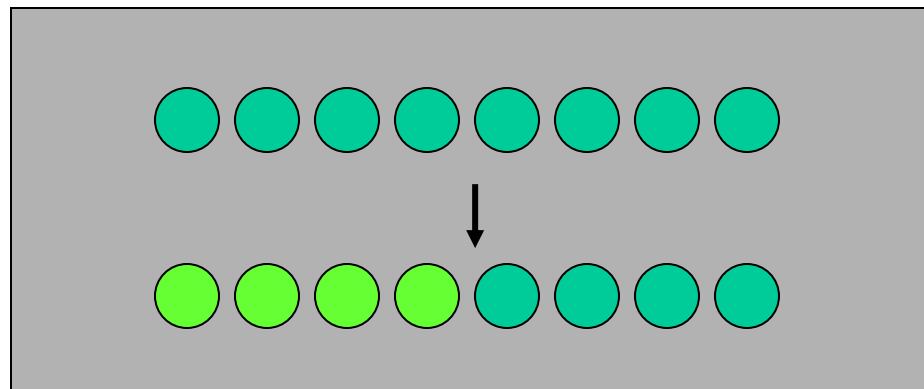
```
.include "m2560def.inc"
...
    ser r15                                ; macro oneSecondDelay
    out DDRB, r15                           ; set Port B for output

    ldi r15, 0xAA                          ; write the pattern
    out PORTB, r15
    oneSecondDelay                         ; 1 second delay
    ldi r15, 0x00                          ; turn off the LEDs
    out PORTB, r15

end:
    rjmp end
```

# Example 3

- Design a simple control system that can control a set of LEDs to display a fixed pattern *that is specified by the user.*
  - Assume there are switches. Each switch can provide two possible values (switch-on for logic 1 and switch-off for logic 0)



# Example 3 (solution)

- Design
  - 1) Connect the switches to the pins of a port
  - 2) Set the port for input
  - 3) Read the input
  - 4) Set another port for the output operation; Each pin of the port is connected to one LED
  - 5) Write the pattern value provided by the input switches to the port so that it can drive the display of LEDs
- Execution
  - Set the switches for a desired input value
  - Start the control system

# Code for Example 3

```
.include "m2560def.inc"

    clr r17
    out DDRC, r17          ; set Port C for input
    ser r17
    out PORTC, r17         ; activate the pull up

    in r17, PINC           ; read the pattern set by the user
                           ; from the switches
    ser r16
    out DDRB, r16          ; set Port B for output

    out PORTB, r17         ; write the input pattern

end:
    rjmp end
```

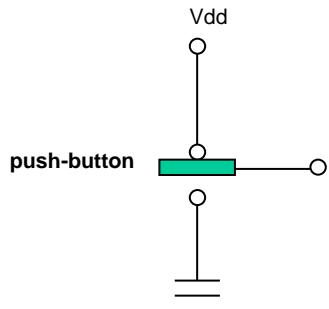
# Example 4

- Design a simple control system that can control a set of LEDs to display a pattern specified by the user *during the execution.*

# Example 4 (solution)

- Use polling to handle dynamic input
  - The processor continues checking if there is an input for read. If there is, the processor reads the input and goes to next task, otherwise the processor is in a waiting state for the input.

# Example 4 (solution)



- Design
  - 1) Set one port for input and connect each pin of the port to one switch
  - 2) Set another port for the output operation; Each pin of the port is connected to one LED
  - 3) Set a pin for input and connect the pin to the push-button,
    - When the button is pressed, it indicates “Input Pattern is ready”
  - 4) Poll the pin until “Input Pattern is ready”
  - 5) Read the input pattern
  - 6) Write the pattern to the port so that it can drive the display of LEDs
- During execution
  - Set the switches for the input value
  - Press the push button
  - The LEDs will show the pattern as specified by the user.

# Code for Example 4

```
.include "m2560def.inc"

        cbi DDRD, 7          ; set Port D bit 7 for input
        clr r17
        out DDRC, r17        ; set Port C for input
        ser r17
        out PORTC, r17       ; activate the pull up
        out DDRB, r17         ; set Port B for output

waiting:
        sbic PIND, 7          ; check if that bit is clear
        rjmp waiting           ; if yes skip the next instruction
                                ; waiting

        in r17, PINC          ; read pattern set by the user
                                ; from the switches
        out PORTB, r17
        rjmp waiting
```

# Reading Materials

- Chapter 9: Computer Buses and Parallel Input and Output. Microcontrollers and Microcomputers by Fredrick M. Cady.
- Mega2560 Data Sheet
  - Ports

# Homework

1. Refer to the AVR Instruction Set manual, study the following instructions:
  - Arithmetic and logic instructions
    - ser
  - Data transfer instructions
    - in, out
  - Bit operations
    - sbi, cbi
  - Program control instructions
    - sbic, sbis
  - MCU control instructions
    - nop

# Homework

3. Refer to “Introduction to Lab Board”. Study the lab board. Write the assembly code to display pattern 10110111 on the LED bar through each of the following I/O ports:
- (a) port C
  - (b) port F
  - (c) port L

You are strongly encouraged to work with your lab group members to complete this task.

# **Microprocessors & Interfacing**

*AVR Programming (IV)*

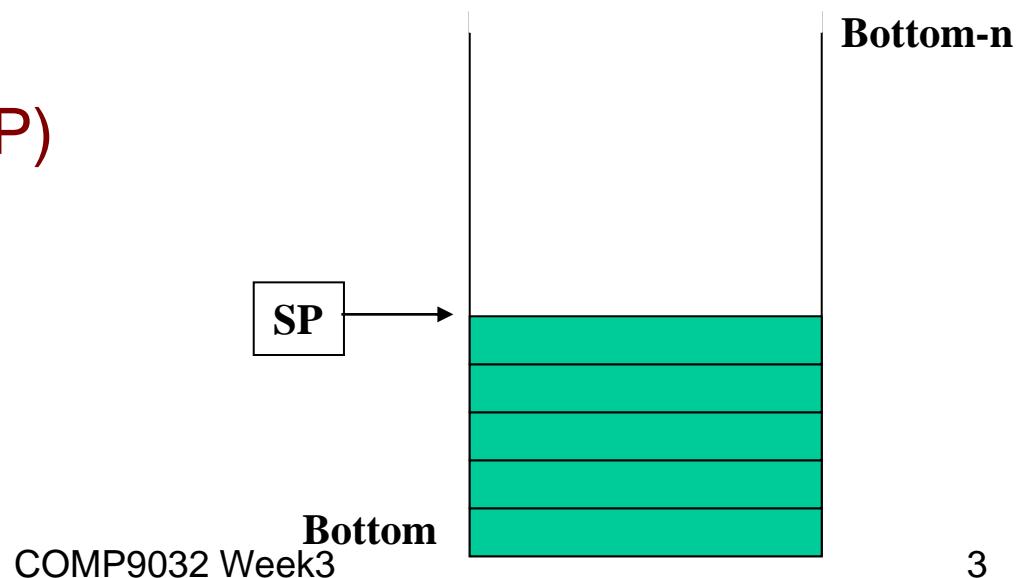
Lecturer : Annie Guo

# Lecture Overview

- Stack and stack operation
- Assembly function and function call
  - Calling convention
  - Examples

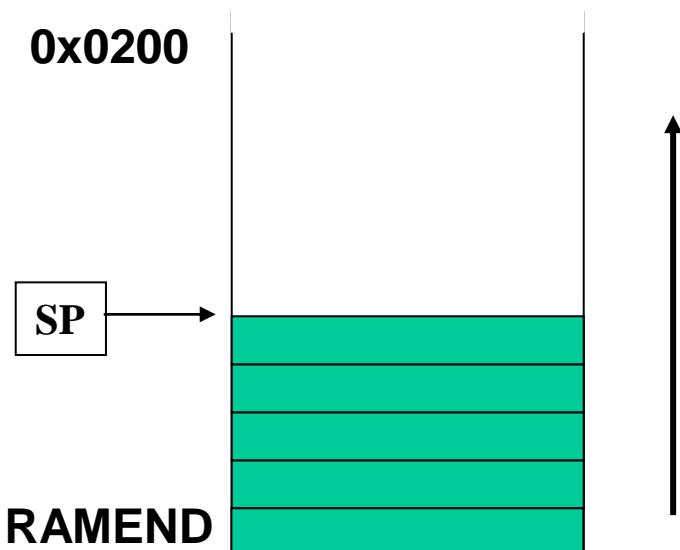
# Stack

- What is stack?
  - A data structure in which a data item that is Last In is First Out (LIFO)
- In AVR, a stack is implemented as a block of consecutive locations in the data memory
- A stack has at least two parameters:
  - Bottom
  - Stack pointer (SP)



# Stack Bottom

- The stack usually *grows from high addresses to low addresses*
- The stack bottom is the location with the highest address in the stack
- In AVR, 0x0200 is the lowest address for stack
  - i.e. stack bottom  $\geq 0x0200$



# Stack Pointer

- In AVR, the stack pointer,  $SP$ , is an I/O register pair,  $SPH:SPL$ , they are defined in the device definition file
  - m2560def.inc
- Default value of the stack pointer is 0x21FF
- The stack pointer always points to the top of the stack
  - Definition of the stack top varies:
    - the location of the Last-In element;
      - E.g. in 68K
    - the location available for the next element to be stored
      - E.g. in AVR

# Stack Operations

- There are two stack operations:
  - Push
    - Implemented by instruction *PUSH*
  - Pop
    - Implemented by instruction *POP*

# PUSH

- Syntax:  $\text{push } Rr$
- Operands:  $Rr \in \{r0, r1, \dots, r31\}$
- Operation:  $(SP) \leftarrow Rr$   
 $SP \leftarrow SP - 1$
- Words: 1
- Cycles: 2

# POP

- Syntax:  $\text{pop } Rd$
- Operands:  $Rd \in \{r0, r1, \dots, r31\}$
- Operation:  $SP \leftarrow SP + 1$   
 $Rd \leftarrow (SP)$
- Words: 1
- Cycles: 2

# Functions

- Stack is used in function calls
- Functions are used
  - in top-down design
    - Conceptual decomposition - easy to design
  - for modularity
    - Readability and maintainability
  - for reuse
    - Design once and use many times
      - Common code with parameters
    - **Store once** and use many times
      - Saving code size, hence memory space

# C Code Example

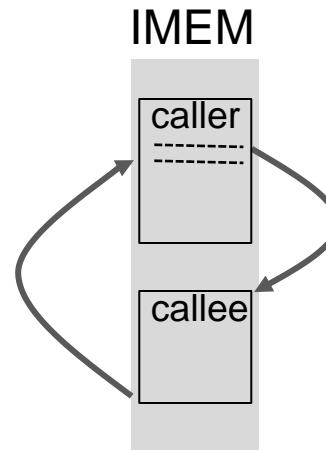
```
unsigned int pow(unsigned int b, unsigned int e) {          //parameters b & e,  
    unsigned int i, p;                                     // returns an integer  
    p = 1;                                                 // local variables  
    for (i=0; i<e; i++)                                    // p = be  
        p = p*b;  
    return p;                                              // return value of the function  
}  
  
int main(void) {  
    unsigned int m, n;  
    m = 2;  
    n = 3;  
    m = pow(m, n);  
    return 0;  
}
```

# C Code Example (cont.)

- In this program:
  - Caller
    - main
  - Callee
    - pow
  - Passing parameters
    - b, e
  - Return value
    - p

# Function Call

- A function call involves
  - program flow control between caller and callee
    - target/return addresses
  - value passing
    - parameters/return values
- Certain rules/conventions are used for implementing functions and function calls.
  - Often used by the compiler



# Rules (I)

- Use **stack** for parameter passing
- Registers can be used as well for parameter passing
  - For example, WINAVR uses
    - registers r8 ~ r25 to store passing parameters
    - r25:r24 to store the return value
  - The parameters may eventually be saved on the stack to free registers.
- Some parameters that are used in several places in the program must be saved in the stack.
  - E.g. input to recursive call

# Rules (II)

- Parameters can be passed by *value* or *reference*
  - Passing by value
    - Pass the value of a parameter to the callee
      - Not efficient for structures and arrays
        - » Need to pass the value of each element in the structure or array
  - Passing by reference
    - Pass the address of the parameter to the callee
    - Efficient for structures and array passing
    - Using *passing by reference* when the parameter is to be modified by the function
      - Example is given in the next two slides

# Example: Passing by Value

- C program

```
void swap(int x, int y){          // the swap(x,y) in fact
    int temp = x;                // does not work since
    x = y;                      // the new x, y values
    y = temp;                   // are not copies back.

    return;
}

int main(void) {
    int a = 1, b = 2;
    swap(a,b);
    printf("a=%d, b=%d", a, b)
    return 0;
}
```

# Example: Passing by Reference

- C program

```
swap(int *px, int *py){  
    int temp;  
    temp = *px  
    *px = *py;  
    *py = temp;  
    return;  
}  
  
int main(void) {  
    int a = 1, b = 2;  
    swap(&a,&b);  
    printf("a=%d, b=%d", a, b)  
    return 0;  
}
```

// call by reference  
// allows callee to change  
// the value in caller, since the  
// “referenced” memory  
// is altered.

# Rules (III)

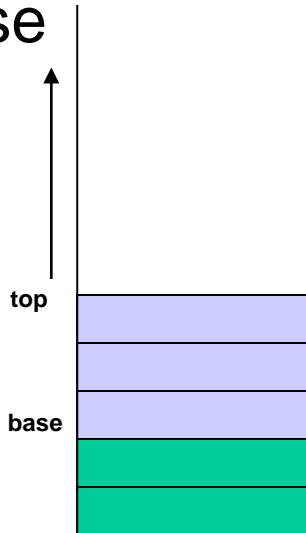
- If a register is being used by both caller and callee and the caller needs its old value after the callee returns, then a *register conflict* occurs.
- Compilers or assembly programmers need to
  - check for register conflict, and
  - save conflict registers on the stack
- Caller or callee or both can save conflict registers.
  - In WINAVR, callee saves conflict registers

# Rules (IV)

- Local variables and parameters need to be stored contiguously on the stack for easy access.
- How are the local variables or parameters stored on the stack?
  - In the order that they appear in the high-level program from left to right, or the reverse order.
  - Either is OK. But the consistency should be maintained.
  - Example will be provided later

# Stack Frame and Function Call

- Each function call creates a *stack frame* in the stack.
- The stack frame occupies some space and has an associated pointer, called *stack frame pointer*.
  - WINAVR uses **Y (r29: r28)** as the stack frame pointer
- The stack frame space is freed when the function returns.
- The stack frame pointer can point to either the base (starting address) or the top of **the stack frame**
  - In AVR, it points to the top of the stack frame



# Typical Stack Frame Contents

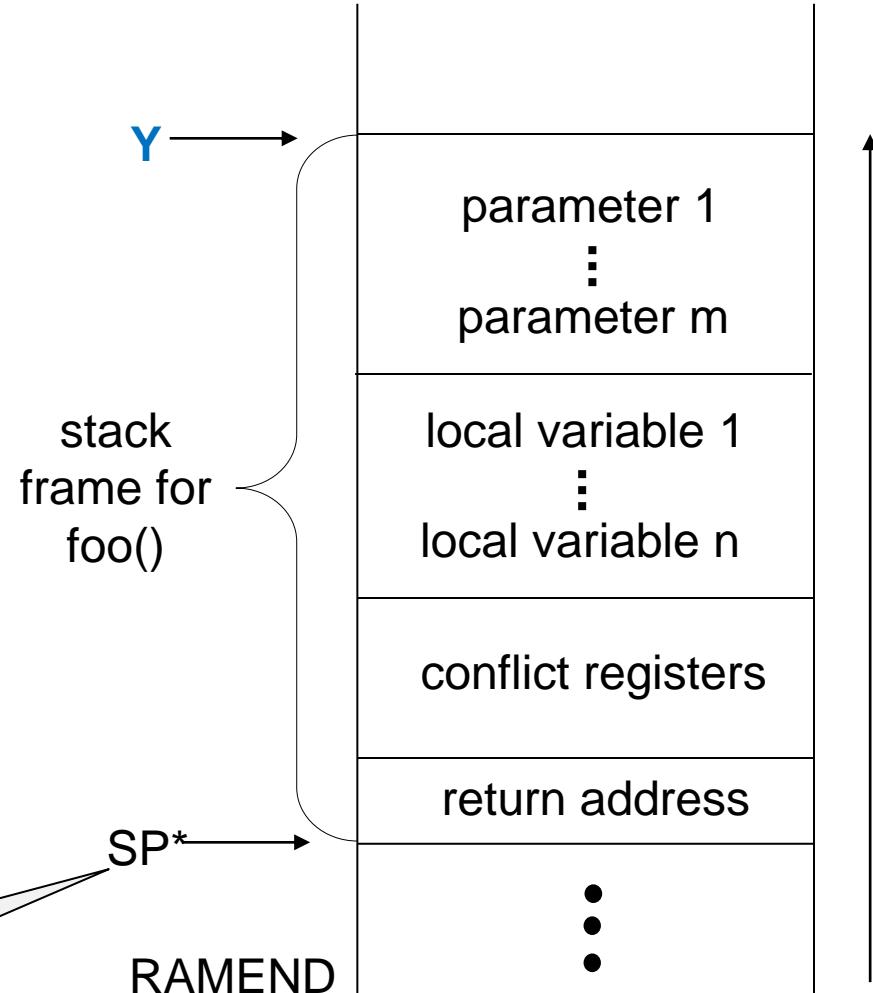
- Return address
  - To be used when the function returns
- Conflict registers
  - E.g. the stack frame pointer
  - The original contents of these registers need to be restored when the function returns
- Optionally, parameters
- Optionally, local variables

# Stack Frame Structure: an example

```
int main(void)
{
    ...
    foo(arg1, arg2, ..., argm);
}

void foo(arg1, arg2, ..., argm){
    int var1, var2, ..., varn;
    ...
}
```

SP before function call



# A Template for Caller

Basic operations by caller:

- Before calling the callee, store passing parameters in the designated registers
- Call callee.
  - Using instructions for function call
    - rcall, icall, call.

# Relative Call to Function

- Syntax:  $\text{rcall } k$
- Operands:  $-2K \leq k < 2K$
- Operation:  
 $\text{stack} \leftarrow \text{PC}+1, \text{SP} \leftarrow \text{SP}-2$   
 $\text{PC} \leftarrow \text{PC}+k+1$
- Words: 1
- Cycles: 3
- For device with 16-bit PC

# A Template for Callee

Callee (function):

- Prologue
- Function body
- Epilogue

# A Template for Callee (cont.)

## Prologue:

- Save conflict registers, including the stack frame pointer on the stack by using *push* instruction
- If required, allocate space for local variables and passing parameters
  - by updating the stack pointer SP
    - $SP = SP - \text{the size of all parameters and local variables.}$
    - Using *OUT* instruction
- Update the stack frame pointer Y to point to the top of its stack frame and the stack pointer SP to point to the top of the stack
- If required, pass the parameters' values to the parameters' locations on the stack

## Function body:

- Perform the task of the function on the stack frame and registers.

# A Template for Callee (cont.)

## Epilogue:

- Store the return value in the designated registers
- De-allocate the stack frame
  - Deallocate the space reserved for local variables and parameters by updating the stack pointer SP.
    - $SP = SP + \text{the size of all parameters and local variables.}$
    - Using *OUT* instruction
  - Restore conflict registers from the stack by using *pop* instruction
    - The conflict registers must be popped in the reverse order that they were pushed on the stack.
      - The stack frame pointer register of the caller is also restored.
- Return to the caller by using *ret* instruction

# Return from Subroutine Instruction

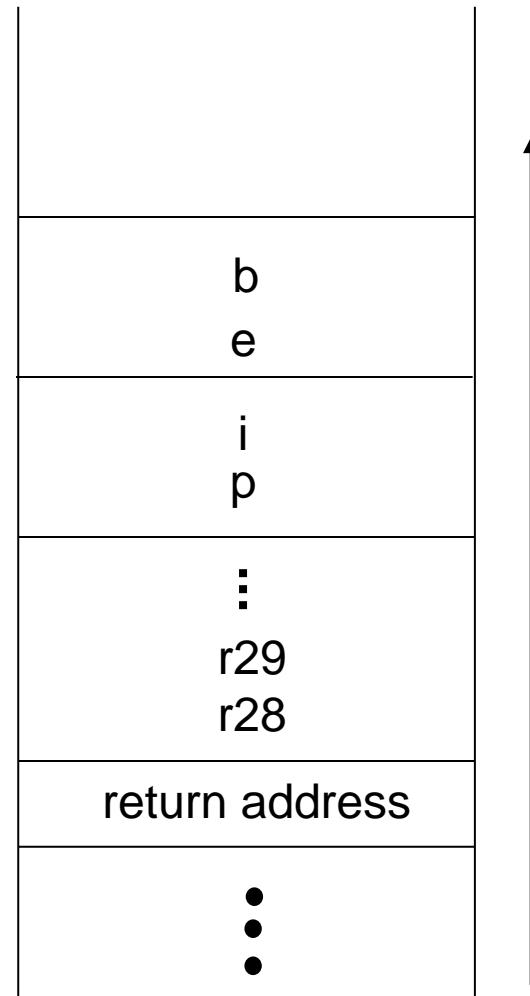
- Syntax:            ***ret***
- Operands:        none
- Operation:        $PC \leftarrow (SP), SP \leftarrow SP+2,$
- Words:            1
- Cycles:           4
- For device with 16-bit PC

# Example 1

- C program (pow function)
  - Assume an integer takes two bytes

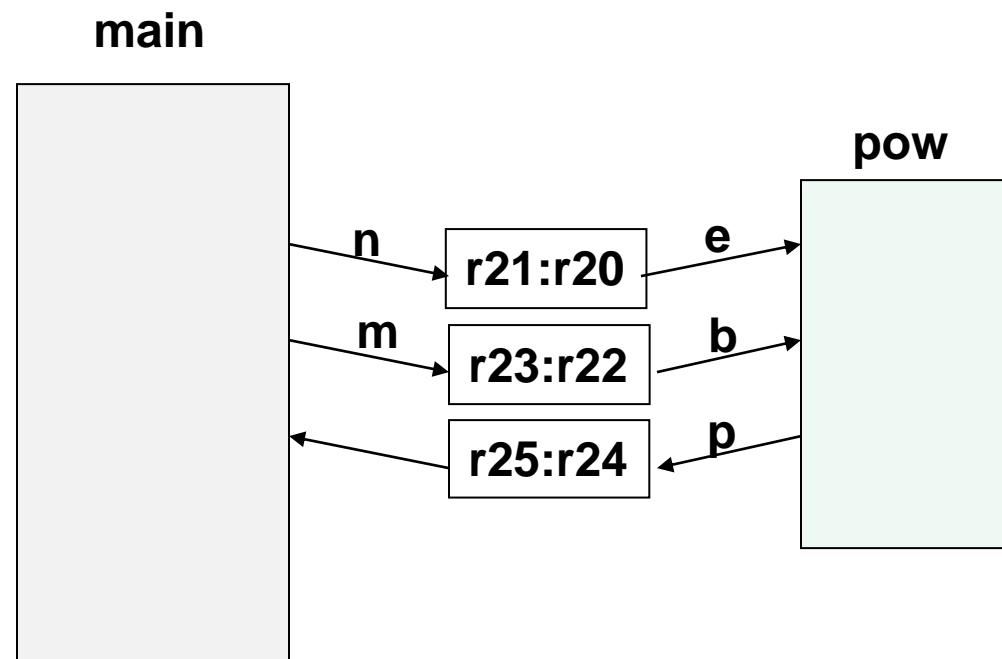
```
unsigned int pow(unsigned int b, unsigned int e) {  
    // parameters b & e,  
    // returns an integer  
    // local variables  
    unsigned int i, p;  
    p = 1;  
    for (i=0; i<e; i++)  
        p = p*b;  
    return p;  
    // return value of the function  
}  
  
int main(void) {  
    unsigned int m, n;  
    m = 2;  
    n = 3;  
    m = pow(m, n);  
    return 0;  
}
```

# Stack Frame for pow()



# Parameter Passing

- Assume an integer takes two bytes



# Example 1 (cont.)

- Assembly program
  - Assume an integer takes two bytes

```
.include "m2560def.inc"
.equ m = 2
.equ n = 6

; Macro mul2: multiplication of two 2-byte unsigned numbers with a 2-byte result
; All parameters are registers, @5:@4 should be in the form: rd+1:rd, where d is
; the even number, and rd+1:rd are not r1:r0.
; Operation: (@5:@4) = (@1:@0)*(@3:@2)

.macro mul2          ; a * b
    mul  @0, @2        ; al * bl
    movw @5:@4, r1:r0
    mul  @1, @2        ; ah * bl
    add  @5, r0
    mul  @0, @3        ; bh * al
    add  @5, r0
.endmacro            ; @5 @4
```

@1 @0	X @3 @2	
		@0x@2
		@1x@2
		@0x@3
+	@5 @4	@1x@3

# Example 1 (cont.)

- Assembly program
  - Assume an integer takes two bytes

```
; main
ldi r22, low(m)           ; m = 2
ldi r23, high(m)
ldi r20, low(n)           ; n = 6
ldi r21, high(n)
rcall pow                  ; Call function pow
movw r23:r22, r25:r24    ; Get the return result
end:
rjmp end                  ; end of main
```

# Example 1 (cont.)

- Assembly program
  - Assume an integer takes two bytes

pow:

; Prologue:

push r29:r28 ; r29:r28 will be used as the frame pointer

push YL ; Save r29:r28 in the stack

push YH

push r16

; Save registers used in the function body

push r17

push r18

push r19

in YL, SPL

; Initialize the stack frame pointer value

in YH, SPH

sbiw Y, 8

; Reserve space for local variables

; and parameters.

