

Introduction to Assembly Language Programming

- Reference Book: Assembly Language Programming and Organization of IBM PC
 - Ytha Yu
 - Charles Marut

Chapter -1

Microcomputer Systems

Component of a Microcomputer Systems

- Composed of Digital circuits
- Mainly consist of 3 parts
 - Central processing unit (CPU)
 - Memory circuits
 - I/O circuits

Memory

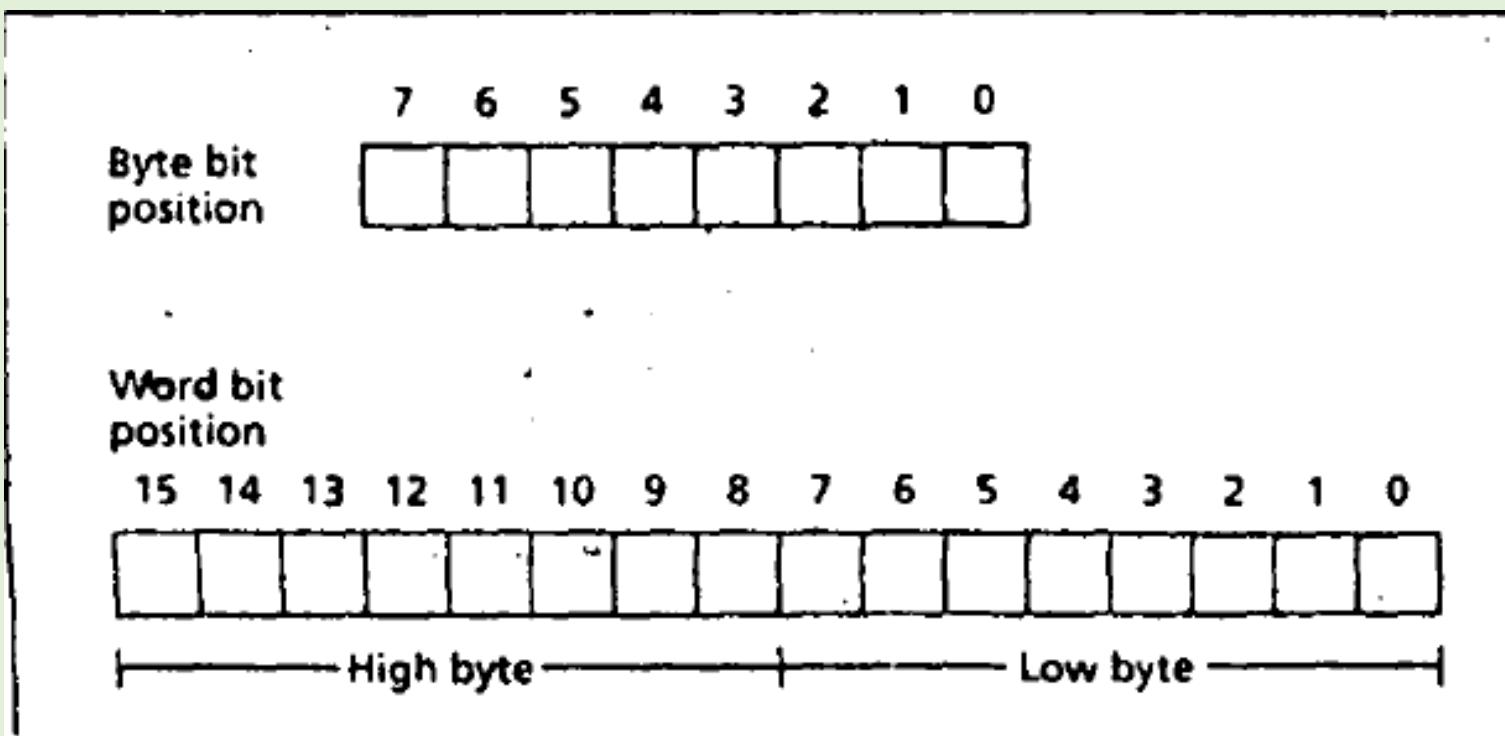
- A memory circuit element can store one bit of data
- Organized into groups of 8 bits = 1 byte
- Each memory byte is identified by a number called **address**
- Data stored in a memory byte is called its **contents**
- Address of a memory byte is fixed
- Content of a memory byte is not fixed and subject to change

- Address	Contents
.	.
.	.
.	.
7	0 0 1 0 1 1 0 1
6	1 1 0 0 1 1 1 0
5	0 0 0 0 1 1 0 1
4	1 1 1 0 1 1 0 1
3	0 0 0 0 0 0 0 0
2	1 1 1 1 1 1 1 1
1	0 1 0 1 1 1 1 0
0	0 1 1 0 0 0 0 1

Memory of Intel 8086 Microprocessor

- 8086 processor uses 20 bits of address
- How many memory bytes can be accessed?
- Answer : 1 MB
- Two bytes form a word
- The lower address of the two memory bytes is used as the address of the memory word.
- Example: So, memory word of address 2 is made up of the memory bytes with the address 2 and 3

Bit Position

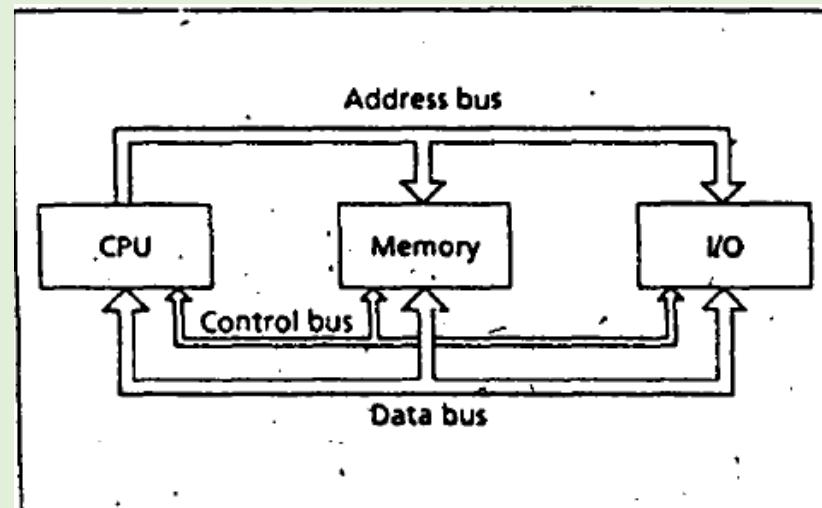


Memory Operations

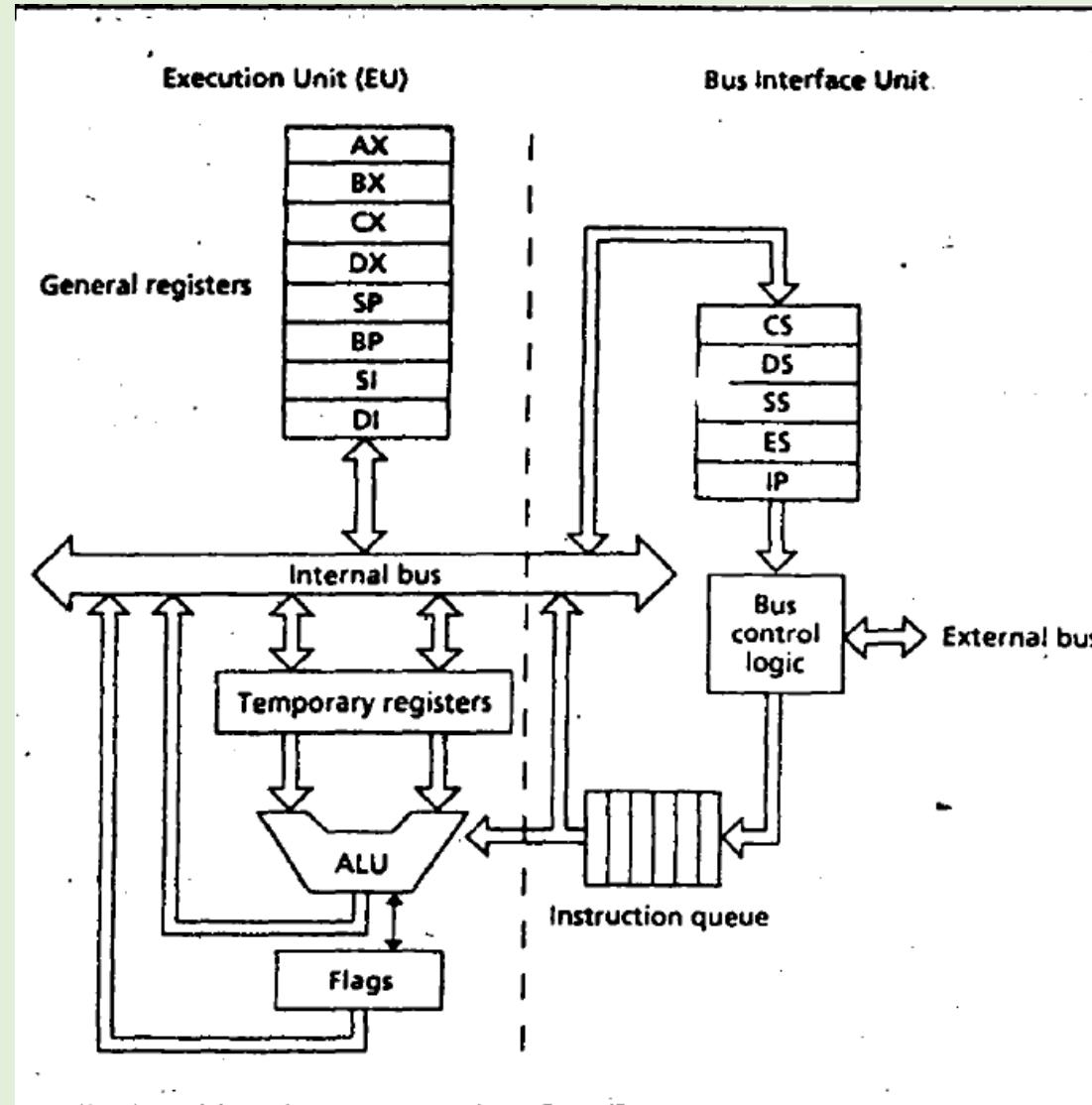
- Read: fetch the contents of a location. Processor only gets a copy of the data.
- Write: the data written become the new contents of the location; original contents are thus lost.

RAM, ROM and Buses

- RAM: Random Access Memory
- ROM: Read Only Memory
- Buses: processor communicates with memory and I/O circuits using signals that travel along a set of wires or connections called buses that connect the different components.
- Three types of buses
 - Address bus
 - Data bus
 - Control Bus



Intel 8086 Microprocessor Organization



EU and BIU

- EU (Execution Unit) :
 - Contains a circuit called ALU
 - Registers : like a memory location except referred with a name rather than number
- BIU (Bus Interface Unit): facilitates communication between the EU and the memory or I/O circuits.