# Example 1 (cont.)

- Assembly program

```
out SPH, YH      ; Update the stack pointer to  
out SPL, YL      ; point to the new stack top  
  
                      ; Pass the actual parameters  
std Y+1, r22      ; Pass m to b  
std Y+2, r23  
std Y+3, r20      ; Pass n to e  
std Y+4, r21  
; End of prologue
```

# Example 1 (cont.)

- Assembly program

```
; Function body  
  
clr r23; ; Use r23:r22 for i and r25:r24 for p,  
clr r22; ; r21:r20 temporarily for e and r17:r16 for b  
clr r25; ; Initialize i to 0  
ldi r24, 1 ; Initialize p to 1  
... ; Store the local values to the stack  
     ; if necessary  
  
ldd r21, Y+4 ; Load e to registers  
ldd r20, Y+3  
ldd r17, Y+2 ; Load b to registers  
ldd r16, Y+1
```

# Example 1 (cont.)

- Assembly program

```
loop:    cp r22, r20          ; Compare i with e
          cpc r23, r21
          brsh done           ; If i >= e
          mul2 r24,r25, r16, r17, r18, r19   ; p *= b
          movw r25:r24, r19:r18
          subi r22, Low(-1)      ; i++
          sbci r23, High(-1)
          rjmp loop

done:    ; End of function body
```

# Example 1 (cont.)

- Assembly program

```
; Epilogue
```

```
adiw Y, 8          ; De-allocate the reserved space
out SPH, YH
out SPL, YL
pop r19
pop r18          ; Restore registers
pop r17
pop r16
pop YH
pop YL
ret              ; Return to main()
; End of epilogue
```

# Remarks about Example 1 code

- The stack frame was designed by following the typical structure
  - i.e., without knowing how the function body is going to be implemented
- The code can be further refined and optimized
  - For specific function implementation
    - For example, the space reserved for local variable i, p are never used in the function body and can be removed from the stack frame, hence saving memory space and the execution time.

# Recursive Function

- A recursive function
  - is both a caller and a callee of itself
  - is formed by a looped function calls
  - has a termination point or base case
- It can be hard to compute the maximum stack space needed for a recursive function call.
  - Need to know how many times the function is nested (the depth of the call).
  - And it often depends on the input values of the function
- An example is given next

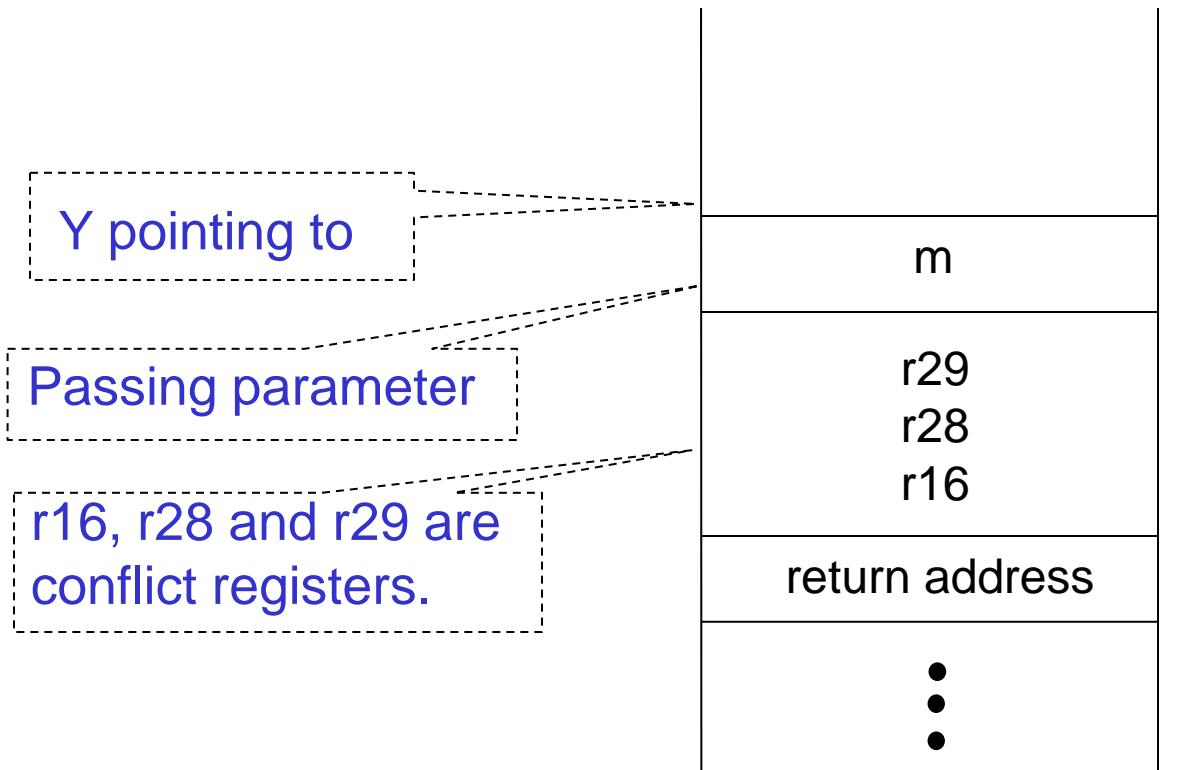
# Example 2

- C program (Fibonacci number function)
  - Assume an integer takes one byte

```
int n = 12;  
  
void main(void)  
{  
    fib(n);  
}  
  
  
int fib(int m)  
{  
    if(m == 0) return 1;  
    if(m == 1) return 1;  
    return (fib(m - 1) + fib(m - 2));  
}
```

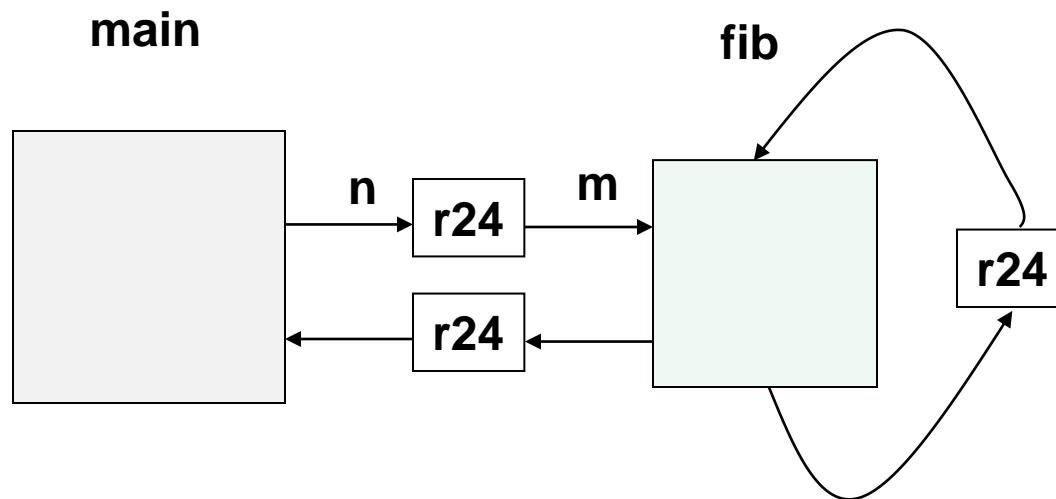
# Stack Frame for fib()

- Assembly program
  - Assume an integer takes one byte



# Parameter Passing

- Assume an integer takes one byte



# Example 2 (cont.)

- Assembly program

```
.include "m2560def.inc"
.cseg
    rjmp main
n:    .db 12 ; n is given in program memory

main:
    ldi ZL, low(n <<1) ; Let Z point to n
    ldi ZH, high(n <<1)
    lpm r24, Z ; Pass n via r24
    rcall fib ; Call fib(n)

halt:
    rjmp halt
```

# Example 2 (cont.)

- Assembly program

```
; fib(m)
fib:
    push r16          ; Prologue
    push YL           ; Save r16 on the stack
    push YH           ; Save Y on the stack
    in YL, SPL        ; Y←SP
    in YH, SPH
    sbiw Y, 1         ; Let Y point to the top of the stack frame
    out SPH, YH       ; Update SP so that it points to
    out SPL, YL       ; the new stack top
    std Y+1, r24      ; get the parameter

    cpi r24, 2         ; Check whether m is larger than 1
    brsh L2            ; If m!=0 or 1
    ldi r24, 1         ; m==0 or 1, return 1
    rjmp L1            ; Jump to the epilogue
```

# Example 2 (cont.)

- Assembly program

```
L2:    ldd r24, Y+1          ; m>=2, load the actual parameter m
        dec r24              ; Pass m-1 to the callee
        rcall fib             ; call fib(m-1)
        mov r16, r24           ; Store the return value in r16
        ldd r24, Y+1           ; Load the actual parameter m
        subi r24, 2             ; Pass m-2 to the callee
        rcall fib             ; call fib(m-2)
        add r24, r16            ; r24=fib(m-1)+fib(m-2)
```

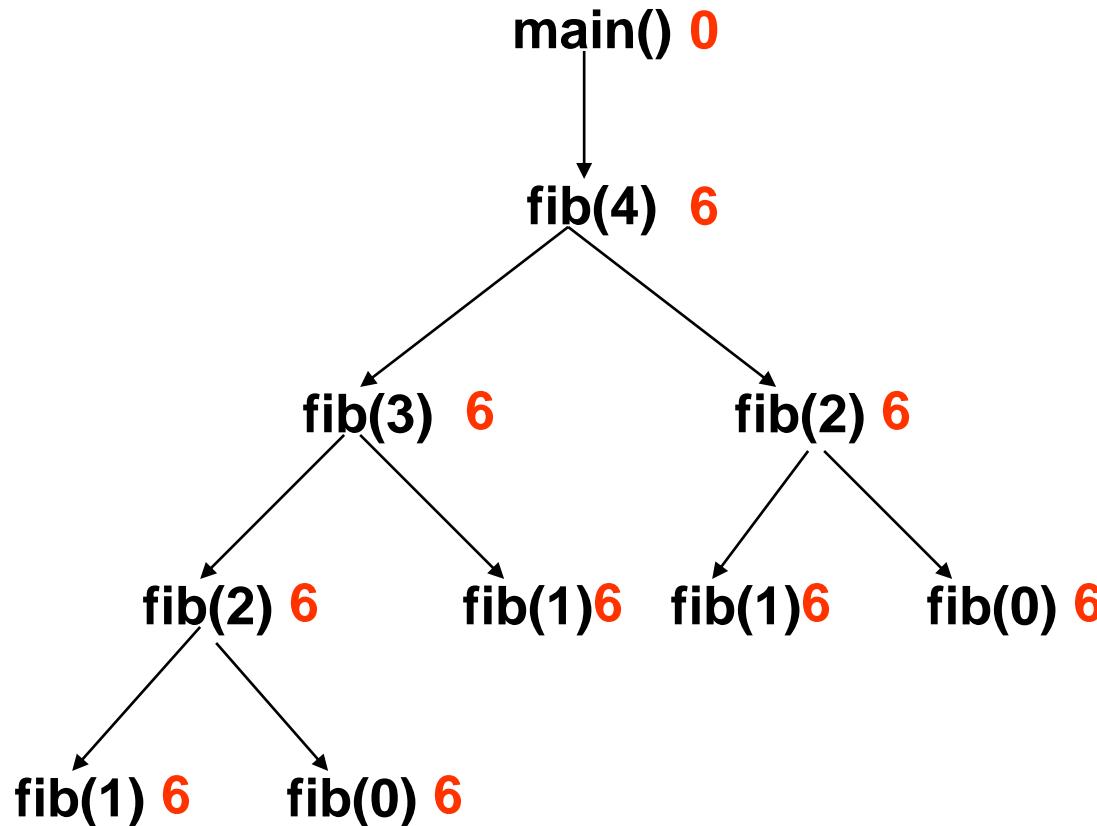
# Example 2 (cont.)

- Assembly program

L1:

```
; Epilogue
adiw Y, 1          ; De-allocate the stack frame for fib()
out SPH, YH       ; Restore SP
out SPL, YL
pop YH            ; Restore Y
pop YL
pop r16           ; Restore r16
ret
```

# Stack Size



The call tree for  $n=4$

The longest path: `fib(4)→fib(3)→fib(2)→fib(1)`

# Reading Material

- AVR ATmega2560 data sheet
  - Stack, stack pointer and stack operations

# Homework

1. Refer to the AVR Instruction Set manual, study the following instructions:
  - Arithmetic and logic instructions
    - lsl, rol
  - Data transfer instructions
    - pop, push
    - in, out
  - Program control
    - rcall
    - ret
  - Bit
    - clc
    - sec

# Homework

2. Read Introduction to AVR Microprocessor Development Board available on the Labs page

- You will be required to work with your lab group members to test the lab boards distributed to your group

# **Microprocessors & Interfacing**

*AVR Programming (III)*

Lecturer : Annie Guo

# Lecture Overview

- Memory access
- Assembly process
  - First pass
  - Second pass

# Memory Access Operations

- Access to data memory
  - Using instructions
    - ld, lds, st, sts
- Access to program memory
  - Using instructions
    - lpm
    - spm
      - Not covered in this course
  - Most of the time, that we access the program memory is to load data

# Load Program Memory Instruction

- Syntax:  $\text{Ipm Rd, Z}$
- Operands:  $Rd \in \{r0, r1, \dots, r31\}$
- Operation:  $Rd \leftarrow (Z)$
- Words: 1
- Cycles: 3

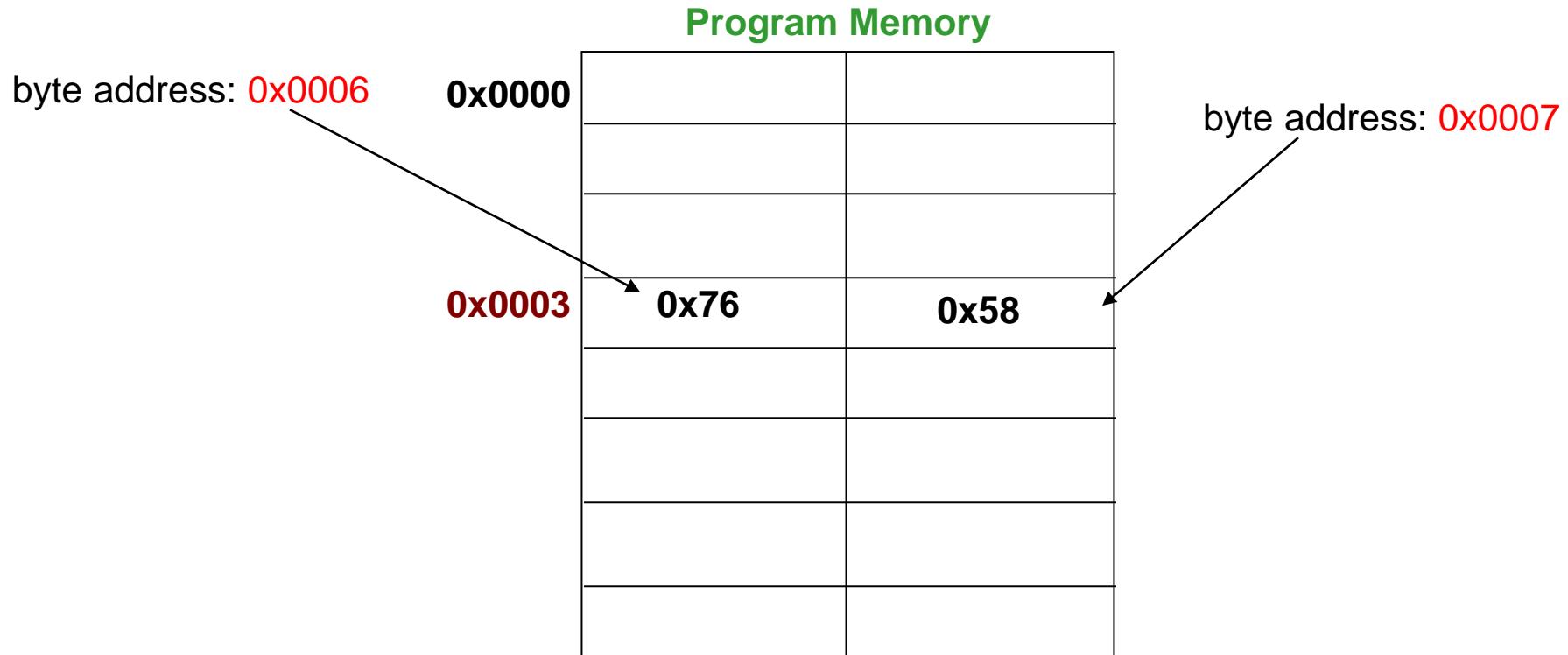
# Load Data From Program Memory

- The address label in the program memory is a ***word address***.
- To access constant data in the program memory with instruction *lpm*, ***byte address*** should be used.
- Address register, Z, is used to point to a byte in the program memory.