 - Responsible for transmitting address, data and control signals on the buses.

Chapter - 2

Representation of Numbers and Characters

Number Systems

- Decimal
- Binary
- Hexadecimal

Conversion of Number Systems

- *Converting Binary and Hex to Decimal*
- *Converting Decimal to Binary and Hex*
- *Conversions Between Hex and Binary*

Addition and Subtraction

- Binary Addition and Subtraction
- Hexadecimal Addition and Subtraction

Integer Representation in Computer

- LSB (Least Significant Bit)
- MSB (Most Significant Bit)
- Unsigned Integer: represents a magnitude, so it is never negative.
 - Byte : 0-255
 - Word : 0-65535
- Signed Integer: can be positive or negative. The most significant bit is reserved for the sign: 1 means negative and 0 means positive. Negative integers are stored in the computer in a special way known as two's complement.

One's Complement and Two's Complement

- Word Interpretation of integer 5
 - 5: 0000 0000 0000 0101
 - one's complement of 5 = 1111 1111 1111 1010
 - two's complement of 5 = 1111 1111 1111 1011

Decimal Interpretation

- Unsigned Decimal Interpretation: Just do a binary to decimal conversion.
- Signed Decimal Interpretation:
 - If the msb is 0, signed decimal is same as unsigned decimal
 - If the msb is 1, the number is negative, call it $-N$. to find N , just take the two's complement of and convert to decimal as before.