0x0000	'C'	'O'
0x0001	'M'	'P'
0x0002	'9'	'0'
0x0003	'3'	'2'
0x0004	0	0
0x0005	0x48	90320x23

# Byte Address vs Word Address

- First-byte-address (in a word) =  $2 * \text{word-address}$
  - Second-byte-address (in a word) =  $2 * \text{word-address} + 1$



# Example

```
.include "m2560def.inc" ; include definition for Z

ldi ZH, high(Table_1<<1) ; initialize Z
ldi ZL, low(Table_1<<1)

lpm r16, Z ; load constant from the program
             ; memory pointed to by Z (r31:r30)
            .
            .

Table_1:
.dw 0x5876 ; 0x76 is the value when ZLSB = 0
             ; 0x58 is the value when ZLSB = 1
```

# Complete Example 1

- Copy data from Program memory to Data memory

# Complete Example 1 (cont.)

- C description

```
struct STUDENT_RECORD
{
    int student_ID;
    char name[20];
    char WAM;
};

typedef struct STUDENT_RECORD student;

student s1 = {123456, "John Smith", 75};
```

# Complete Example 1 (cont.)

- Assembly translation

```
.include "m2560def.inc"

.set student_ID=0
.set name = student_ID+4
.set WAM = name + 20
.set STUDENT_RECORD_SIZE = WAM + 1

.cseg
start: ldi zh, high(s1_value<<1)          ; pointer to student record
       ldi zl, low(s1_value<<1)           ; value in the program memory

               ldi yh, high(s1)              ; pointer to student record holder
               ldi yl, low(s1)               ; in the data memory

clr r16
```

# Complete Example 1 (cont.)

- Assembly translation (cont.)

load:

```
cpi r16, STUDENT_RECORD_SIZE  
brge end  
lpm r10, z+  
st y+, r10  
inc r16  
rjmp load
```

end:

```
rjmp end
```

s1\_value:

```
.dw    LWRD(123456)  
.dw    HWRD(123456)  
.db    "John Smith      ", 0      ;take 20 bytes  
.db    75
```

.dseg

.org 0x200

s1: .byte STUDENT\_RECORD\_SIZE

# Complete Example 2

- Convert lowercase to uppercase for a string (for example, “hello”)
  - The string is stored in the program memory
  - The resulting string after conversion is stored in the data memory.
  - In ASCII, uppercase letter + 32 = lowercase letter
    - e.g. ‘A’+32=‘a’

# Complete Example 2 (cont.)

- Assembly program

```
.include "m2560def.inc"
.equ size = 6                                ; string length
.def counter = r17
.dseg
.org 0x200                                     ; set the starting address
                                                ; of data segment to 0x200
ucase_string: .byte size

.cseg
    ldi zl, low(lcase_string<<1)      ; get the low byte for
                                            ; the address of "h"
    ldi zh, high(lcase_string<<1)      ; get the high byte for
                                            ; the address of "h"
    ldi yh, high(ucase_string)
    ldi yl, low(ucase_string)
    clr counter                         ; initialize counter
```

# Complete Example 2 (cont.)

- Assembly program (cont.)

```
main:
```

```
    lpm r20, z+    ; load a letter from flash memory
    subi r20, 32    ; convert it to the uppercase letter
    st y+,r20      ; store the uppercase letter in SRAM
    inc counter
    cpi counter, size-1
    brlt main
    lpm r20, z      ; copy null
    st y, r20
end:
    rjmp end
```

```
Icase_string: .db "hello", 0
```

# Assembly

- Assembly programs need to be converted to machine code before execution
  - This translation/conversion from assembly program to machine code is called **assembly** and is done by the **assembler**
- There are two general steps in the assembly process:
  - Pass one
  - Pass two

# Two Passes in Assembly

- Pass One
  - Do lexical and syntax analysis: checking for syntax errors
  - Expand macros
  - Record all the symbols (labels etc) in a symbol table
- Pass Two
  - Use the symbol table to substitute values for symbols and evaluate functions.
  - Assemble each instruction
    - i.e. generate machine code

# Example 1

Assembly program

```
.equ    bound = 5  
  
loop:  
    clr r16  
    cpi r16, bound  
    brlo end  
    inc r16  
    rjmp loop  
  
end:  
    rjmp end
```

Symbol table

Symbol	Value
bound	5
loop	1
end	5

# Example 1 (cont.)

## Code generation

<u>Address</u>	<u>Code</u>	<u>Assembly statement</u>	
00000000:	2700	clr	r16
00000001:	3005	cpi	r16,0x05
00000002:	F010	brlo	PC+0x02
00000003:	9503	inc	r16
00000004:	CFFC	rjmp	PC-0x0004
00000005:	CFFF	rjmp	PC-0x0001

# Example 2

- Treatment of labels in macro
  - Not directly associated with address, rather the offset.

```
.include "m2560def.inc"

.def    a = r16
.def    b = r17
.def    c = r18

.macro try
    cp @0, @1
    brne end1
    inc a
end1:
.endmacro

try a, b
try a, c

end:
    rjmp end
```

```
--- C:\Users\mrig\Documents\teaching\9032\AVR\lecture_e
try a, b
00000000 CP R16,R17      Compare
00000001 BRNE PC+0x02    Branch if not equal
00000002 INC R16         Increment
try a, c
00000003 CP R16,R18      Compare
00000004 BRNE PC+0x02    Branch if not equal
00000005 INC R16         Increment
    rjmp end
00000006 RJMP PC-0x0000  Relative jump
--- No source file ---
00000007 NOP             Undefined
```

```
.include "m2560def.inc"

.def    a = r16
.def    b = r17
.def    c = r18

.macro try
    cp @0, @1
    brne end1
    inc a
end1:
.endmacro

try a, b
try a, c

end:
    rjmp end
```

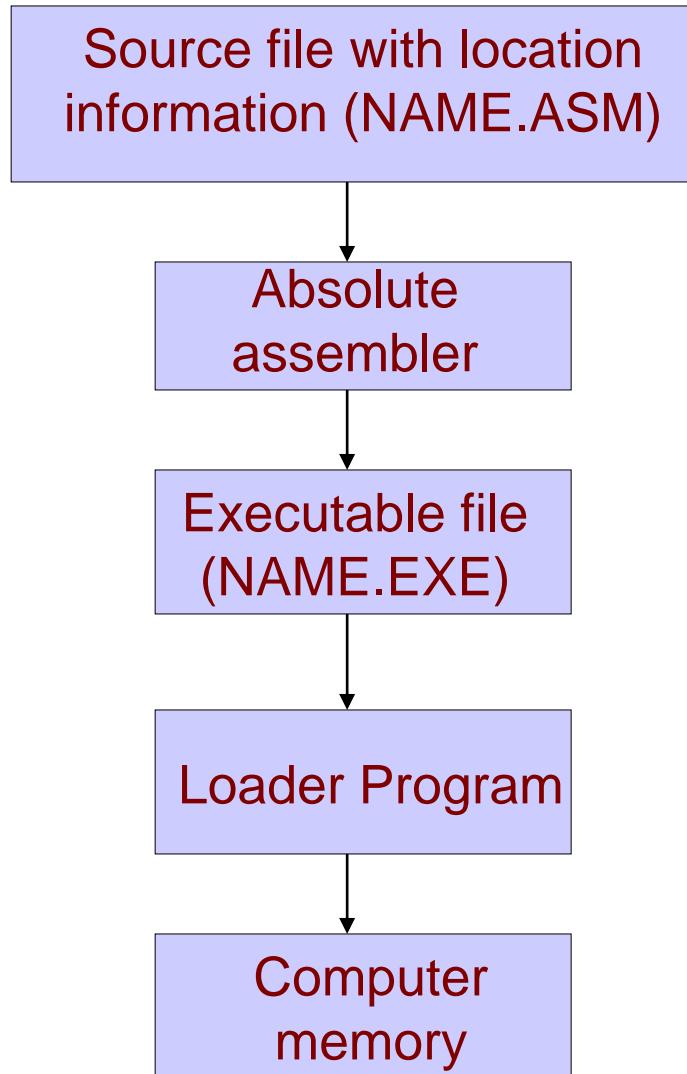
```
--- C:\Users\mrig\Documents\teaching\9032\AVR\lecture_e
try a, b
00000000 CP R16,R17      Compare
00000001 BRNE PC+0x05    Branch if not equal
00000002 INC R16         Increment
try a, c
00000003 CP R16,R18      Compare
00000004 BRNE PC+0x02    Branch if not equal
00000005 INC R16         Increment
    rjmp end
00000006 RJMP PC-0x0000  Relative jump
--- No source file ---
00000007 NOP             Undefined
00000008 NOP             Undefined
```

# Absolute Assembly

- A type of assembly process.
  - Can only be used for the source file that contains all the source code of the program
- Programmers use .org to tell the assembler the starting address of a segment (data segment or code segment)
- Whenever any change is made in the source program, all code must be assembled.
- A loader transfers an **executable file** (machine code) to the target system.

# Absolute Assembly

## - workflow

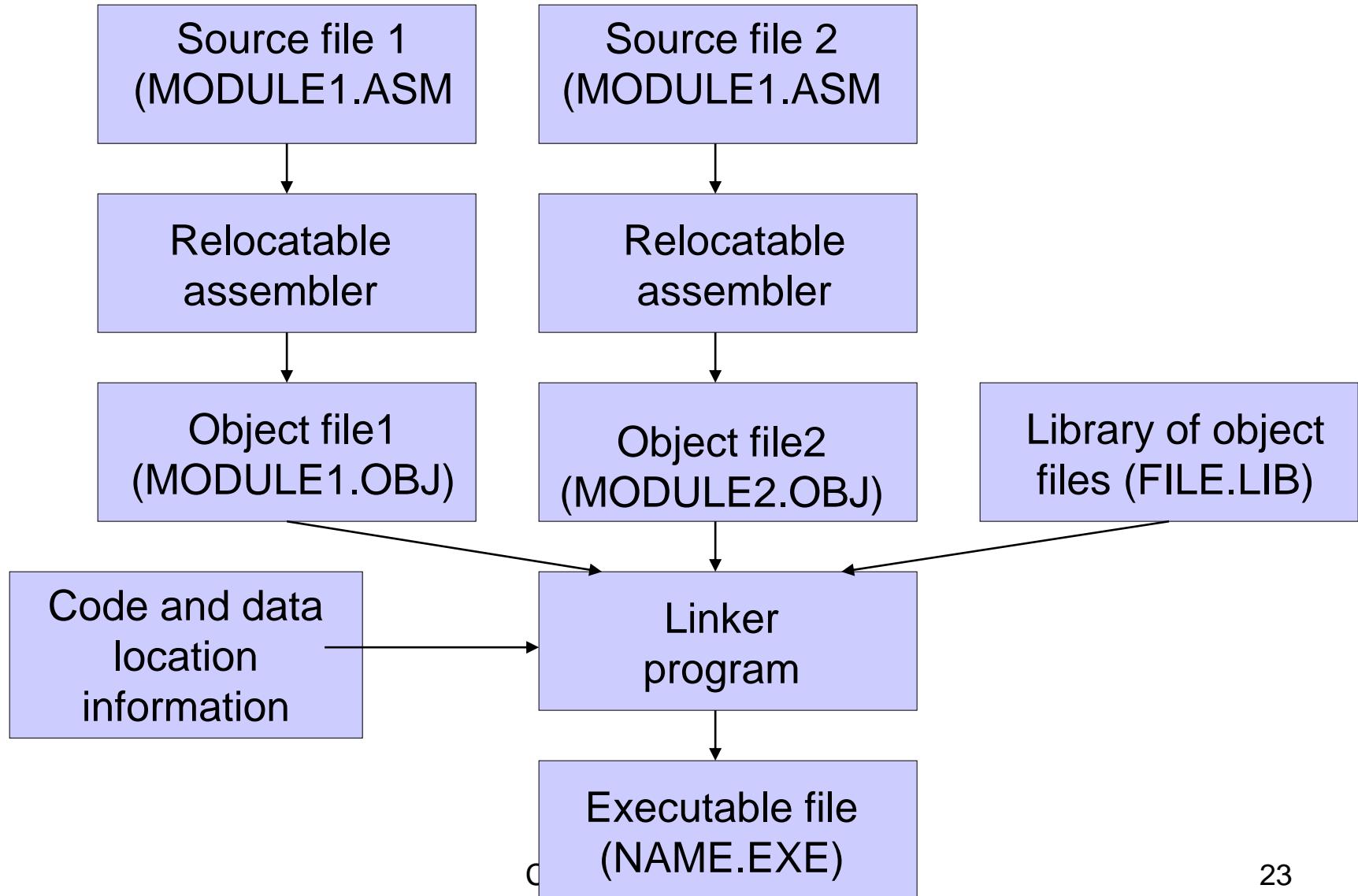


# Relocatable Assembly\*

- Another type of assembly process.
- Each source file can be assembled separately
- Each file is assembled into **an object file** where some addresses may not be resolved
- A linker program is needed to resolve all unresolved addresses and make all object files into a single executable file

# Relocatable Assembly\*

## - workflow



# Homework

1. Write a macro that can perform either logical shift left or arithmetic shift right on a register by a given number of bits.
2. Write a macro to check whether a register holds a valid hexadecimal digit character.
3. Complete Quiz 2



# More on how labs are run

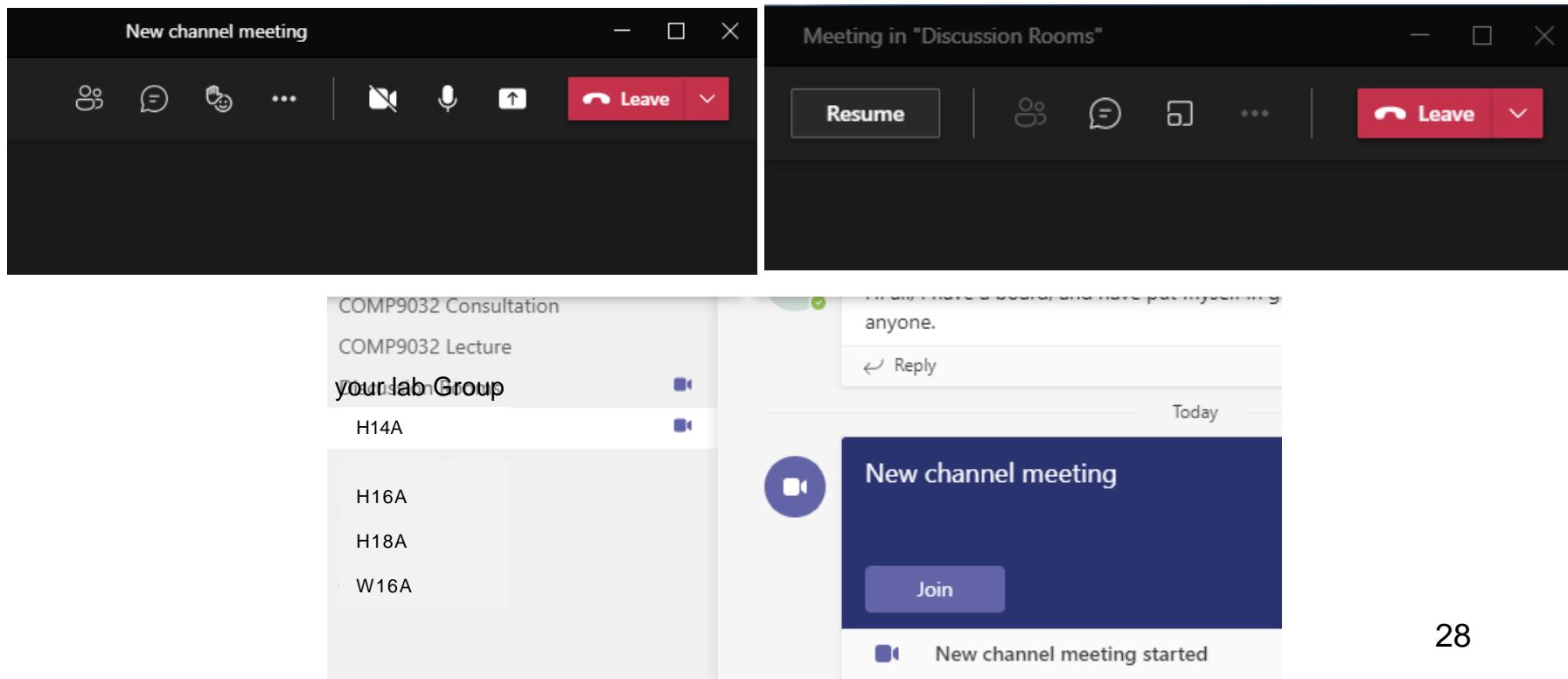
- For f2f classes
  - If there are no scheduled assessments
    - For individual tasks, work on your lab tasks, or
    - For group lab tasks, work with your group members
  - For any questions, you can talk to your tutor in the lab.
  - Safety procedure should be followed
    - See the guide on the Labs page of the course website

# More on how labs are run (cont.)

- For the online class
  - There are two types of meetings
    - General meeting in your lab class channel
      - Everyone can join in
    - Private group meetings in your private group channel
      - Used for group discussions
      - Your tutor can join in
    - For any questions, you can talk to your tutor either in the general meeting or your private group meeting
  - Each lab session starts in the general meeting and then individual groups go to their own meetings

# More on how labs are run (cont.)

- For the online class (cont.)
  - You can move between meetings
    - To seek help from your tutor.
    - To get your work assessed by your tutor.



# How your lab work is assessed

- For some lab exercises
  - Assessed individually
    - You need to develop your own design and present your work to the lab tutor
- For some lab exercises
  - Assessed in group
    - The whole group collaboratively develop one solution and present the group work to the lab tutor
    - Your participation and contribution to the group work will be considered for your marks from the group assessment

# How your lab work is assessed (cont.)

- For each assessment, you will be randomly assigned to a time slot.
  - For the online class, your tutor will invite you or your group to the general meeting for the assessment when it is your turn.
  - For the f2f classes, your tutor will come to your group

Time	Tutor 1	Tutor 2
16:20	student 4	student 9
16:30	student 8	student 10
16:40	student 12	student 20
16:50	student 1	student 14
17:00	student 2	student 18
17:10	student 3	student 13
17:20	student 5	student 15
17:30	student 11	student 17
17:40	student 6	student 19
17:50	student 7	student 16

– e.g. W16A

for individual assessment

Time	Tutor 1	Tutor 2
16:00	G3	G1
16:30	G2	G4

for group assessment

# About Labs of Week 2

- Due to the public holiday on Thursday this week, labs in Week 2 were cancelled
- Lab 1 spec is available on the course website
  - The first task of Lab 1 will be assessed in Week 3

# Lab board distribution

- The lab boards will be distributed in your lab classes in Week 3
  - Currently two boards per group
    - May increase to one board per two persons once the enrolment is stabilized.
  - Select two members who can be in charge of the board for your group
  - For the online class, contact your tutor for a pick up time at the campus
- Information about lab board will be available on the course website in Week 3.

# **Microprocessors & Interfacing**

*AVR Programming (II)*

Lecturer : Annie Guo

# Lecture Overview

- Assembly program structure
  - Assembler directive
  - Assembler expression
  - Macro

# Assembly Program Structure

- An assembly program basically consists of
  - Assembler directives
    - E.g. **.def temp = r15**
  - Executable instructions
    - E.g. **add r1, r2**
- A line in an assembly program takes one of the following forms :
  - [label:] directive [operands] [comment]
  - [label:] instruction [operands] [comment]
  - Comment
  - Empty line

Note: [ ] indicates optional

```
;This program performs two-byte subtraction: d = a-b;
```

```
.def ah=r10 ;a  
.def al=r11  
.def bh=r12 ;b  
.def bl=r13  
.def dh=r14 ;d  
.def dl=r15
```

```
        mov dh,ah  
        mov dl,al  
        sub dl,bl  
        sbc dh,bh  
end:    rjmp end
```

# Assembly Program Structure (cont.)

- The label for an instruction or a data item in the memory is **associated with the memory address** of that instruction or that data item.
- All instructions are not case sensitive
  - “add” is same as “ADD”
  - “.def” is same as “.DEF”

# Comments

- A comment line has the following form:

`:[text]`

Items within the brackets are optional

- The text between the comment-delimiter(;) and the end of line (EOL) is ignored by the assembler.

# Assembly Directives

- Assembly directives are instructions to the assembler. They are used for a number of purposes:
  - For symbol definition
    - For readability and maintainability
    - All symbols used in a program will be replaced by the real values associated with the symbol during assembling
    - E.g. .def, .set
  - For program and data organization
    - E.g. .org, .cseg, .dseg
  - For data/variable memory allocation
    - E.g. .db
  - For others

# Typical AVR Assembler directives

Directive	Description
BYTE	Reserve byte to a variable
CSEG	Code Segment
DB	Define constant byte(s)
DEF	Define a symbolic name on a register
DEVICE	Define which device to assemble for
DSEG	Data Segment
DW	Define constant word(s)
ENDMACRO	End macro
EQU	Set a symbol equal to an expression
ESEG	EEPROM Segment
EXIT	Exit from file
INCLUDE	Read source from another file
LIST	Turn listfile generation on
LISTMAC	Turn macro expansion on
MACRO	Begin macro
NOLIST	Turn listfile generation off
ORG	Set program origin
SET	Set a symbol to an expression

NOTE: All directives must be preceded by **a period, ‘.’**

# Directives for Symbol Definition

## .def

- Define a symbol/name for a **register**

```
.def      symbol = register
```

- E.g.

```
.def t = r17
```

- Symbol *t* can be used for r17 anywhere in the program after the definition

## How to define address register X, Y, Z?

# Directives for Symbol Definitions (cont.)

## .equ

- Define a symbol/name for a **value**

```
.equ    symbol = expression
```

- Non-redefinable. Once set, the symbol cannot be later redefined to other value in the program
- E.g.
  - .equ length = 2
  - Symbol *length* with value 2 can be used anywhere in the program after the definition

# Directives for Symbol Definitions (cont.)

## .set

- Define a symbol/name for a **value**

```
.set      symbol = expression
```

- **Re-definable.** The symbol can be changed later to represent other value in the program.
- E.g.
  - .set input = 5
    - Symbol *input* with value 5 can be used anywhere in the program after this definition and before its redefinition.

# Program/Data Memory Organization

- AVR has three different memories
  - Data memory
  - Program memory (aka instruction or code memory)
  - EEPROM memory (not covered in this course)
- The memories are corresponding to memory segments to the assembler:
  - Data segment
  - Program segment (or Code segment)

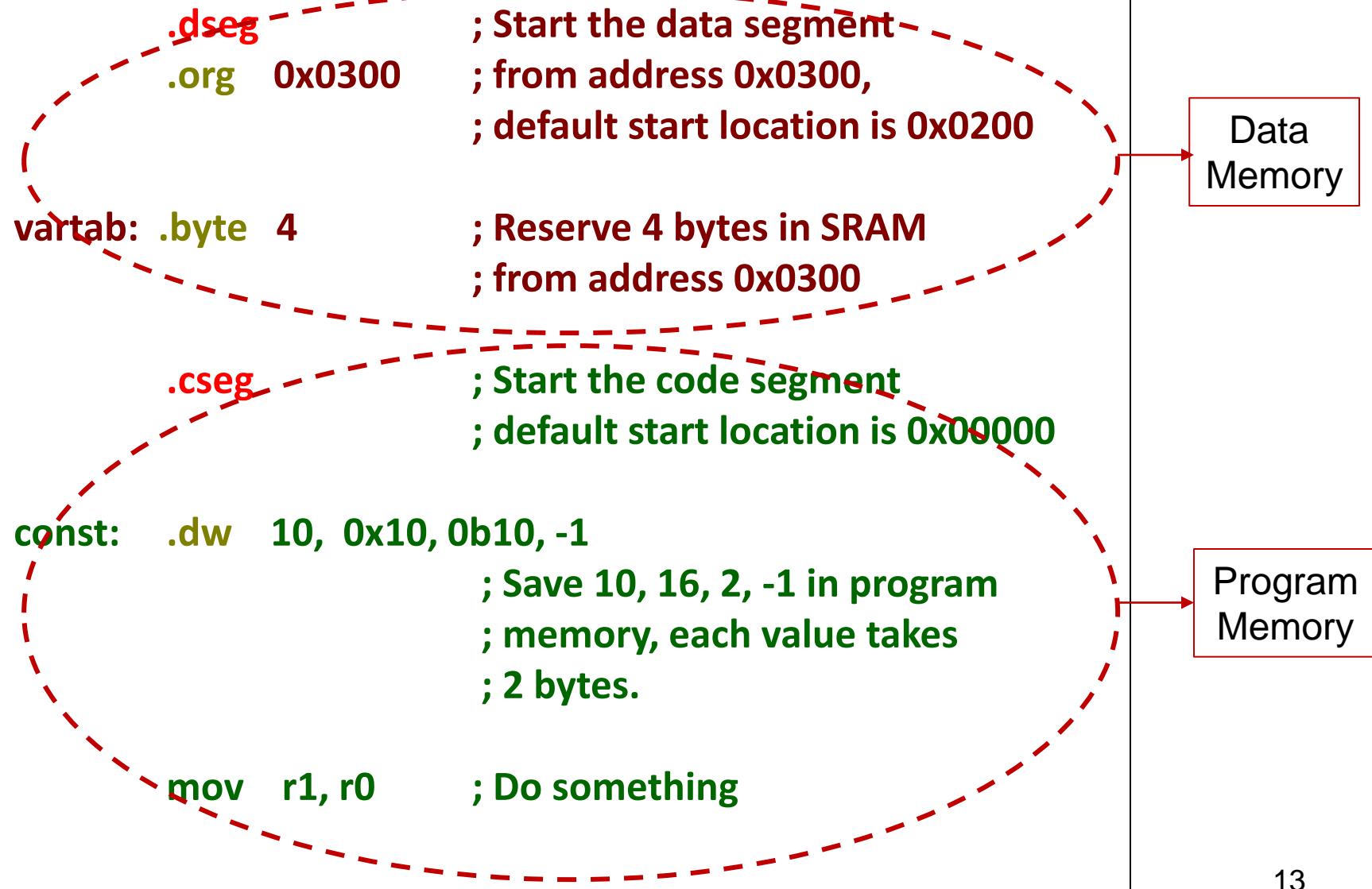


• Segment here is referred to as  
a memory space

# Program/Data Memory Organization Directives

- Memory segment directives specify which memory to use
  - **.dseg**
    - Data memory
  - **.cseg**
    - Code/Program memory
- The default segment is cseg
- The **.org** directive specifies the start address for the related code/data to be saved

# Example



# Data/Variable Memory Allocation Directives

- Specify the memory locations/sizes for
  - Constants
    - In program memory
  - Variables
    - In data memory
- All directives must start with a label so that the related data/variables can be accessed later.

# Directives for Constants

- Store data in **program memory**
  - **.db**
    - Store **byte** constants in program memory
    - **label: .db expr1, expr2, ...**
    - **expr\*** is a byte constant
  - **.dw**
    - Store **word** (16-bit) constants in program memory
    - **little endian** rule is used
- The labels here are associated with a **word address**

# Directives for Variables

- Reserve bytes in **data memory**
  - **.byte**
    - Reserve a number of bytes for a variable

```
Label: .byte  expr
```

- *expr* is the number of bytes to be reserved.
- The label is associated with a **byte address**

# Other Directives

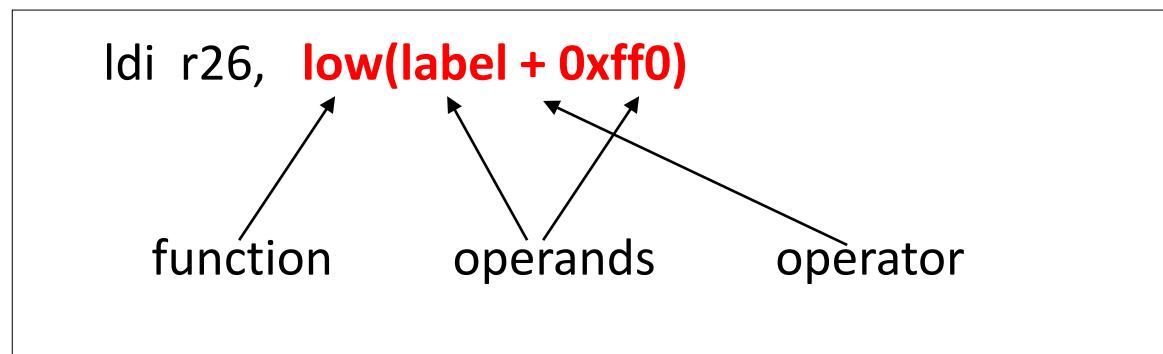
- Include a file
  - **.include** “m2560def.inc”
- Stop processing assembly file
  - **.exit**
- Define macro
  - **.macro**
  - **.endmacro**
  - Will be discussed in detail later

# Assembler Expressions

- In the assembly program, you can use expressions for values.
- **During assembling**, the assembler evaluates each expression and replaces the expression with the calculated value.

# Assembler Expressions (cont.)

- The expressions are in a form similar to normal math expressions
  - Consisting of operands, operators and functions.  
All expressions can be of a value up to 32 bits.
- Example



# Operands in Assembler Expression

- Operands can be any of the following:
  - User defined labels
    - associated with memory addresses
  - User defined variables
    - defined by the ‘set’ directive
  - User defined constants
    - defined by the ‘equ’ directive
  - Integer constants
    - can be in several formats, including
      - decimal (default): e.g. 10, 255
      - hexadecimal (two notations): e.g. 0x0a, \$0a, 0xff, \$ff
      - binary: e.g. 0b00001010, 0b11111111
  - PC
    - Program Counter value.

# Operators in Assembler Expression

Same  
meanings  
as in C

Symbol	Description
!	Logical Not
~	Bitwise Not
-	Unary Minus
*	Multiplication
/	Division
+	Addition
-	Subtraction
<<	Shift left
>>	Shift right
<	Less than
<=	Less than or equal
>	Greater than
>=	Greater than or equal
==	Equal
!=	Not equal
&	Bitwise And
^	Bitwise Xor
	Bitwise Or
&&	Logical And
	Logical Or

# Functions in Assembler Expression

- **LOW(expression)**
  - Returns the low byte of an expression
- **HIGH(expression)**
  - Returns the second (low) byte of an expression
- **BYTE2(expression)**
  - The same function as HIGH
- **BYTE3(expression)**
  - Returns the third byte of an expression
- **BYTE4(expression)**
  - Returns the fourth byte of an expression
- **LWRD(expression)**
  - Returns low word (bits 0-15) of an expression
- **HWRD(expression):**
  - Returns bits 16-31 of an expression
- **PAGE(expression):**
  - Returns bits 16-21 of an expression
- **EXP2(expression):**
  - Returns 2 to the power of expression
- **LOG2(expression):**
  - Returns the integer part of log2(expression)

# Examples of Assembler Expression

; Example 1:

```
ldi r17, 1<<5 ; load r17 with 1 left-shifted by 5 bits
```

# Examples of Assembler Expression

```
; Example 2: compare r21:r20 with 3167
```

```
ldi r16, high(3167)
cpi r20, low(3167)
cpc r21, r16
brlt case1
...
case1: inc r10
```

# Data/Variables Implementation

- With the assembler directives, you can implement/translate data/variables into machine level descriptions
  - See some examples in the next a few slides.

# Remarks

- Data have scope and duration in the program
- Data have types and structures
- Those features determine where and how to store data in memory.
- Constants are usually stored in the non-volatile memory and variables are allocated in SRAM memory.
- In this lecture, we will only take a look at how to implement basic data type.
  - Implementation of advanced data structures/variables will be covered later.

# Example 1

- Translate the following variables in a C program. Assume each integer takes four bytes.

```
int a;  
unsigned int b;  
char c;  
char* d;
```

# Example 1: Solution

- Translate the following variables. Assume each integer takes four bytes.

```
.dseg      ; in data memory

.org 0x200    ; start from address 0x200

a: .byte 4    ; 4 byte integer
b: .byte 4    ; 4 byte unsigned integer
c: .byte 1    ; 1 character
d: .byte 2    ; address, pointing to the string
```

- All variables are allocated in data memory (SRAM)
- Labels are given the same names as the variables for convenience and readability.

- FLASH – Code memory
- ASCII is used for character

# Example 2

- Translate the following C constants and variables.

C code:

```
int a;  
const char b[ ] = "COMP9032";  
const int c = 9032;
```

Assembly  
code:

```
.dseg  
a: .byte 4  
  
.cseg  
;b: .db 'C', 'O', 'M', 'P', '9', '0', '3', '2', 0  
b: .db "COMP9032", 0  
c: .dw 9032
```

- All variables are in SRAM and constants are in FLASH

# Example 2 (cont.)

- Program memory mapping
  - In the program memory, data are packed in words.  
If only a single byte left, that byte is stored in the first (left) byte and the second (right) byte is filled with 0, as highlighted in the example.

	low addr	high addr	Hex values
0x0000	'C'	'O'	43 4F
0x0001	'M'	'P'	4D 50
0x0002	'9'	'0'	39 30
0x0003	'3'	'2'	33 32
0x0004	0	0	0 0
0x0005	0x48	0x23	48 23

COMP9032 Week2

# Example 3

- Translate variables with structured data type

```
struct STUDENT_RECORD
{
    int student_ID;
    char name[20];
    char WAM;
};

typedef struct STUDENT_RECORD student;

student s1;
student s2;
```

# Example 3 : Solution

- Translate variables with structured data type

```
.set    student_ID=0
.set    name = student_ID+4
.set    WAM = name + 20
.set    STUDENT_RECORD_SIZE = WAM + 1

.dseg
s1:    .BYTE   STUDENT_RECORD_SIZE
s2:    .BYTE   STUDENT_RECORD_SIZE
```

# Example 4

- Translate variables with structured data type
  - with initialization

```
struct STUDENT_RECORD
{
    int student_ID;
    char name[20];
    char WAM;
};

typedef struct STUDENT_RECORD student;

struct student s1 = {123456, "John Smith", 75};
struct student s2;
```

# Example 4: Solution

- Translate variables with structured data type

```
.set    student_ID=0
.set    name = student_ID+4
.set    WAM = name + 20
.set    STUDENT_RECORD_SIZE = WAM + 1
```

```
.cseg
```

```
s1_value: .dw LWRD(123456)
          .dw HWRD(123456)
          .db "John Smith      ", 0
          .db 75
```

20 bytes

```
.dseg
```

```
s1:     .byte STUDENT_RECORD_SIZE
s2:     .byte STUDENT_RECORD_SIZE
```

```
; copy the data from instruction memory to s1
```

```
...
```

# Remarks

- The constant values for initialization are usually stored in the program memory in order to keep the values when power is off.
- The variables will be populated with the initial values when the program is started.

# Macro

- Sometimes, a sequence of instructions in an assembly program need to be repeated several times
- Macros help programmers to write code efficiently and nicely
  - Write/define a section of code once and reuse it
    - Neat representation
  - Like an inline function in C
    - When assembled, the macro is expanded at the place it is used

# Directives for Macro

## .macro

- Tells the assembler that this is the start of a macro
- Takes the macro name and (implicitly) parameters
  - Up to 10 parameters
    - Which are referenced by @0, ...@9 in the macro definition body

## .endmacro

- Specifies the end of a macro definition.

# Macro (cont.)

- Macro definition structure:

```
.macro macro_name  
    ; macro body  
.endmacro
```

- Usage

```
macro_name [para0, para1, ...,para9]
```

@0, @1 ...@9

# Example 1

- Swapping two memory data

```
.macro swap2
```

```
    lds r2, @0      ; load data from provided
```

```
    lds r3, @1      ; two locations.
```

```
    sts @1, r2      ; store the data from one
```

```
    sts @0, r3      ; location to other location
```

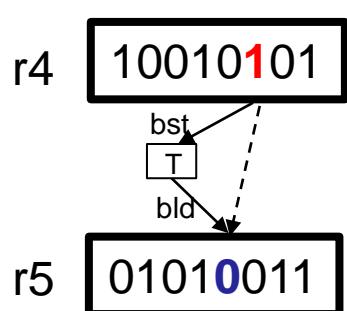
```
.endmacro
```

```
swap2 a, b          ; a is @0, b is @1.
```

```
swap2 c, d          ; c is @0, d is @1.
```

# Example 2

- Register bit copy
  - copy a bit from one register to a bit of another register



```
; Copy bit @1 of register @0
; to bit @3 of register @2
```

```
.macro bitcopy
    bst @0, @1
    bld @2, @3
.endmacro
```

```
bitcopy r4, 2, r5, 3
bitcopy r5, 4, r7, 6
```

# Example 3

- Code execution termination

```
; end of code execution

.macro exit
end:
    rjmp end
.endmacro
```

# Reading Material

- Cady “Microcontrollers and Microprocessors”, Chapter 6 for assembly programming style, or
- User’s guide to AVR assembler
  - This guide is also a part of the on-line documents accompanied with Atmel Studio. Click help in the Studio to explore.

# Homework

1. Refer to the AVR Instruction Set manual, study the following instructions:
  - Arithmetic and logic instructions
    - clr
    - inc, dec
  - Data transfer instructions
    - movw
    - sts, lds
    - lpm
    - bst, bld
  - Program control
    - jmp
    - sbrs, sbrc

# **Microprocessors & Interfacing**

*AVR ISA &  
AVR Programming (I)*

Lecturer : Annie Guo

# Lecture Overview

- AVR ISA and Instructions
  - An overview of our target machine
- AVR Programming (I)
  - Implementation of basic programming structures

# Atmel AVR (8-bit)

- RISC architecture
  - Most instructions have 16-bit fixed length
  - Most instructions take 1 clock cycle to execute
- Load-store memory access architecture
  - All arithmetic and logic (AL) calculations are performed on registers
- Internal program memory and data memory
- Wide variety of on-chip peripherals (digital I/O, ADC, EEPROM, UART, pulse width modulator (PWM) ...).

# AVR Registers

- General purpose registers
  - 32 8-bit registers, R0 ~ R31 or r0 ~ r31
  - Can be further divided into two groups
    - First half group (R0 ~ R15) and second half group (R16 ~ R31)
    - Some instructions work only on the second half group R16~R31
      - Due to the limitation of instruction encoding bits
        - » Will be covered later
      - E.g. *ldi rd, #number* ;rd ∈ R16~R31

# AVR Registers (cont.)

- General-purpose registers
  - The following register pairs can work as **address registers** (or address pointers)
    - X, R27:R26
    - Y, R29:R28
    - Z, R31:R30
  - The following registers can be used for specific purposes
    - R1:R0 stores the result of a multiplication instruction
    - R0 stores the data loaded from the program memory

# AVR Registers (cont.)

- I/O registers
  - 64+ 8-bit registers
    - Their names are defined in the *m2560def.inc* file
  - Used in input/output operations
    - Mainly for storing data/addresses and control signal bits
  - Will be covered in detail later
- Status register (SREG)
  - A special I/O register

# SREG

- The Status REGister (SREG) contains information about the result of the most recently executed AL instruction. This information can be used for altering program execution flow in order to perform conditional operations.
- SREG is updated by hardware after an AL operation.
  - Some instructions such as load do not affect SREG.
- SREG is not automatically saved when entering an interrupt routine and restored when returning from an interrupt. This must be handled by software.
  - Using in/out instruction to store/restore SREG
  - To be covered later

# SREG (cont.)

I	T	H	S	V	N	Z	C
Bit 7	6	5	4	3	2	1	0

- Bit 0 – C: Carry Flag
  - Its meaning depends on the operation.
    - For addition  $x+y$ , it is the carry from the most significant bit.
    - For subtraction  $x-y$ , where  $x$  and  $y$  are unsigned integers, it indicates whether  $x < y$  or not. If  $x < y$ ,  $C=1$ ; otherwise,  $C=0$ .

# SREG (cont.)

	I	T	H	S	V	N	Z	C
Bit	7	6	5	4	3	2	1	0

- Bit 1 – Z: Zero Flag
  - Z indicates **a zero result** from an arithmetic or logic operation. 1: zero. 0: non-zero.
- Bit 2 – N: Negative Flag
  - N is the most significant bit of the result.

# SREG (cont.)

	I	T	H	S	V	N	Z	C
Bit	7	6	5	4	3	2	1	0

- Bit 3 – V: Overflow Flag
  - For two's complement arithmetic operations
- Bit 4 – S: Sign Flag
  - Exclusive OR of the Negative Flag N and the Two's Complement Overflow Flag V ( $S = N \oplus V$ ).

N	V	$N \oplus V$
0	0	0
0	1	1
1	0	1
1	1	0

# SREG (cont.)\*

I	T	H	S	V	N	Z	C
Bit 7	6	5	4	3	2	1	0

- Bit 5 – H: Half Carry Flag
  - The Half Carry Flag H indicates a Half Carry (carry from bit 3) in some arithmetic operations.
  - Half Carry is useful in BCD arithmetic.
  - Not covered in this course

• BCD: Binary Coded Decimal

# SREG (cont.)

	I	T	H	S	V	N	Z	C
Bit	7	6	5	4	3	2	1	0

- Bit 6 – T: Temporary Storage for Bit Copy
  - Used for transferring a bit from one register to another register
    - The Bit Copy instructions BLD (Bit LoaD) and BST (Bit STore) use the T-bit as source or destination for the transferred bit.

# SREG (cont.)

	I	T	H	S	V	N	Z	C
Bit	7	6	5	4	3	2	1	0

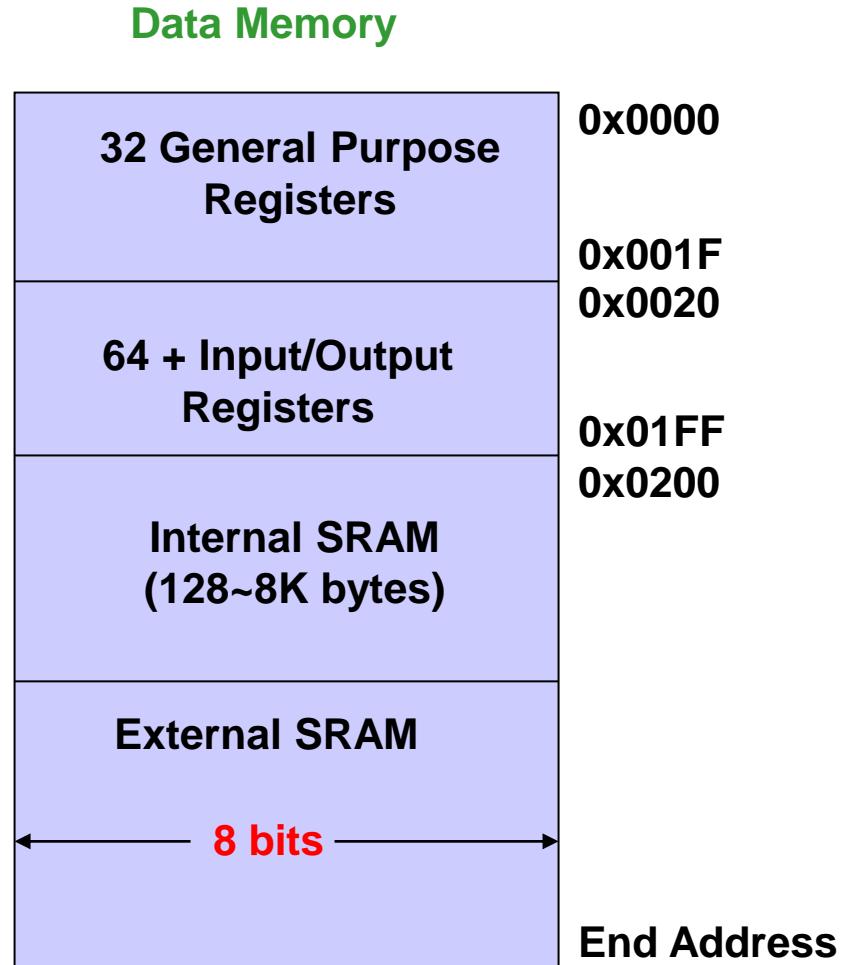
- Bit 7 – I: Global Interrupt Enable
  - Used to enable and disable interrupts.
  - 1: enable. 0: disable.
  - The I-bit is cleared by hardware after an interrupt has occurred, and is set by the RETI instruction to enable subsequent interrupts.
    - Will be covered later

# AVR Address Spaces

- Three address spaces
  - Data memory
    - Storing data to be processed
  - Program memory
    - Storing program code and constants
  - EEPROM memory
    - Large permanent data storage
      - Not covered in this course

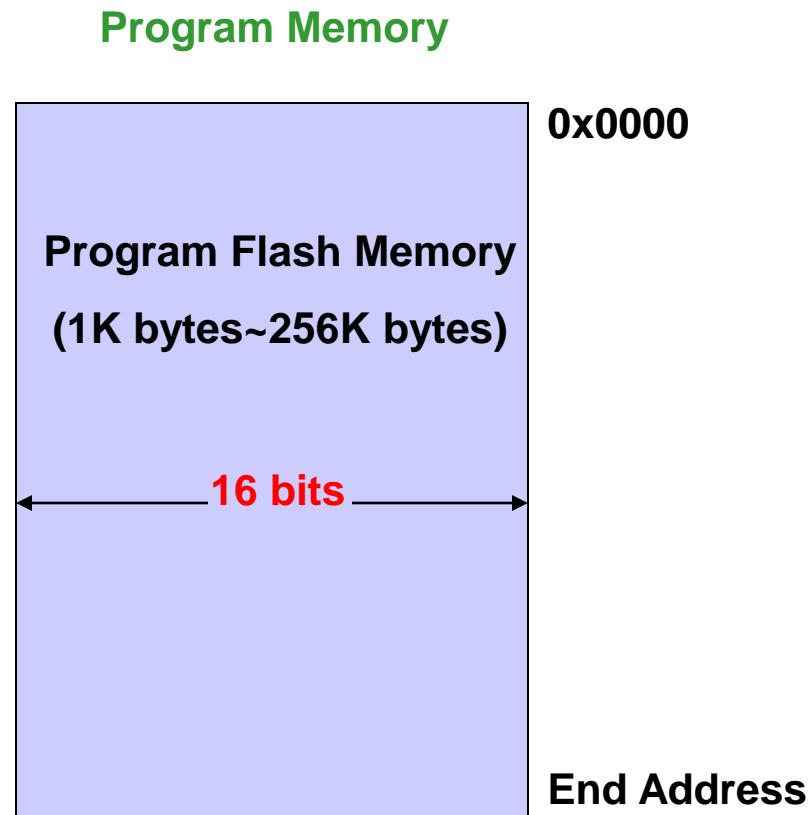
# Data Memory Space

- The space covers
  - Register file
    - i.e. registers in the register file also have memory addresses
  - I/O registers
    - I/O registers have two versions of addresses
      - I/O addresses
      - **Memory addresses**
  - SRAM data memory
    - The highest data memory location is defined as **RAMEND**



# Program Memory Space

- The space covers
  - 16-bit flash memory
    - Non-volatile
      - Instructions and data are retained when power off
    - Mainly for read only
  - Can be accessed with special instructions
    - LPM
    - SPM
      - Not covered



# AVR Instruction Format

- For AVR, almost all instructions are 16 bits long
  - For example
    - *add Rd, Rr*
    - *sub Rd, Rr*
    - *mul Rd, Rr*
    - *brge k*
- Some instructions are 32 bits long
  - For example
    - *lds Rd, k ( 0 \leq k \leq 65535 )*
      - loads 1 byte from data memory to a register.
- Some examples are given in the next slides

i.e. 16 bits  
or 64K

# Instruction Examples (1)

## - 16 bits long

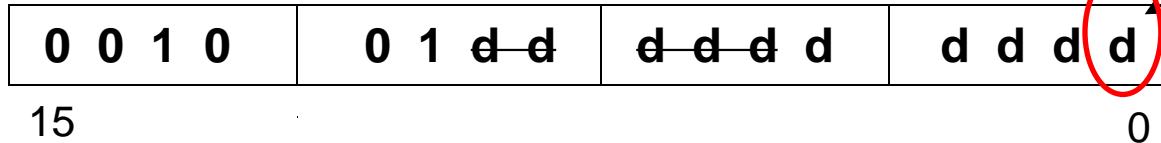
- Instruction for “clear register”

Syntax: *clr Rd*

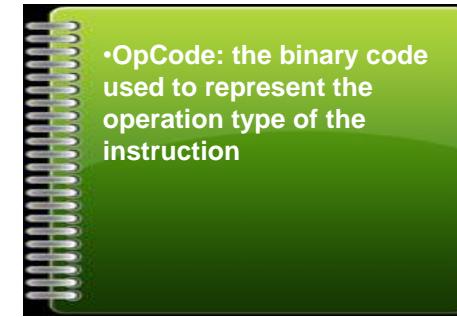
Operand:  $0 \leq d \leq 31$

Operation:  $Rd \leftarrow 0$

- Instruction format



- OpCode uses 6 bits (bit 10 to bit 15).
- The operand uses the remaining 10 bits (only 5 bits, bit 0 to bit 4, are needed).
- Execution time  
1 clock cycle



# Instruction Examples (2)

## - 32 bits long

- Instruction for “unconditional branch”

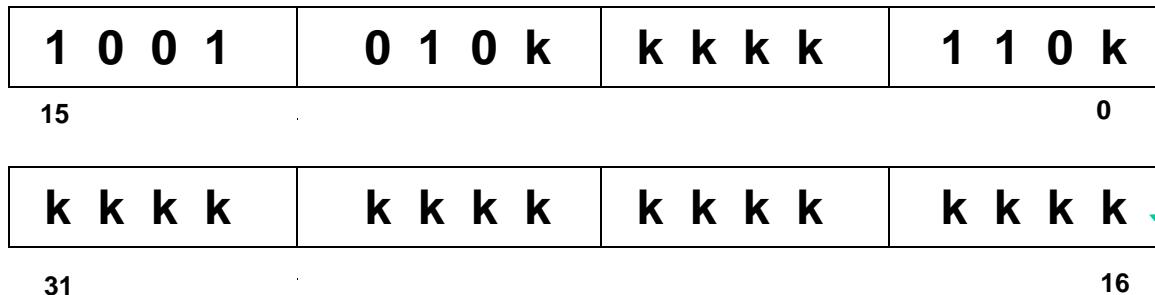
Syntax:  $jmp\ k$

Operand:  $0 \leq k < 4M$

Operation:  $PC \leftarrow k$

How many bits does it indicate?  
Based on binary number system  
 $M \rightarrow 2^{20}$

- Instruction format



- Execution time  
3 clock cycles

bits used for value  $k$

# Instruction Examples (3)

## - with variable exec. time

- Instruction for “conditional branch”

Syntax: *breq k*

Operand:  $-64 \leq k \leq +63$

Operation: If status bit Z=1(e.g when Rd=Rr),  
then  $\text{PC} \leftarrow \text{PC}+k+1$ , else  $\text{PC} \leftarrow \text{PC}+1$

- Instruction format

1 1 1 1	0 0 k k	k k k k	k 0 0 1
---------	---------	---------	---------

- Execution time
  - 1 clock cycle if condition is false
  - 2 clock cycles if condition is true

# AVR Instructions

- AVR has the following classes of instructions:
  - Arithmetic and logic
  - Data transfer
  - Program execution flow control
  - Bit and others
    - Bit and Bit test
    - MCU control
- An overview of the instructions is given in the next slides.



•MCU: Microcontroller

# AL Instructions

- Arithmetic
  - addition
    - e.g. ADD Rd, Rr
  - subtraction
    - e.g. SUB Rd, Rr
  - increment/decrement
    - e.g INC Rd
  - multiplication
    - e.g. MUL Rd, Rr
- Logic
  - e.g. AND Rd, Rr
- Shift
  - e.g. LSL Rd

# Data Transfer Instructions

- GP register
  - e.g. MOV Rd, Rr
- I/O registers
  - e.g. IN Rd, PORTA  
OUT PORTB, Rr
- Stack
  - PUSH Rr
  - POP Rd
- Immediate values
  - e.g. LDI Rd, K8
- Memory
  - Data memory
    - e.g. LD Rd, X  
ST X, Rr
  - Program memory
    - e.g. LPM

# Execution Flow Control Instructions

- Branch
  - Conditional
    - Jump to address
      - E.g. BREQ dst
        - » test ALU flag and jump to specified address if the condition is true
      - Skip
        - E.g. SBIC A, k
          - » test a bit in a register or an IO register and skip the next instruction if the condition is true.
    - Unconditional
      - Jump to the specified address
        - E.g. RJMP dst
  - Call subroutine
    - E.g. RCALL k
  - Return from subroutine
    - E.g. RET

# Bit & Other Instructions

- Bit
  - Set bit
    - E.g. SBI PORTA, b
  - Clear bit
    - E.g CBI PORTA, b
  - Bit copy
    - E.g. BST Rd, b
- Others
  - NOP
  - BREAK
  - SLEEP
  - WDR

# AVR Instructions (cont.)

- Not all instructions are implemented in an Atmel microcontroller
  - Refer to the data sheet of a specific microcontroller
- Refer to online AVR instruction document for the detail description of each instruction
  - Get a general view of the instruction set
  - Learn each instruction when use it.

# AVR Addressing Mode

- Immediate
- Register direct
- Memory related addressing modes
  - Data memory
    - Direct
    - Indirect
    - Indirect with Displacement
    - Indirect with Pre-decrement
    - Indirect with Post-increment
  - Program memory

# Immediate Addressing

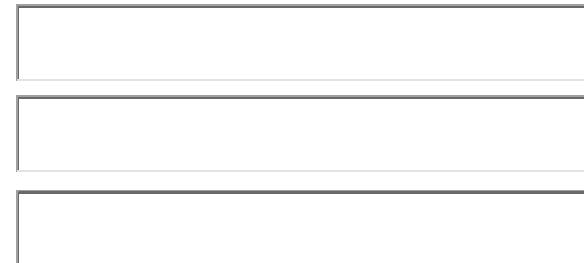
- The operand comes from instruction
- For example

`andi r16, $0F`

0x0F

X	Y	X•Y
0	0	0
0	1	0
1	0	0
1	1	1

- Bitwise logic AND operation
  - Clear left four bits in register r16



# Register Direct Addressing

- The operand comes from general purpose register
- For example

and r16, *r0*



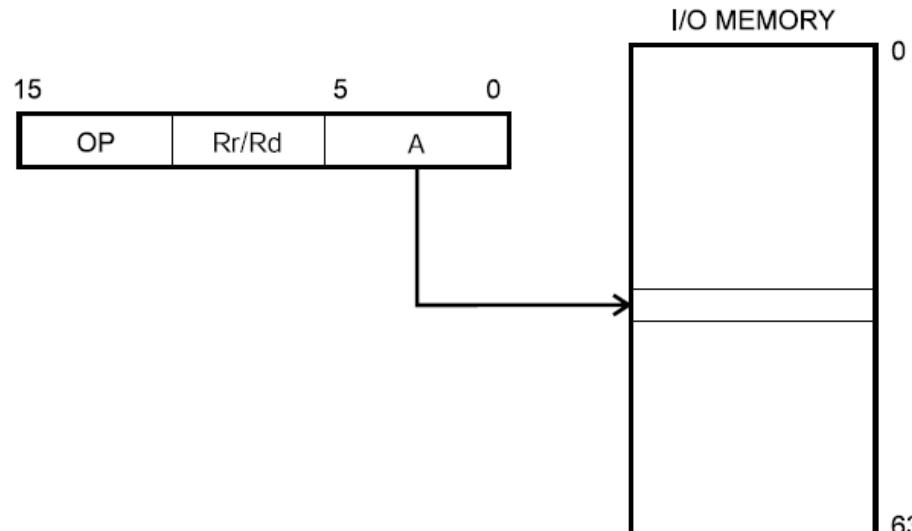
–  $r16 \leftarrow r16 \text{ AND } r0$

- Clear left four bits in register r16 if  $r0 = 0x0F$

# Register Direct Addressing

- The operand comes from the I/O registers
- For example

*in r25, PINA*  
-- r25 ← PIN A



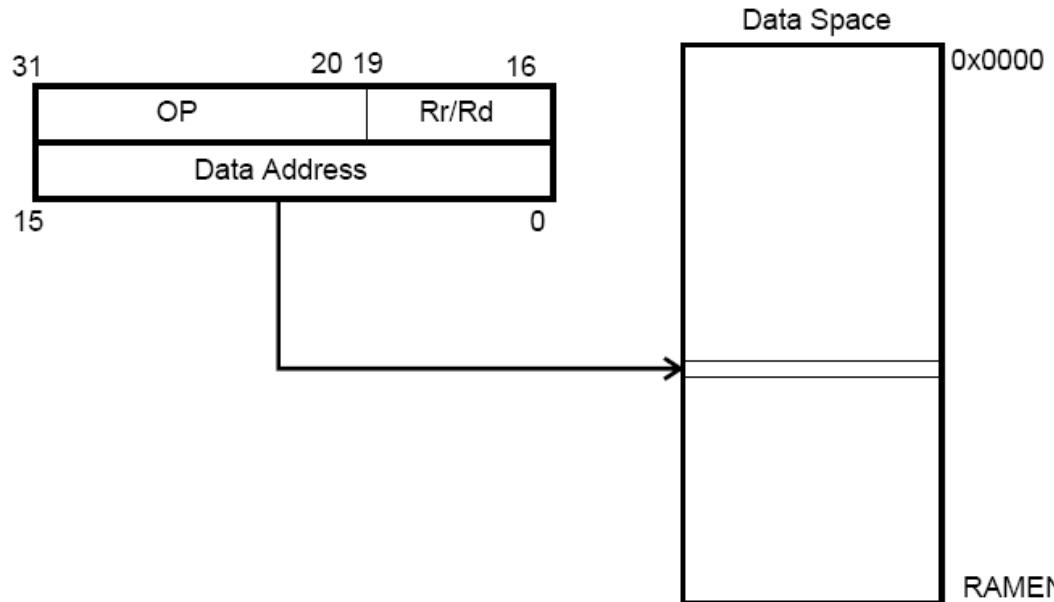
# Data Memory Addressing

# Data Memory Direct Addressing

- The data memory address is given directly from the instruction
- For example

*lds r5, \$F123*

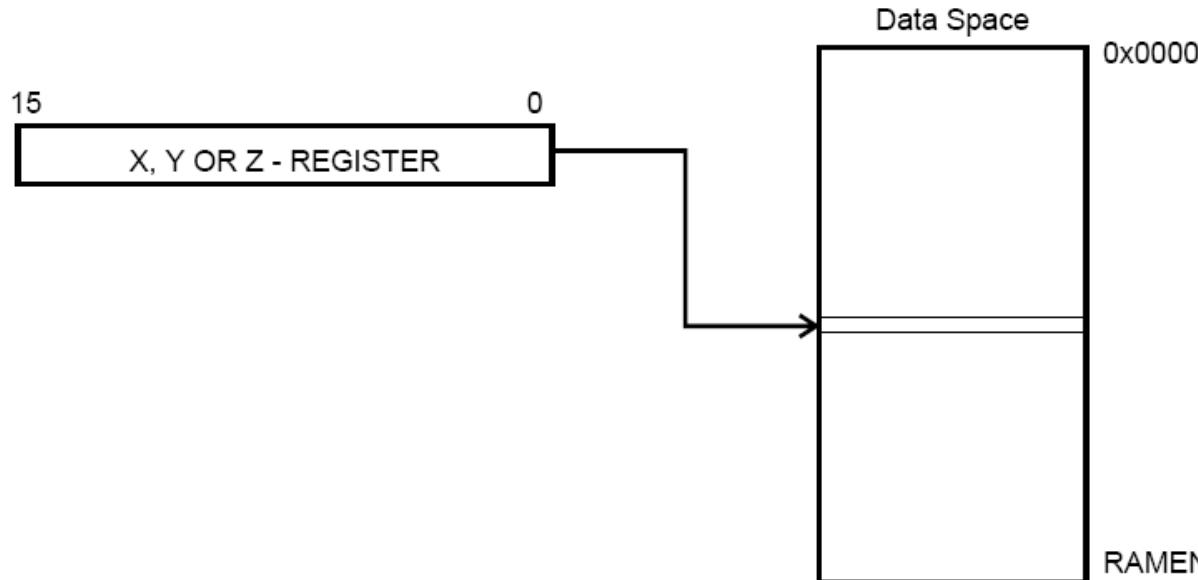
--  $r5 \leftarrow \text{Mem}(\$F123)$ , or  $r5 \leftarrow (\$F123)$



# Data Memory Indirect Addressing

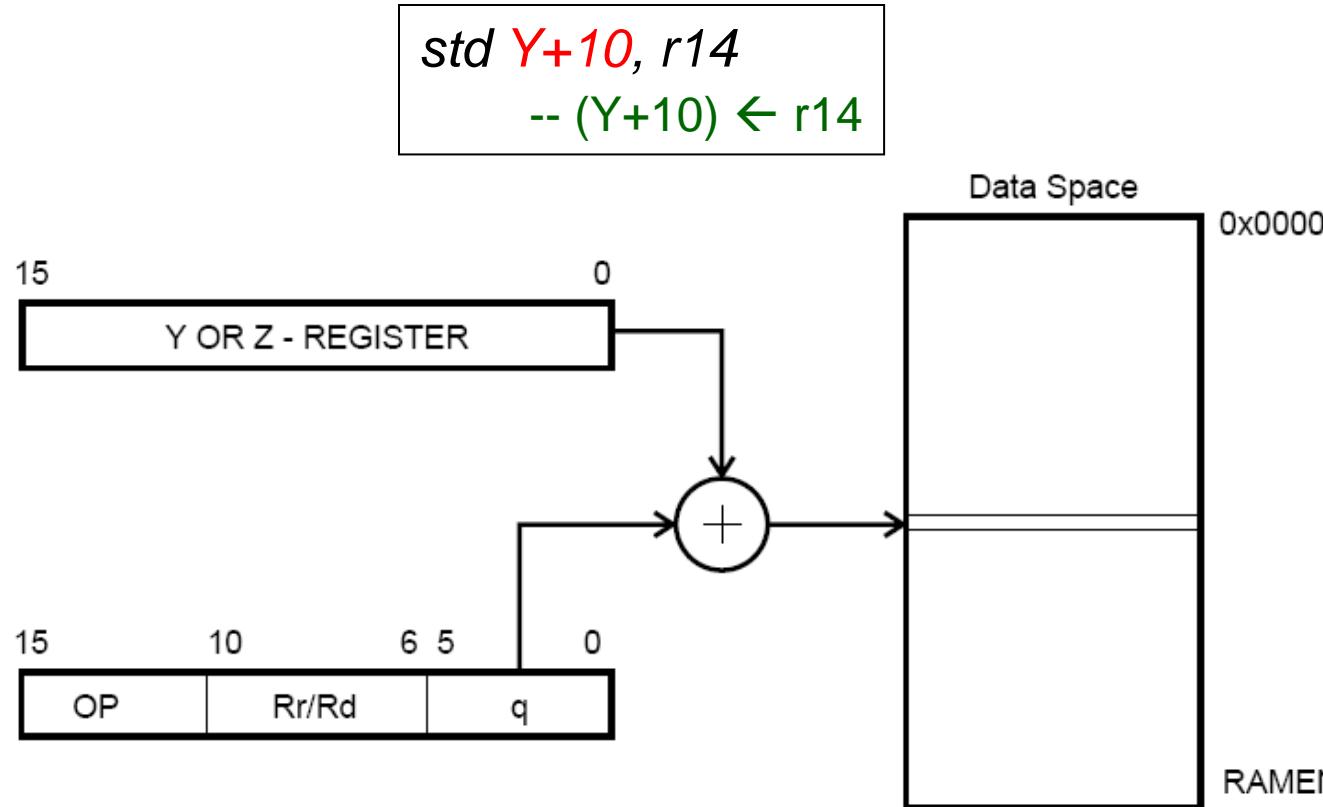
- The address of memory data is from an address pointer (X, Y, Z)
- For example

```
ld r11, X  
-- r11 ← Mem(X), or r11 ← (X)
```



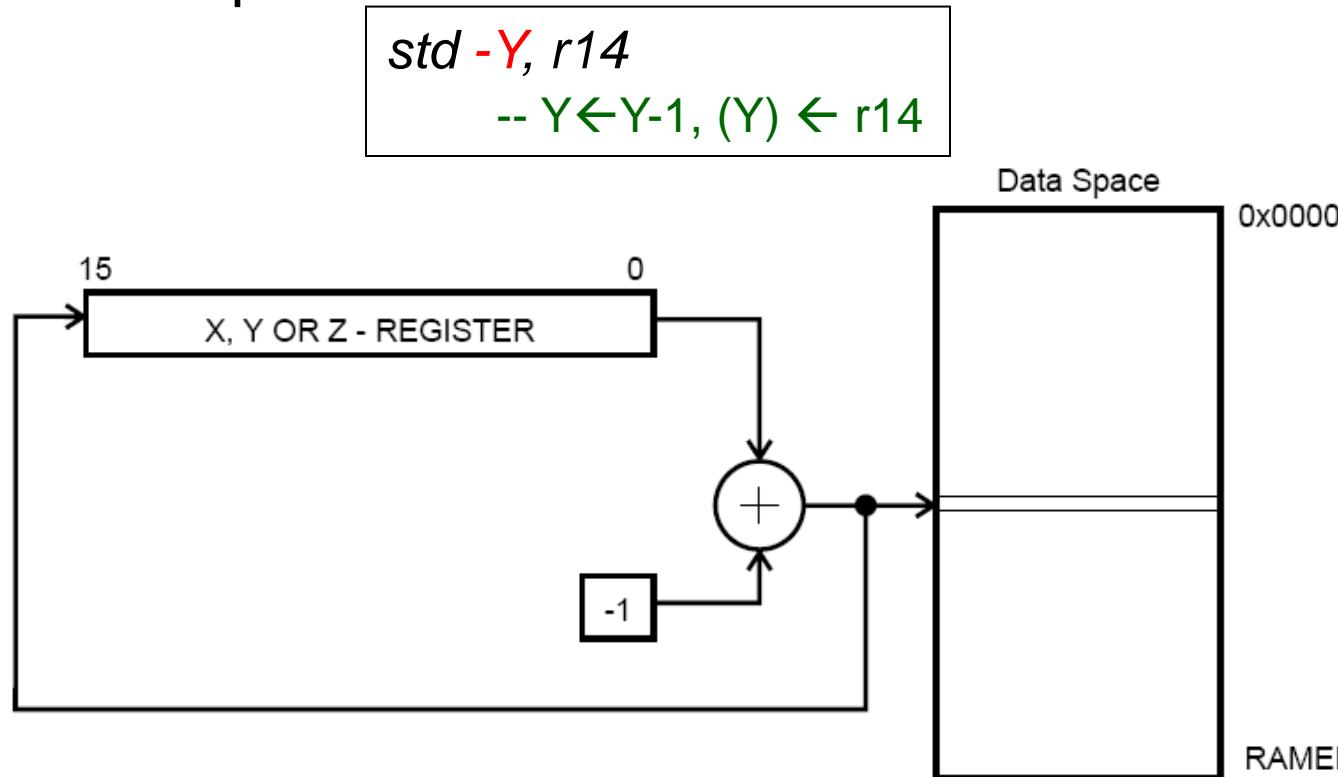
# Data Memory Indirect Addressing with Displacement

- The address of memory data is from  $(Y, Z) + q$ 
  - Offset  $q \geq 0$
- For example



# Data Memory Indirect Addressing with Pre-decrement

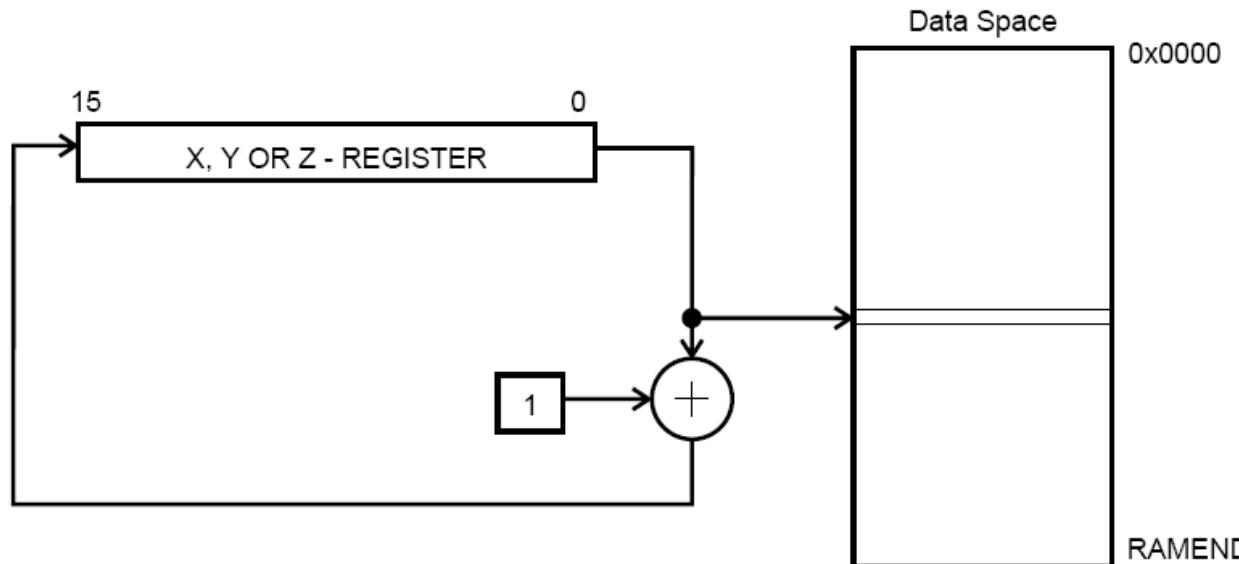
- The address of memory data is from an address pointer (X, Y, Z) and the value of the pointer is auto-decreased **before** each memory access.
- For example



# Data Memory Indirect Addressing with Post-increment

- The address of memory data is from an address pointer (X, Y, Z) and the value of the pointer is auto-increased **after** each memory access.
- For example

```
std Y+, r14  
--(Y) <- r14, Y <- Y+1
```

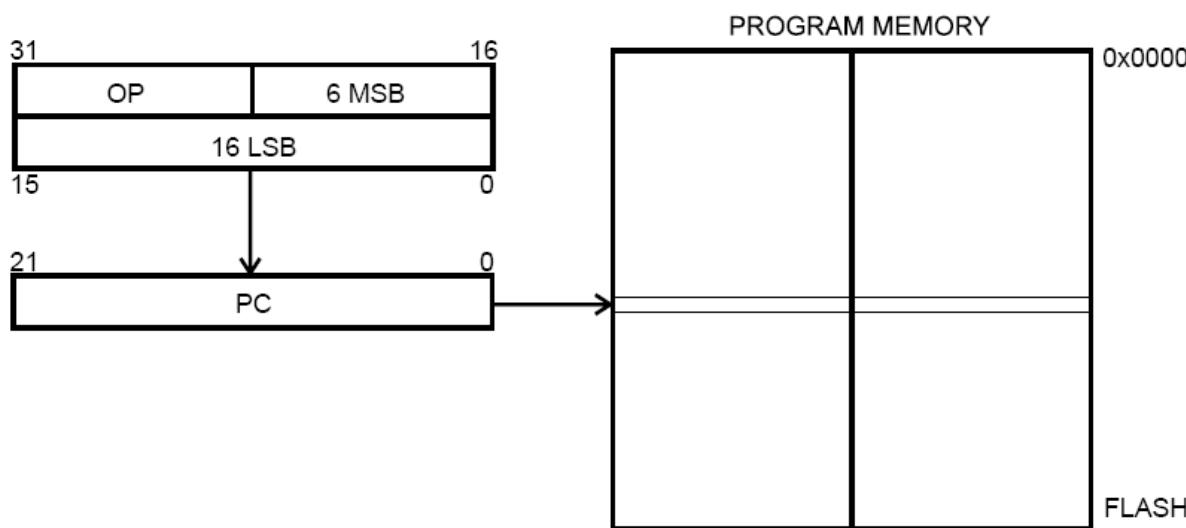


# Program Memory Addressing

# Program Memory Direct Addressing

- The instruction address is from instruction
- For example

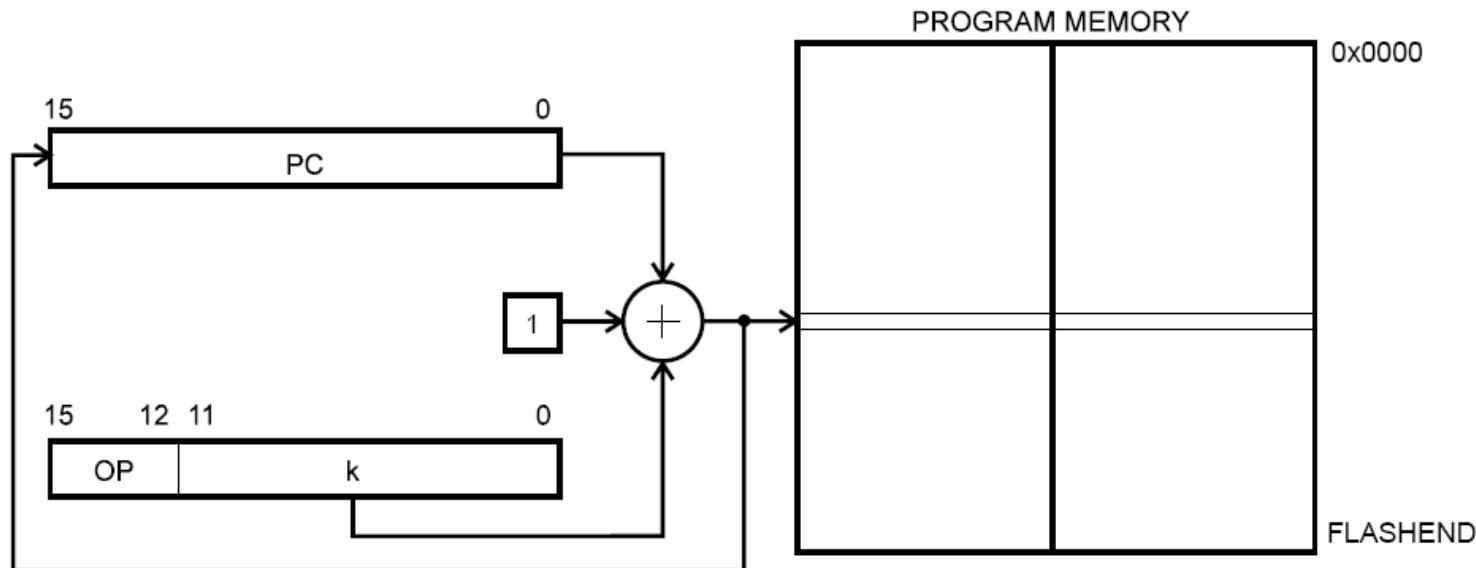
*jmp k*  
-- PC ← k



# Program Memory Relative Addressing

- The instruction address is  $\text{PC}+k+1$
- For example

*rjmp k*  
--  $\text{PC} \leftarrow \text{PC}+k+1$

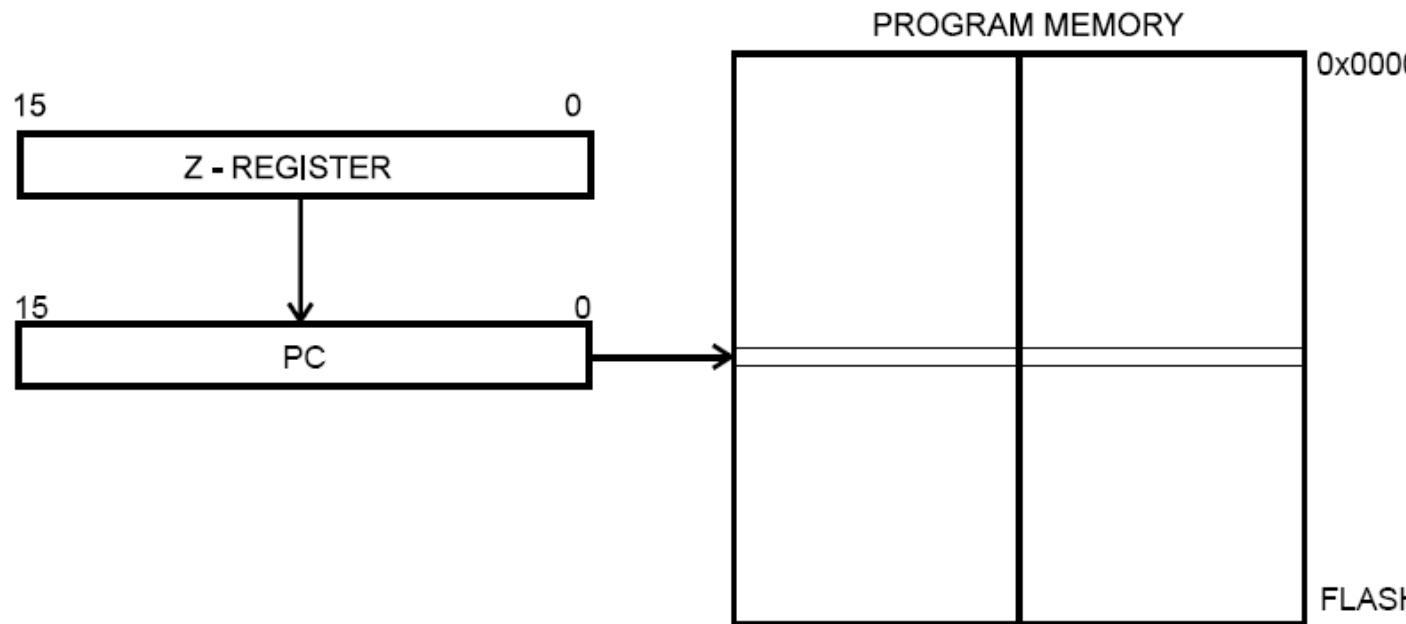


# Program Memory Indirect Addressing

- The instruction address is stored in **Z** register

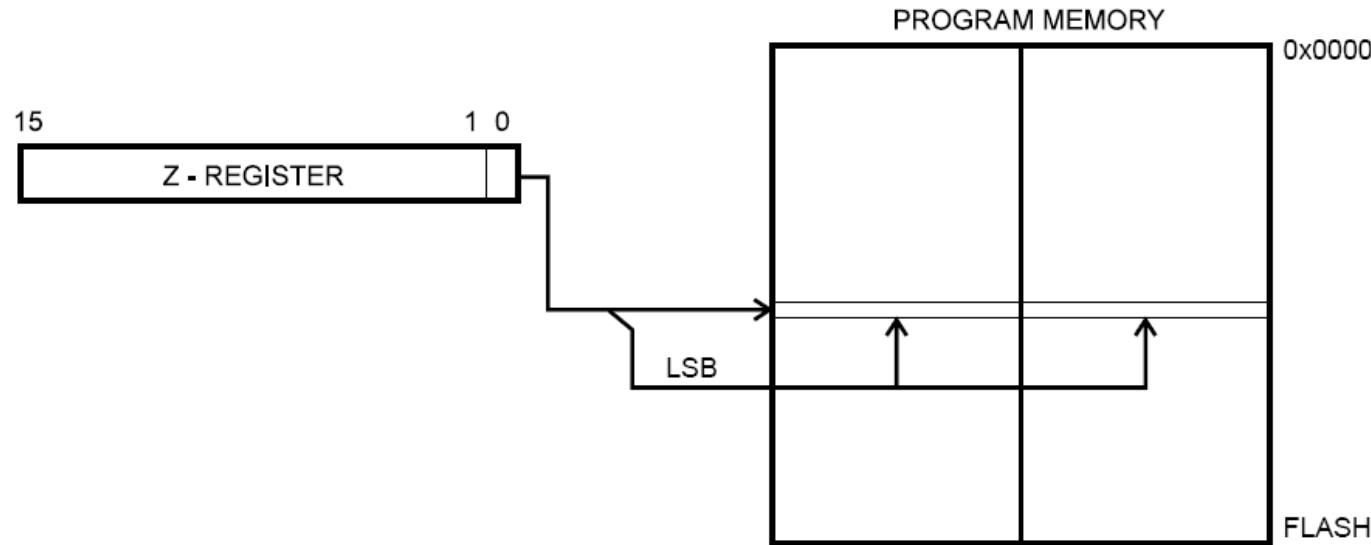
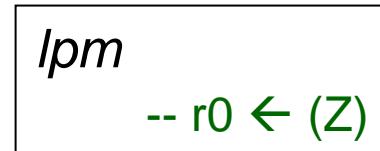
*icall*

-- PC(15:0)  $\leftarrow$  (Z), PC(21:16)  $\leftarrow$  0



# Program Memory Constant Addressing

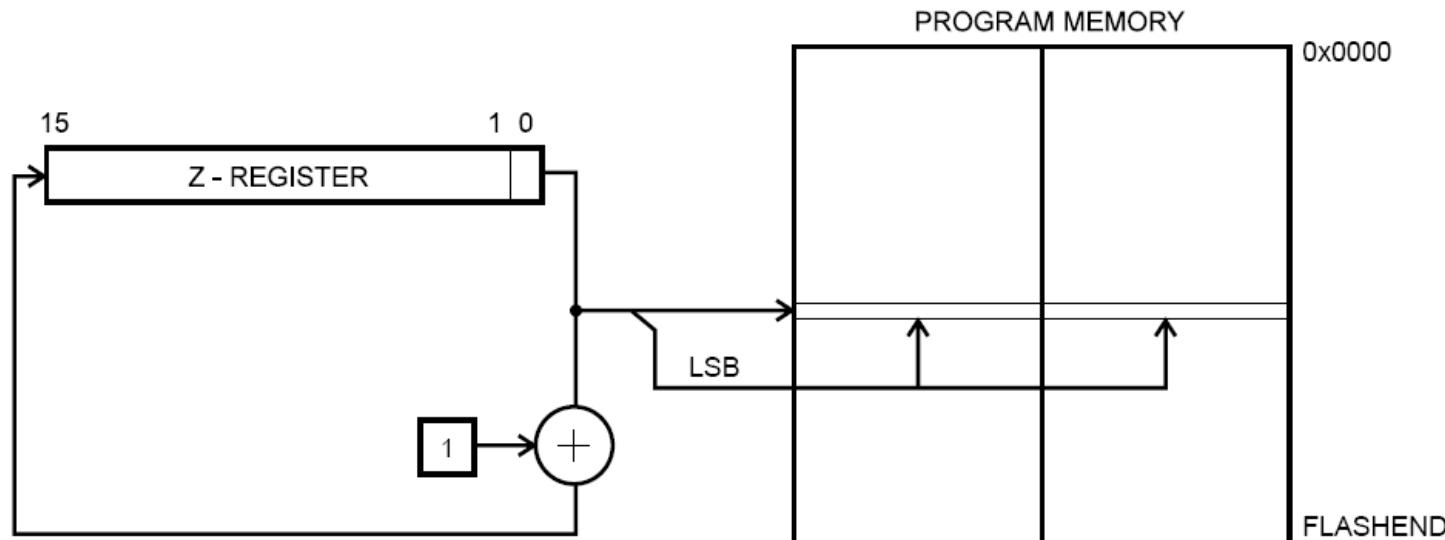
- The address of the **constant data** is stored in **Z** register
  - The address is **a byte address**.
- For example:



# Program Memory Constant Addressing with Post-increment

- For example

*lpm r16, Z+*  
-- r16  $\leftarrow$  (Z), Z  $\leftarrow$  Z+1



# AVR Programming

# AVR Programming

- Refer to the AVR Instruction Set document for the complete list of instructions
  - <http://www.cse.unsw.edu.au/~cs9032>, follow the link: References → Documents → AVR-Instruction-Set.pdf
  - We will learn individual instructions through
    - Lectures, homework, and lab exercises
- The rest of the lecture demonstrates AVR assembly programming
  - By implementing some basic structures with examples
    - Sequence
    - Selection
    - Iteration

# Sequence (1/5)

## - example

- Find the value of expression

$$z = 2x - xy - x^2$$

- where all values including the results from multiplications are **8-bit unsigned numbers**, and  $x$ ,  $y$ ,  $z$  are stored in registers r2, r3, and r4, respectively.

# What instructions do we need?

- sub
- mul

$$z = 2x - xy - x^2$$

# Subtract without Carry

- Syntax:            ***sub Rd, Rr***
- Operands:         $Rd, Rr \in \{r0, r1, \dots, r31\}$
- Operation:        $Rd \leftarrow Rd - Rr$
- Flags affected: H, S, V, N, Z, C
- Words:            1
- Cycles:           1

# Multiply Unsigned

- Syntax:            ***mul Rd, Rr***
- Operands:         $Rd, Rr \in \{r0, r1, \dots, r31\}$
- Operation:        $r1:r0 \leftarrow Rr^*Rd$ 
  - ( $\text{unsigned} \leftarrow \text{unsigned} * \text{unsigned}$  )
- Flags affected: Z, C
  - C is set if bit 15 of the result is set; cleared otherwise.
- Words:            1
- Cycles:           2

# What instructions do we need?

- sub
- mul
- ldi
- mov

# Load Immediate

- Syntax:  $ldi\ Rd, k$
- Operands:  $Rd \in \{r16, \dots, r31\}$ ,  $0 \leq k \leq 255$
- Operation:  $Rd \leftarrow k$
- Flag affected: None
- Words: 1
- Cycles: 1
- Encoding: 1110 kkkk dddd kkkk
- Example:  
`ldi r16, $42 ; Load $42 to r16`

8-bit binary

# Copy Register

- Syntax:            ***mov Rd, Rr***
- Operands:         $Rd, Rr \in \{r0, r1, \dots, r31\}$
- Operation:        $Rd \leftarrow Rr$
- Flag affected: None
- Words:            1
- Cycles:           1

# Sequence (2/5)

## - example

- AVR code for  $z = 2x - xy - x^2$ 
  - where all values including results from multiplications are 8-bit unsigned numbers; and  $x$ ,  $y$ ,  $z$  are stored in registers r2, r3, and r4, respectively.

ldi	r16, 2	; r16 $\leftarrow$ 2
mul	r16, r2	; r1:r0 $\leftarrow$ 2x
mov	r5, r0	; r5 $\leftarrow$ 2x
mul	r2, r3	; r1:r0 $\leftarrow$ xy
sub	r5, r0	; r5 $\leftarrow$ 2x-xy
mul	r2, r2	; r1:r0 $\leftarrow$ $x^2$
sub	r5, r0	; r5 $\leftarrow$ 2x-xy- $x^2$
mov	r4, r5	; r4 $\leftarrow$ z

- 8 instructions and 11 cycles

# Sequence (3/5)

- AVR code for  $z = 2x - xy - x^2$ 
  - where all data including products from multiplication are 8-bit unsigned numbers; and x, y, z are stored in registers r2, r3, and r4, respectively.
  - Improved Sol 1:

ldi	r16, 2	; r16 $\leftarrow$ 2
mul	r16, r2	; r1:r0 $\leftarrow$ 2x
mov	r4, r0	; r4 $\leftarrow$ 2x
mul	r2, r3	; r1:r0 $\leftarrow$ xy
sub	r4, r0	; r4 $\leftarrow$ 2x-xy
mul	r2, r2	; r1:r0 $\leftarrow$ x <sup>2</sup>
sub	r4, r0	; r4 $\leftarrow$ 2x-xy- x <sup>2</sup>

- 7 instructions and 10 cycles

# Sequence (4/5)

- Find the value of the expression

$$\begin{aligned}z &= 2x - xy - x^2 \\&= x(2 - (x + y))\end{aligned}$$

- where all data including products from multiplications are 8-bit unsigned numbers; and x, y, z are stored in registers r2, r3, and r4, respectively.

# What instructions do you need?

- sub
- mul
- ldi
- mov
- add

# Add without Carry

- Syntax:            ***add Rd, Rr***
- Operands:         $Rd, Rr \in \{r0, r1, \dots, r31\}$
- Operation:        $Rd \leftarrow Rd + Rr$
- Flags affected: H, S, V, N, Z, C
- Words:            1
- Cycles:           1

# Sequence (5/5)

- AVR code for

$$\begin{aligned}z &= 2x - xy - x^2 \\&= x(2 - (x + y))\end{aligned}$$

- where all data including products from multiplications are 8-bit unsigned numbers; and x, y, z are stored in registers r2, r3, and r4, respectively.
- Improved sol 2

mov	r4, r2	; r4 $\leftarrow$ x
add	r4, r3	; r4 $\leftarrow$ x+y
ldi	r16, 2	; r16 $\leftarrow$ 2
sub	r16, r4	; r16 $\leftarrow$ 2-(x+y)
mul	r2, r16	; r1:r0 $\leftarrow$ x(2-(x+y))
mov	r4, r0	; r4 $\leftarrow$ z

- 6 instructions and 7 cycles

# Selection (1/2)

## - example

- IF-THEN-ELSE control structure

```
if(a<0)
    b=1;
else
    b=-1;
```

- Assume  $a$ ,  $b$  are 8-bit **signed** integers and stored in registers. You need to decide which registers to use.
- Instructions involved:
  - Compare
  - Conditional branch
  - Unconditional jump

# Compare

- Syntax:  $cp \text{ } Rd, Rr$
- Operands:  $Rd \in \{r0, r1, \dots, r31\}$
- Operation:  $Rd - Rr$  ( $Rd$  is not changed)
- Flags affected: H, S, V, N, Z, C
- Words: 1
- Cycles: 1
- Example:

```
    cp r4, r5          ; Compare r4 with r5
    brne noteq         ; Branch if r4 ≠ r5
    ...
noteq: nop           ; Branch destination (do nothing)
```

# Compare with Immediate

- Syntax:        ***cpi Rd, k***
- Operands:       $Rd \in \{r16, r17, \dots, r31\}$  and  $0 \leq k \leq 255$
- Operation:      $Rd - k$  ( $Rd$  is not changed)
- Flags affected: H, S, V, N, Z, C
- Words:           1
- Cycles:          1

# Conditional Branch

- Syntax: ***brge k*** S bit in SREG: S=0
- Operands:  $-64 \leq k < 64$
- Operation: If  $Rd \geq Rr$  ( $N \oplus V = 0$ ) then  $PC \leftarrow PC + k + 1$ ,  
else  $PC \leftarrow PC + 1$  if condition is false
- Flag affected: None
- Words: 1
- Cycles: 1 if condition is false; 2 if condition is true

# Relative Jump

- Syntax:  $rjmp\ k$
- Operands:  $-2K \leq k < 2K$
- Operation:  $PC \leftarrow PC + k + 1$
- Flag affected: None
- Words: 1
- Cycles: 2

# Selection (2/2)

- IF-THEN-ELSE control structure

```
if(a<0)
    b=1;
else
    b=-1;
```

- Numbers *a*, and *b* are 8-bit **signed integers** and stored in registers. You need to decide which registers to use.

```
.def    a=r16
.def    b=r17
        cpi    a, 0           ;a-0
        brge   ELSE           ;if a≥0, go to ELSE
        ldi    b, 1           ;b=1
        rjmp   END             ;end of IF statement
ELSE:   ldi    b, -1          ;b=-1
END:    ...
```

# Iteration (1/2)

- WHILE loop

```
sum =0;  
i=1;  
while (i<=n){  
    sum += i*i;  
    i++;  
}
```

- Numbers  $i$  and  $sum$  are 8-bit unsigned integers and stored in registers. You need to decide which registers to use.

# Iteration (2/2)

- WHILE loop

```
.def      i = r16
.def      n = r17
.def      sum = r18

ldi i, 1                      ;initialization
clr sum

loop:
    cp n, i
    brlo end
    mul i, i
    add sum, r0
    inc i
    rjmp loop

end:
    rjmp end
```

# Reading Material

- AVR Instruction Set online document about:
  - Instruction set nomenclature
  - I/O Registers (can skip for now)
  - The program and data memory Addressing
  - Arithmetic instructions, program execution flow control instructions

# Homework

1. Refer to the AVR Instruction Set document (available at <http://www.cse.unsw.edu.au/~cs9032>, under the link References → Documents → AVR-Instruction-Set.pdf).

Study the following instructions:

- **Arithmetic and logic instructions**
  - add, adc, adiw, sub, subi, sbc, sbci, sbiw, mul, muls, mulsu
  - and, andi, or, ori, eor
  - com, neg

# Homework

## 1. Study the following instructions (cont.)

- Branch instructions
  - cp, cpc, cpi
  - rjmp
  - breq, brne
  - brge, brlt
  - brsh, brlo
- Data transfer instructions
  - mov
  - ldi, ld, st

# Homework

2. Write assembly code for the following functions

- 1) 2-byte addition (i.e, addition on 16-bit numbers)
- 2) 2-byte signed subtraction
- 3) Sign-extension of one byte value to two bytes

# **Microprocessors & Interfacing**

## *Number Conversion*

Lecturer : Annie Guo

# Number Representation

- Any types of numbers can be represented in the form of

$$(a_n a_{n-1} \dots a_1 a_0 . a_{-1} \dots a_{-m})_r$$
$$= a_n \times r^n + a_{n-1} \times r^{n-1} + \dots + a_1 \times r + a_0 + a_{-1} \times r^{-1} + \dots + a_{-m} \times r^{-m}$$

**r : radix, base**

**$0 \leq a_i < r$**

# Example 1

- Decimal number

$$\begin{aligned} (3597)_{10} \\ = 3 \times 10^3 + 5 \times 10^2 + 9 \times 10 + 7 \end{aligned}$$

- The base or radix is 10
- All digits must be less than the base, namely be one of 0~9
- The place values, from right to left, are  $10^0$ ,  $10^1$ ,  $10^2$ ,  $10^3$

# Example 2

- Binary number

$$\begin{aligned} & (1011)_2 \\ & = 1 \times 2^3 + 0 \times 2^2 + 1 \times 2 + 1 \end{aligned}$$

- The base or radix is 2
- All digits must be less than the base, namely, be one of 0~1
- The place values, from right to left, are  $2^0, 2^1, 2^2, 2^3$

# Example 3

- Hexadecimal number

$$\begin{aligned}(\text{F24B})_{16} &= \text{F} \times 16^3 + 2 \times 16^2 + 4 \times 16 + \text{B} \\ &= 15 \times 16^3 + 2 \times 16^2 + 4 \times 16 + 11\end{aligned}$$

- The base or radix is 16
- All digits must be less than the base, namely, 0~9,A,B,C,D,E,F
- The place values, from right to left, are  $16^0$ ,  $16^1$ ,  $16^2$ ,  $16^3$

# Number Conversion

- From base  $r$  to base 10
  - By using expended form

$$(a_n a_{n-1} \dots a_1 a_0 . a_{-1} \dots a_{-m})_r = a_n \times r^n + a_{n-1} \times r^{n-1} + \dots + a_1 \times r + a_0 + a_{-1} \times r^{-1} + \dots + a_{-m} \times r^{-m}$$

- Examples (next slide)

# Example 4

- From base 2

$$(1011.1)_2 = 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 + 1 \times 2^{-1} = 11.5$$

- From base 16

$$(10A)_{16} = 1 \times 16^2 + 0 \times 16^1 + 10 = 266$$

# Number Conversion (cont.)

- From base 10 to base r

Based on the expended form of r-base number, we can get digits for whole number part and digits for fraction part

$$(a_n a_{n-1} \dots a_1 a_0 . a_{-1} \dots a_{-m})_r \\ = a_n \times r^n + a_{n-1} \times r^{n-1} + \dots + a_1 \times r + a_0 + a_{-1} \times r^{-1} + \dots + a_{-m} \times r^{-m}$$

- For the whole number part,  $a_n a_{n-1} \dots a_1 a_0$ 
  - Divide** the number/quotient repeatedly by r until the quotient is zero.
  - The remainders from the divisions are the digits of the whole number part in reverse order
- For the fraction part,  $a_{-1} \dots a_{-m}$ 
  - Multiply** the number/fraction repeatedly by r, the whole numbers of the products are the digits of the fraction part.

# Example 5

- To base 2
  - To convert  $(11.25)_{10}$  to binary
    - For whole number part  $(11)_{10}$  – repeated division (by 2)

$$\begin{array}{r} 11 \\ \hline 5 \\ \hline 2 \\ \hline 1 \\ \hline 0 \end{array} \quad \begin{array}{l} 1 \\ 1 \\ 0 \\ 1 \end{array}$$

↑

- For fraction part  $(0.25)_{10}$  – repeated multiplication (by 2)

$$\begin{array}{r} 0.25 \\ \times 2 \\ \hline 0.5 \\ \times 2 \\ \hline 0.0 \end{array} \quad \begin{array}{l} 0 \\ 1 \end{array}$$

↓

$$(11.25)_{10} = (1011.01)_2$$

# Example 6

- To base 16
  - To convert  $(99.25)_{10}$  to hexadecimal
    - For whole number  $(99)_{10}$  – repeated division **(by 16)**

$$\begin{array}{r} 99 \\ \hline 6 \\ \hline 0 \end{array} \quad \begin{array}{c} 3 \\ 6 \\ \uparrow \end{array}$$

- For fraction  $(0.25)_{10}$  – repeated multiplication **(by 16)**

$$\begin{array}{r} 0.25 \\ \times 16 \\ \hline 0.0 \end{array} \quad \begin{array}{c} 4 \\ \downarrow \end{array}$$

$$(99.25)_{10} = (63.4)_{\text{hex}}$$

# Number Conversion (cont.)

- Between binary and hexadecimal
  - Binary to hexadecimal
    - The binary digits are grouped from the radix point, **four** binary digits a group. Each group corresponds to a hexadecimal digit.
  - Hexadecimal to binary
    - Each of hexadecimal digits is expanded to four binary digits.

# Example 7

- Binary to hexadecimal
  - Convert  $10101100011010001000.10001_2$  to hexadecimal :
$$\begin{aligned} & 1010 \ 1100 \ 0110 \ 1000 \ 1000 . \ 1000 \ 1000_2 \\ = & \quad A \quad C \quad 6 \quad 8 \quad 8 \quad . \quad 8 \quad 8 \quad 8_{16} \\ = & \text{AC688.88}_{16} . \end{aligned}$$
- Note:
  - The whole number part digits are grouped from right to left. The leading 0s are optional
  - The fractional part digits are grouped from left to right and padded with 0s

# Example 8

- Hexadecimal to binary

- Convert  $2F6A.78_{16}$  to binary :

$$\begin{aligned} & \quad 2 \quad F \quad 6 \quad A \quad . \quad 7 \quad 8_{16} \\ = & 0010\ 1111\ 0110\ 1010\ .\ 0111\ 1000_2 \\ = & 10111101101010.01111_2 \end{aligned}$$

- Note:

- For the whole number part, the leading 0's can be omitted.
  - For the fractional part, the trailing 0's can be omitted.

# **Microprocessors & Interfacing**

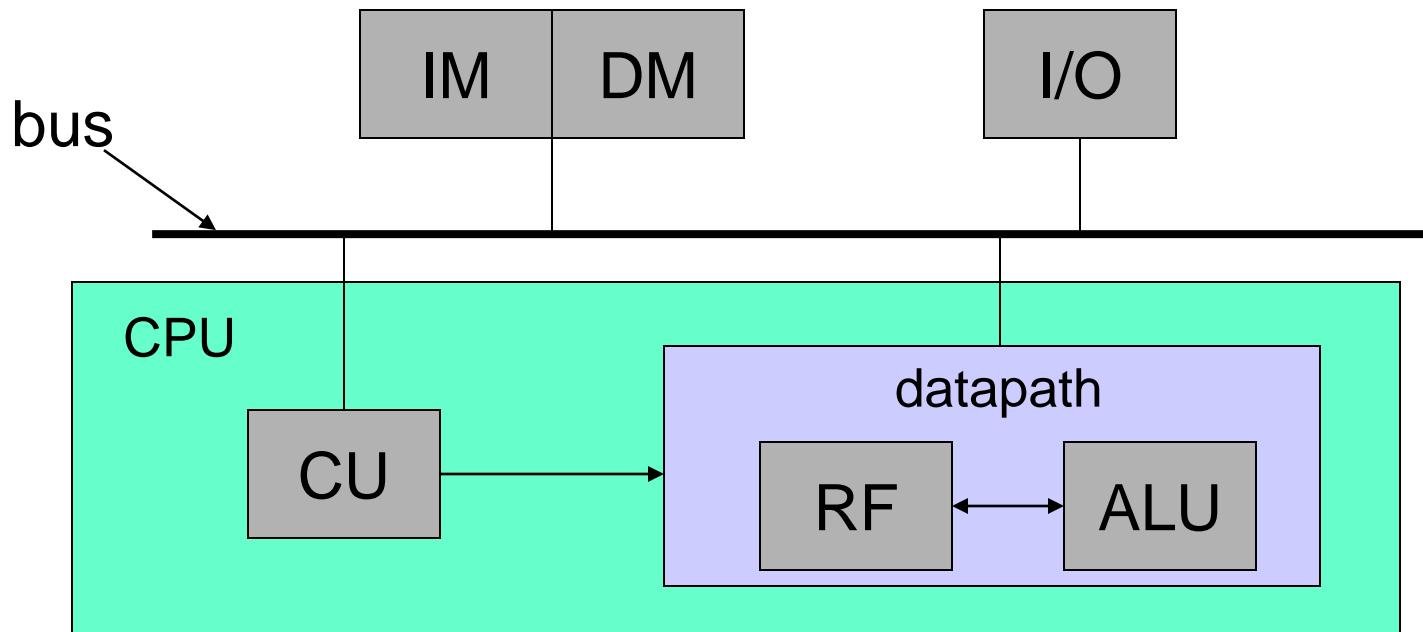
*Basics of Computing with  
Microprocessor Systems*

Lecturer: Annie Guo

# Lecture Overview

- Basic Microprocessor Hardware Structure
- Data Representation
  - Binary
  - Hexadecimal
- Instruction Set Architecture

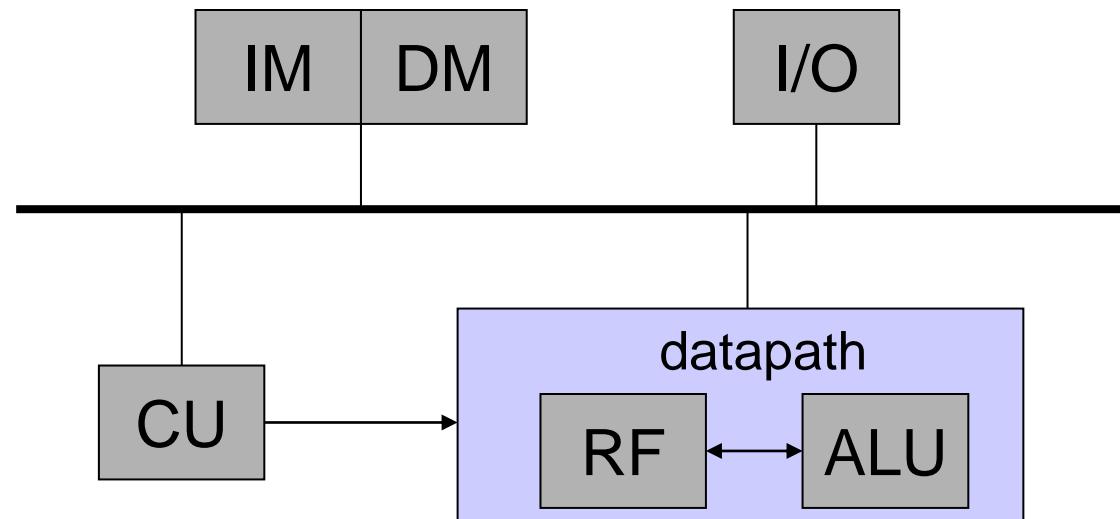
# Fundamental Hardware Components in Computing System



- **ALU:** Arithmetic and Logic Unit
- **RF:** Register File (a set of registers)
- **CU:** Control Unit
- **IM/DM:** Instruction/Data Memory
- **I/O:** Input/Output Devices

# Execution Cycle

**IF:** Instruction Fetch  
↓  
**ID:** Instruction Decode  
↓  
**RR:** Read Register File  
↓  
**EX:** Execution  
↓  
**WR:** Write Result  
↓  
**NPC:** Next Instruction



Note: Steps can be merged/broken down/expanded

# Microprocessor

- A *microprocessor* is the datapath and control unit on a single chip.
  - Note, it often includes other components for functional and performance enhancement
- If a microprocessor, its associated support circuitry, memory and peripheral I/O components are implemented on a single chip, it is a *microcontroller*.
  - We use Atmel AVR microcontroller as the example in our course



# Data Representation

- For a digital microprocessor system to be able to compute and process data, the data must be properly represented
  - How to represent numbers for calculation?
    - Binary number
    - Binary code
  - How to represent characters, symbols and other values for processing?
    - Will be covered later

# Decimal

- Decimal

$$\begin{aligned} (3597)_{10} \\ = 3 \times 10^3 + 5 \times 10^2 + 9 \times 10 + 7 \end{aligned}$$

- The base or radix is 10
- All digits must be less than the base, namely, 0~9
- The place values, from right to left, are  $10^0$ ,  $10^1$ ,  $10^2$ ,  $10^3$

# Binary

- Example

$$\begin{aligned} & (1011)_2 \\ & = 1 \times 2^3 + 0 \times 2^2 + 1 \times 2 + 1 \end{aligned}$$

- All digits (aka bits) must be less than 2 (0~1).

What are the first 16 binary integers?

# Hexadecimal

- Example

$$\begin{aligned}(\text{F24B})_{16} &= \text{F} \times 16^3 + 2 \times 16^2 + 4 \times 16 + \text{B} \\ &= 15 \times 16^3 + 2 \times 16^2 + 4 \times 16 + 11\end{aligned}$$

- All digits must be less than 16 (0~9,A,B,C,D,E,F)
- Conversion between binary to hexadecimal
  - One hexadecimal digit  $\leftrightarrow$  4 binary digits
  - E.g. 0xA1

0b110111  
  ↑  
  ↑



# Binary vs Decimal

- Conversion
  - binary to decimal
    - Based on the expanded form

$$(a_n a_{n-1} \dots a_1 a_0)_r = a_n \times r^n + a_{n-1} \times r^{n-1} + \dots + a_1 \times r + a_0$$

r: radix, base  
 $0 \leq a_i < r$

- $r=2$
- decimal to binary
  - Repeated division by 2
    - Each division generates a remainder → binary digit

# Example

- To binary
  - To convert  $(11)_{10}$  to binary

$$\begin{array}{r} \boxed{11} & 1 \\ \hline 5 & 1 \\ \hline 2 & 0 \\ \hline 1 & 1 \\ \hline 0 & \end{array}$$

↑

A vertical division algorithm diagram for converting decimal 11 to binary. It shows the quotient and remainder at each step: 11 divided by 2 is 5 with remainder 1; 5 divided by 2 is 2 with remainder 1; 2 divided by 2 is 1 with remainder 0; 1 divided by 2 is 0 with remainder 1. The remainders 1, 0, 1, 1 are aligned vertically to the right of the quotient digits 1, 1, 0, 1. An upward-pointing arrow is positioned to the right of the quotient digits.

$$(11)_{10} = (1011)_2$$

# Hexadecimal vs Decimal

- Conversion
  - Hexadecimal to decimal
    - Based on the expanded form

$$(a_n a_{n-1} \dots a_1 a_0)_r = a_n \times r^n + a_{n-1} \times r^{n-1} + \dots + a_1 \times r + a_0$$

r: radix, base  
 $0 \leq a_i < r$

- $r=16$
- decimal to hexadecimal
  - Repeated division by 16
    - Each division generates a remainder → hexadecimal digit

# Example

- To hexadecimal
  - To convert  $(99)_{10}$  to hexadecimal

$$\begin{array}{r} 99 \\ \hline 6 \\ 0 \end{array} \quad \begin{array}{l} 3 \\ 6 \\ \uparrow \end{array}$$

$$(99)_{10} = (63)_{\text{hex}}$$

How many divisions do we need to obtain its binary number?

# Arithmetic Operations of Binary Numbers

- Similar to decimal arithmetic operations
- Examples of addition and multiplication are given in the next two slides.

# Binary Addition

- Example:
  - Addition of two 4-bit binary numbers. How many bits are required for holding the result?

$$1001 + 0110 = (\underline{\hspace{2cm}})$$

# Binary Multiplication

- Example:
  - Multiplication of two 4-bit unsigned binary numbers. How many bits are required for holding the result?

$$1001 * 0110 = (\underline{\hspace{2cm}})$$

# Binary Subtraction

- Subtraction can be defined as addition of the additive inverse (namely signed addition)

$$a - b = a + (-b)$$

- We use **two's complement code/number**,  $b^*$ , to represent  $-b$ .

- for n-bit number

$$b^* = 2^n - b = \boxed{(2^n - 1) - b} + 1$$

bit-wise inversion

- $(b^*)^* = b$
- The **MSB** (Most Significant Bit) of a 2's complement code is the **sign bit**
  - MSB=1, negative number; MSB=0, positive number
  - For example, for a 4-bit 2's complement number
  - $(1001) \rightarrow -7, (0111) \rightarrow 7$

# Exercise 1

- For each of the following decimal numbers, what is its 8-bit 2's complement number?

(a) 7

(b) 127

(c) -12

- An  $n$ -bit binary number can be **interpreted** in two different ways: **signed** (i.e., 2's complement number) or **unsigned**. What decimal value does the 4-bit number, 1011, represent in each of the following two cases?

(a) if it is a signed number

(b) if it is an unsigned number

# Signed Addition

- E.g. 4-bit 2's-complement additions/subtractions

(1)  $0101 + 0010$  ( $5 + 2$ ):

$$\begin{array}{r} 0101 \\ + 0010 \\ \hline = 00111 \end{array}$$

(2)  $0101 - 0010$  ( $5 - 2$ ):

$$\begin{array}{r} 0101 \\ + 1110 \quad (= 0010^*) \\ \hline = 10011 \end{array}$$

(3)  $0010 - 0101$  ( $2 - 5$ ):

$$\begin{array}{r} 0010 \\ + 1011 \quad (= 0101^*) \\ \hline = 1101 \quad (= 0011^*) \end{array}$$

Result means -3.

(4)  $-0101 - 0010$  ( $-5 - 2$ ):

$$\begin{array}{r} 1011 \quad (= 0101^*) \\ + 1110 \quad (= 0010^*) \\ \hline = 11001 \end{array}$$

Result means -7.

# Overflow

- In digital computer systems, values are represented by a fixed number of bits.
- Overflow happens when the calculation result is beyond the range that can be represented with the given number size.

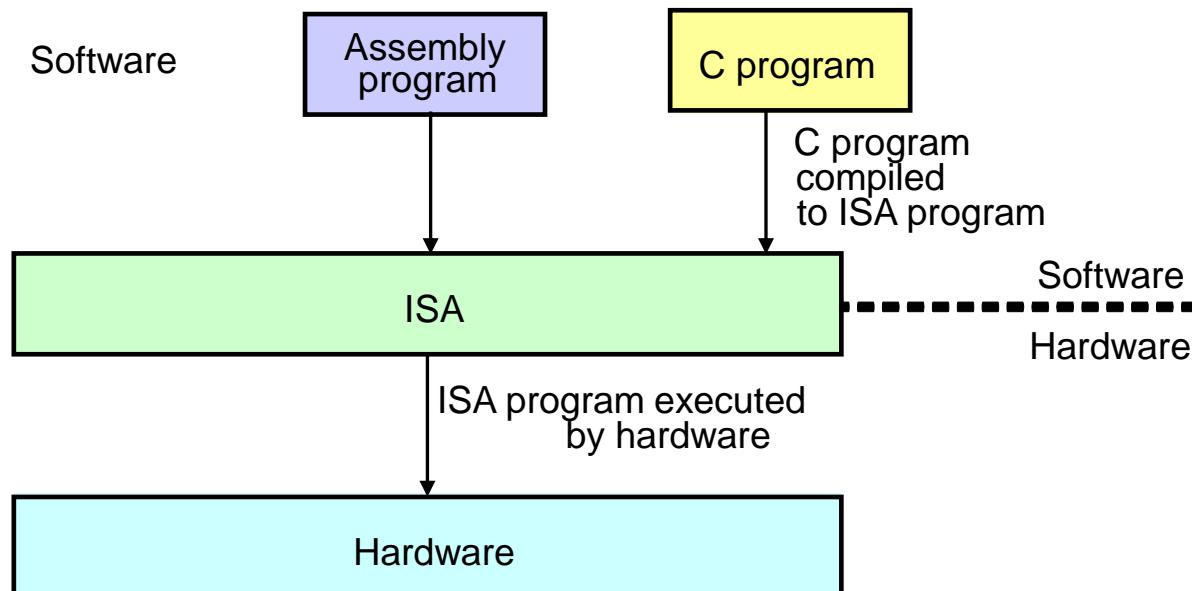
# Exercise 2

For the following 4-bit **signed** calculations, check whether there are any overflows.

- 1) 1000-0001
- 2) 1000+0101
- 3) 0101+0110

# Microprocessor Applications

- A microprocessor application system can be abstracted in a three-level structure
  - ISA (Instruction Set Architecture) is the interface between hardware and software



# Instruction Set

- Instruction set provides the vocabulary and grammar for programmer/software to communicate with the hardware machine.
- It is machine oriented
  - Different type of machines have a different instruction set
    - For example
      - 68K has a more comprehensive instruction set than ARM
      - Same operations could be represented differently in different machines
        - AVR
          - Addition: *add r2, r1* ; $r2 \leftarrow r2+r1$
          - Branching: *breq 6* ;branch if equal condition is true
          - Load: *ldi r30, \$F0* ; $r30 \leftarrow F0$
        - 68K:
          - Addition: *add d1,d2* ; $d2 \leftarrow d2+d1$
          - Branching: *breq 6* ;branch if equal condition is true
          - Load: *mov #1234, d2* ; $d2 \leftarrow 1234$

# Instructions

- Instructions can be written in two languages
  - Machine language
    - Binary representation
    - Used by machines
  - Assembly language
    - Textual representation
    - Easier to understand than machine language
    - Used by human being

# Machine Code vs. Assembly Code

- Basically, there is a one-to-one mapping between machine instructions and assembly instructions
  - For example, AVR instruction for incrementing register r16 by 1:
    - 1001010100000011 (machine code)
    - inc r16 (assembly code)
- Assembly language also includes **directives**
  - Directives
    - Instructions to the assembler
      - **Assembler** is a program to translate assembly code into machine code.
    - Example:
      - **.def temp = r16**
      - **.include "m2560def.inc"**

# Instruction Set Architecture (ISA)

- ISA specifies all aspects of a computer architecture visible to a programmer
  - Instructions (just mentioned)
  - Native data types
  - Registers
  - Memory models
  - Addressing modes

# Native Data Types

- Different machines support different data types in hardware
  - e.g. Pentium II:

Data Type	8 bits	16 bits	32 bits	64 bits	128 bits
Signed integer	✓	✓	✓		
Unsigned integer	✓	✓	✓		
BCD integer	✓				
Floating point			✓	✓	

- e.g. Atmel AVR (we are using):

Data Type	8 bits	16 bits	32 bits	64 bits	128 bits
Signed integer	✓				
Unsigned integer	✓				
BCD integer					
Floating point					

# Registers

- Two types
  - General purpose
  - Special purpose
    - e.g.
      - Program Counter (PC)
      - Status Register
      - Stack Pointer (SP)
      - Input/Output Registers
    - Stack Pointer and Input/Output Registers will be discussed in detail later.

# General Purpose Registers

- A set of registers in the machine
  - Used for storing temporary data/results
  - For example
    - In (68K) instruction add d3, d5, the operands of the operation are stored in general registers d3 and d5, and the result is stored in d5.
- Can be structured differently in different machines
  - For example
    - Separate general-purpose registers for data and address
      - 68K
    - Different number of registers and different size of registers
      - 32 32-bit registers in MIPS
      - 16 32-bit registers in ARM

# Program Counter (PC)

- Special register
  - For storing the memory address of currently executed instruction
- Can be of different size
  - E.g. 16 bits, 32 bits
- Can be auto-incremented
  - By the instruction word size
  - Hence, giving rise to the name “counter”

# Status Register

- Contains a number of bits with each bit being associated with processor (CPU) operations
- Typical status bits
  - V: Overflow
  - C: Carry
  - Z: Zero
  - N: Negative
- Used for controlling the program execution flow

# Memory Model

- Related to how memory is used to store data
- Issues
  - Addressable unit size
  - Address spaces
  - Endianness
  - Alignment

# Addressable Unit Size

- Memory has units, each of which has an address
- Most basic unit size is 8 bits (1 byte)
  - Related addresses are called **byte-addresses**.
- Modern processors can have multiple-byte unit
  - e.g. 32-bit instruction memory in MIPS  
16-bit instruction memory in AVR
  - Related addresses are called **word-addresses**.

# Address Space

- The range of addresses a processor can access.
  - A processor can have one or more address spaces. For example
    - Princeton architecture or Von Neumann architecture
      - A single linear address space for both instructions and data memory
    - Harvard architecture
      - Separate address spaces for instruction and data memories

# Address Space (cont.)

- Address space is not necessarily just for “memory”
  - E.g, all general-purpose registers and I/O registers can be accessed through memory addresses in AVR

# Endianness

- Memory objects
  - Memory objects are basic entities that can be accessed as a function of the **address** and the **size**
    - E.g. bytes, words, longwords
- For large objects (multiple bytes), there are two byte-ordering conventions
  - **Little endian** – little end (least significant byte) stored first (at lowest address)
    - Intel microprocessors (Pentium etc)
  - **Big endian** – big end (most significant byte) stored first
    - SPARC, Motorola microprocessors

# Big Endian & Little Endian

- Example: 0x12345678—a long word of 4 bytes. It is stored in the memory from a byte address 0x00000014

– big endian:

Address	data
0x00000014	0x12
0x00000015	0x34
0x00000016	0x56
0x00000017	0x78

– little endian:

Address	data
0x00000014	0x78
0x00000015	0x56
0x00000016	0x34
0x00000017	0x12

# Alignment

- Modern computers read from or write to a memory address in fix-sized chunks
    - for example, word size
      - 2 bytes, 4 bytes, 8 bytes ...
  - Alignment improves the memory access efficiency by putting the data at a memory address that is multiple of the chunk size
    - for example, with AVR, data of the word type in the program memory are aligned with the word addresses.

# Addressing Mode

- Instructions need to specify where to get operands
- Some possible ways
  - an operand value is in the instruction
  - an operand value is in a register
    - the register number is given in the instruction
  - an operand value is in memory
    - address is given in the instruction
    - address is given in a register
      - the register number is in the instruction
    - address is a register content plus some offset
      - register number is in the instruction
      - offset is in the instruction (or in a register)
- These ways of specifying the operand locations are called **addressing mode**.

# Addressing Mode (cont.)

- Some addressing mode examples, based on the 68K machine, are demonstrated in the next slides.
  - Using instruction: *addw a, b*
    - *addition on operands of the word size,  $b \leftarrow a+b$*
- For each addressing mode, there are
  - a general description and
  - an example to show how the address mode is used.
    - the specified addressing mode for the first operand of instruction is highlighted in red.

# Immediate Addressing

- The data is from the instruction
  - i.e the operand is immediately available from the instruction
- For example, in 68K

```
addw      #99, d7
```

- $d7 \leftarrow 99 + d7$ ; value 99 comes from the instruction
- d7 is a register

# Register Direct Addressing

- The data is from a register, and the register is directly given by the instruction
- For example, in 68K

```
addw      d0,d7
```

- $d7 \leftarrow d7 + d0$ ; add the value in  $d0$  to the value in  $d7$  and store the result to  $d7$
- $d0$  and  $d7$  are registers

# Memory Direct Addressing

- The data is from memory, and the memory address is directly given by the instruction
- We use notion:  $(addr)$  to represent memory value at address,  $addr$
- For example, in 68K

<b>addw</b>	<b>0x123A, d7</b>
-------------	-------------------

- $d7 \leftarrow d7 + (0x123A);$  add value in memory location  $0x123A$  to register  $d7$

# Memory Register Indirect Addressing

- The data is from memory, and the memory address is given by a register, which is given by the instruction
- For example, in 68K

```
addw      (a0),d7
```

- $d7 \leftarrow d7 + (a0)$ ; add value in memory with the address stored in register a0, to register d7
  - For example, if  $a0 = 100$  and  $(100) = 123$ , then this adds 123 to d7

# Memory Register Indirect Auto-increment

- The data is from memory, and the memory address is given by a register, which is given by the instruction; and the value of the register is automatically increased – to point to the next memory object.
- For example, in 68K

<b>addw</b>	<i>(a0)+,d7</i>
-------------	-----------------

–  $d7 \leftarrow d7 + (a0); a0 \leftarrow a0 + 2$

# Memory Register Indirect Auto-decrement

- The data is from memory, and the memory address is given by a register, which is given by the instruction; but the value of the register is automatically decreased before such an operation.
- For example, in 68K

<b>addw</b>	<b>-<i>(a0)</i>,d7</b>
-------------	------------------------

–  $a0 \leftarrow a0 - 2$ ;  $d7 \leftarrow d7 + (a0)$ ;

# Memory Register Indirect with Displacement

- The data is from memory with the address given by the register plus an offset
  - Used to access a member in a data structure
- For example, in 68K

<b>addw</b>	<b>a0@(8), d7</b>
-------------	-------------------

- $d7 \leftarrow (a0+8) + d7$

# Address Register Indirect with Index and Displacement

- The address of the data is sum of a base address and an index address and an offset
  - Used to access elements in an array
- For example, in 68K

<b>addw</b>	<b>a0@(d3)8, d7</b>
-------------	---------------------

- $d7 \leftarrow (a0 + d3+8) + d7$
- With a0 as an initial address and d3 varied to dynamically point to different elements plus a constant for a certain member of an element of an array.

# Suggested Readings

- Cady “Microcontrollers and Microprocessors”, Chapter 1.1, Chapter 2.2-2.4
- Cady “Microcontrollers and Microprocessors”, Appendix A
- Week 1 reference: “Number Conversion”
  - available on the course website

# Homework

1. Install Atmel Studio at home and complete lab0
  - Available on the **Labs** page on the course website
2. Complete Quiz 1
  - Released after the lecture, due before next lecture
  - Available on the **Activities** page on the course website

# **Microprocessors & Interfacing**

Lecturer : Annie Guo

# Notice

- Before the lecture starts, please
  - Turn off Notifications on your computer
  - Mute your mobile
- During the lecture,
  - If you have any questions,
    - raise your hand to ask, or
    - post your questions in chat (preferred)
  - Set the microphone properly in your MS Teams
    - When you are allowed to speak
      - Unmute the microphone
    - When you are not speaking
      - Mute the microphone

# Lecture Overview

- Course Introduction
  - A whole picture of the course
- Basics of Computing with Microprocessor Systems

# Course Organization

- Lecture:
  - Online (4-6pm, Mon. & Tue.)
  - Three areas about the microprocessor
    - Fundamentals
    - Assembly programming
    - Interfacing
- Lab:
  - F2f & Online
  - Week 1: Set up the simulation environment; form lab groups
- Assignment (design project)
  - Microprocessor application

# Aims of the Course

- After completing the course, you should
  - be able to explain the basic microprocessor architecture and the interface between software and hardware
  - be familiar with assembly programming (based on the AVR microprocessor)
  - be able to explain how the communication between microprocessor and I/O devices works (based on the AVR microcontroller)

# Aims of the Course (cont.)

- After completing the course, you should
  - understand how analog signals are converted into digital signals and vice versa
  - demonstrate the ability to solve various problems with (AVR) microcontroller
  - demonstrate the ability to work in teams for lab tasks and design project

# Strategies (1)

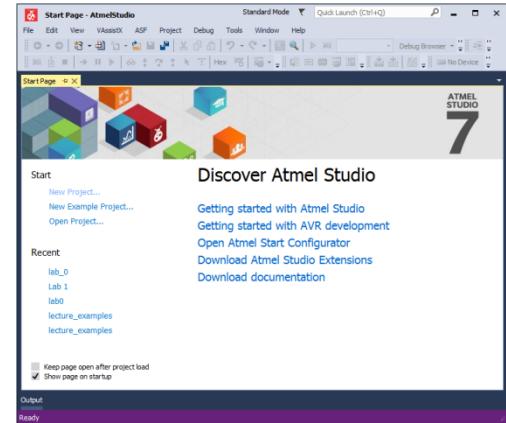
- Lectures
  - Concepts
  - Principles
  - Problem solving approaches and techniques

## Key topics

- Basics of Computing with Microprocessor Systems
- Instruction set architecture
- AVR assembly programming
- Input and Output
- Interrupt
- Analog/digital and digital/analog conversion
- Serial communication

# Strategies (2)

- Labs
  - Lab tools
    - Atmel studio
      - Project development, simulation and debug
    - AVR lab board
      - Devices
      - Programming and testing
  - Lab exercises
    - Prepare before the lab class (strongly suggested)
    - Finish in lab
    - Marked off by the lab tutor
      - Late penalty
        - » 30% per week
        - » Assessment for the late submission will be run after the normal assessment period



# Strategies (3)

- Design project
  - Through a whole design cycle
  - Apply what you have learnt in the course
    - concepts
    - approaches and techniques
  - Collaborate with team members
  - Communicate project work
    - Lab demonstration
    - Written report
      - Late penalty
        - » 10% per day

# Strategies (4)

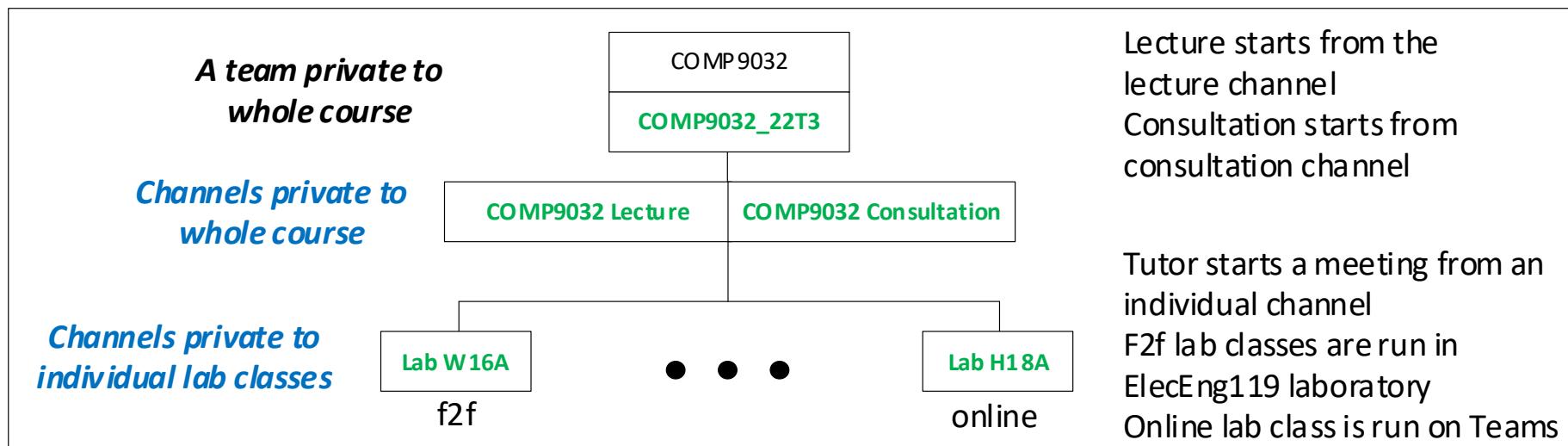
- Homework
  - Readings
  - Questions
    - For quick evaluation and feedback

# How to run lecture/lab online?

- Classes on MS Teams

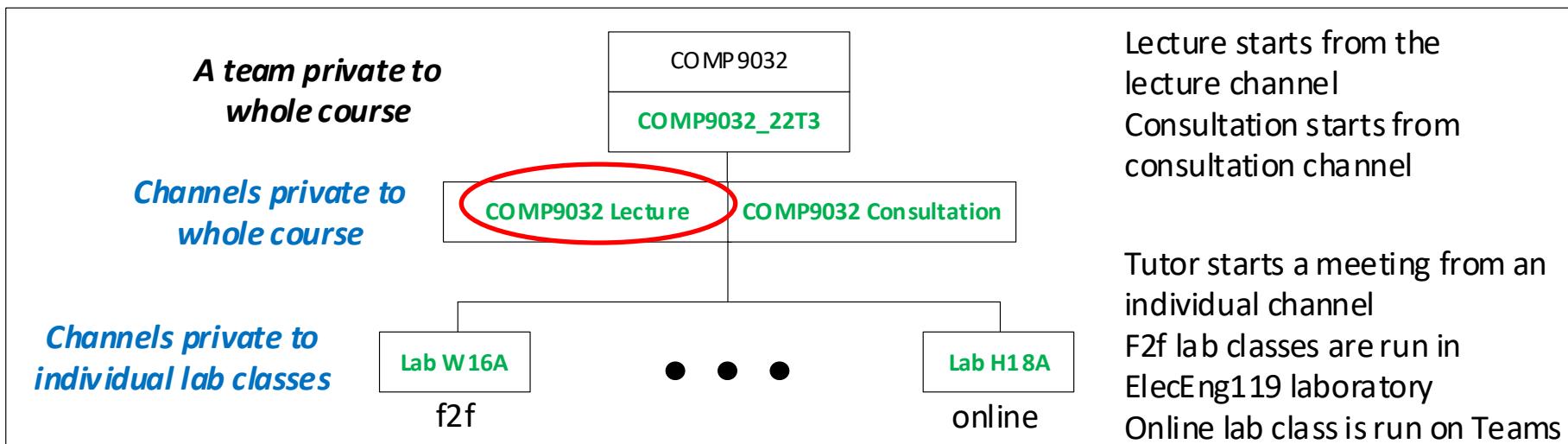
- One team for COMP9032

- Lecture channel
    - Consultation channel
    - Channels for different lab classes



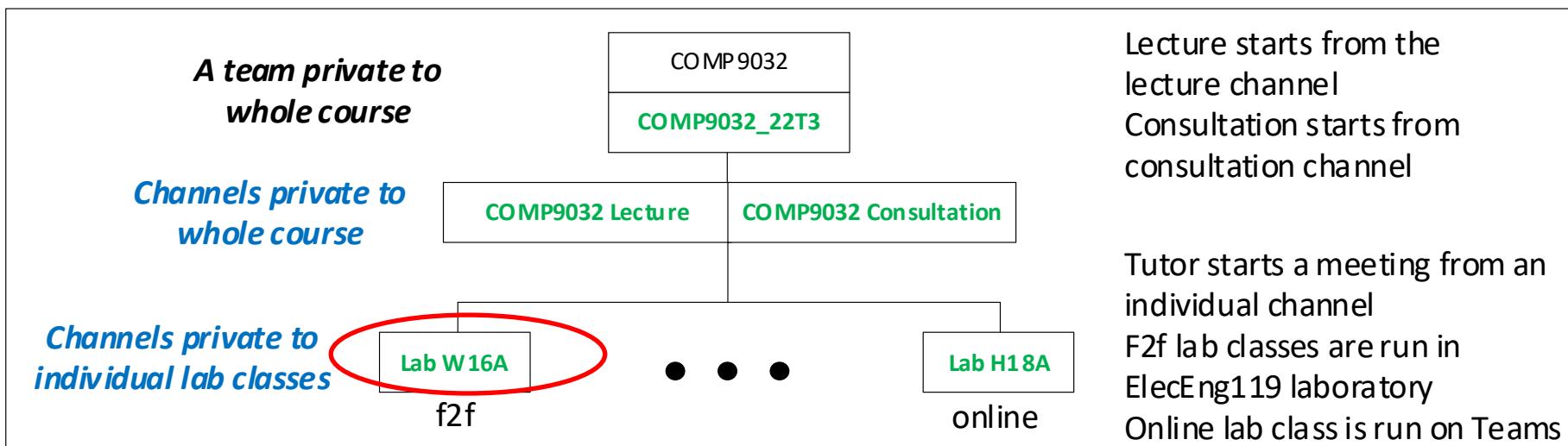
# How to run lecture/lab online?

- For lectures
  - The lecturer will start a meeting **five mins** before the lecture in the Lecture channel
  - You can join the meeting afterwards
    - via a link on the lecture page of the course website or from the MS Teams



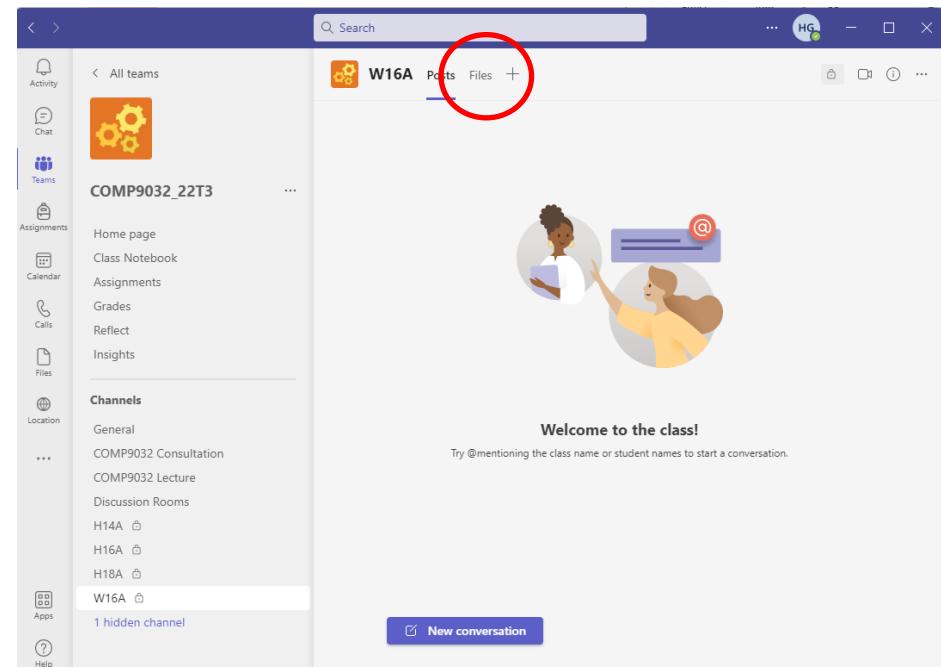
# How to run lecture/lab online?

- Each lab class, including f2f classes, will have an online channel for communication
- For the online lab class
  - The tutor starts a meeting in your lab class channel



# Lab Groups

- You need to form your lab group in Week 1
  - In your lab class
- The student list of your lab class is available in your team channel



# Lab Groups (cont.)

- Each group ideally has five members.
- For the f2f lab classes
  - You can select your members
- For the online lab class
  - Each group should have at least one student who can hold a lab board for the lab group
- We may ask your help to change to or merge with another group if
  - some students drop the course later, or
  - no member in your group is located locally to pick up a lab board for your group

# Assessment

- Lab exercises
  - 25% (of the final result)
  - Carried out either in groups or individually
  - Assessed
    - In the laboratory for the f2f lab classes
    - In the Teams meeting for the online lab class
- Assignment (Project design)
  - 25%
  - Carried out in groups
    - Group demonstration
    - Report
  - But your marks are based on both the group work and the individual performance.
    - Namely, your mark is also determined by your contribution

# Assessment (cont.)

- Final exam
  - 2-hour online exam.
  - 50%
  - In Moodle
- For assessment special considerations, please apply through Student Lifecycle.

# References

- Main references:
  - **Fredrick M. Cady: Microcontrollers and Microcomputers —Principles of Software and Hardware Engineering**
  - **AVR documents (available on the course website)**
    - Data Sheet
    - Instruction Set
  - Additional materials provided on the course website
- Lecture notes
  - Posted before each lecture
- Lecture recordings
  - Available after each lecture
    - MS Teams→COMP9032\_22T3→Lecture Channel→ file folder

# Resources for Help

- Course website
  - [www.cse.unsw.edu.au/~cs9032](http://www.cse.unsw.edu.au/~cs9032)
- Lecturer
  - Consultation
    - Fri. 3:30pm—4:30pm
- Lab tutors
  - Kenny Dow
  - Jing Gong
  - Kisaru Liyanage
- Course forum

# Your tutors

	Kenny	Jing	Kisaru
Wed 16-18		yes	yes
Thu 14-16	yes	yes	
Thu 16-18		yes	yes
Thu 18-20	yes		yes

# **NOTE**

- From time to time, I will post notice on the course website.
- Please check the website frequently for notices, lecture notes, and specifications of lab exercises, and the assignment.