Character Representation

- ASCII Code

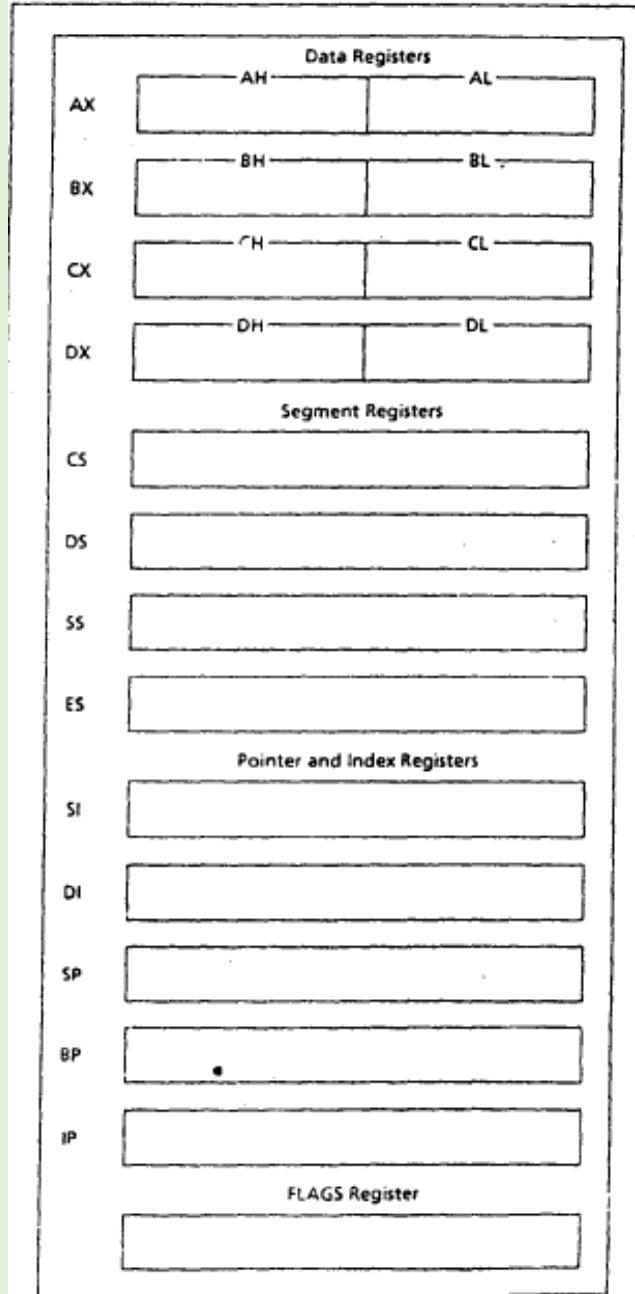
Chapter -3

Organization of the IBM Personal Computers

Registers of 8086 Microprocessors

- The registers are classified according to the functions they perform:
 - Data Registers : hold data for an operation
 - Address Registers: hold the address of instruction or data
 - Status Registers: keeps the current status of the processor
- 8086 has four general data registers
- Address Registers are divided into ***segment, pointer and index registers.***
- Status Register is called the ***Flags*** register
- In total, fourteen ***16 bit*** registers

Registers



Data Registers

- AX (Accumulator Register) : AX is the preferred register to use in arithmetic, logic, and transfer instructions
- BX (Base Register) : BX serves as a base register.
- CX (Count Register): Program loop construction are facilitated by the use of CX. Other operations like shift and rotate also use CX
- DX (Data Register): DX is used in multiplication and division. It is also used in I/O operations.

Segment Registers

- Memory Segment: Memory segment is a block of 2^{16} (or 64 k) of consecutive memory bytes.
- Each segment is identified by a segment number, starting with 0.
- A segment number is 16 bits, so the highest segment number is FFFFh.
- So, a memory location can be specified by providing a segment number and offset in **segment:offset** form which is known as logical address

Mapping from Logical Address to Physical Address

- To obtain the 20 bit address, Shift the segment address by 4 bits and add the offset address to it.

If A4FB:4872 is the address in segment:offset form, it means offset 4872h within segment A4FB h. And the physical address is:

A4FB0h

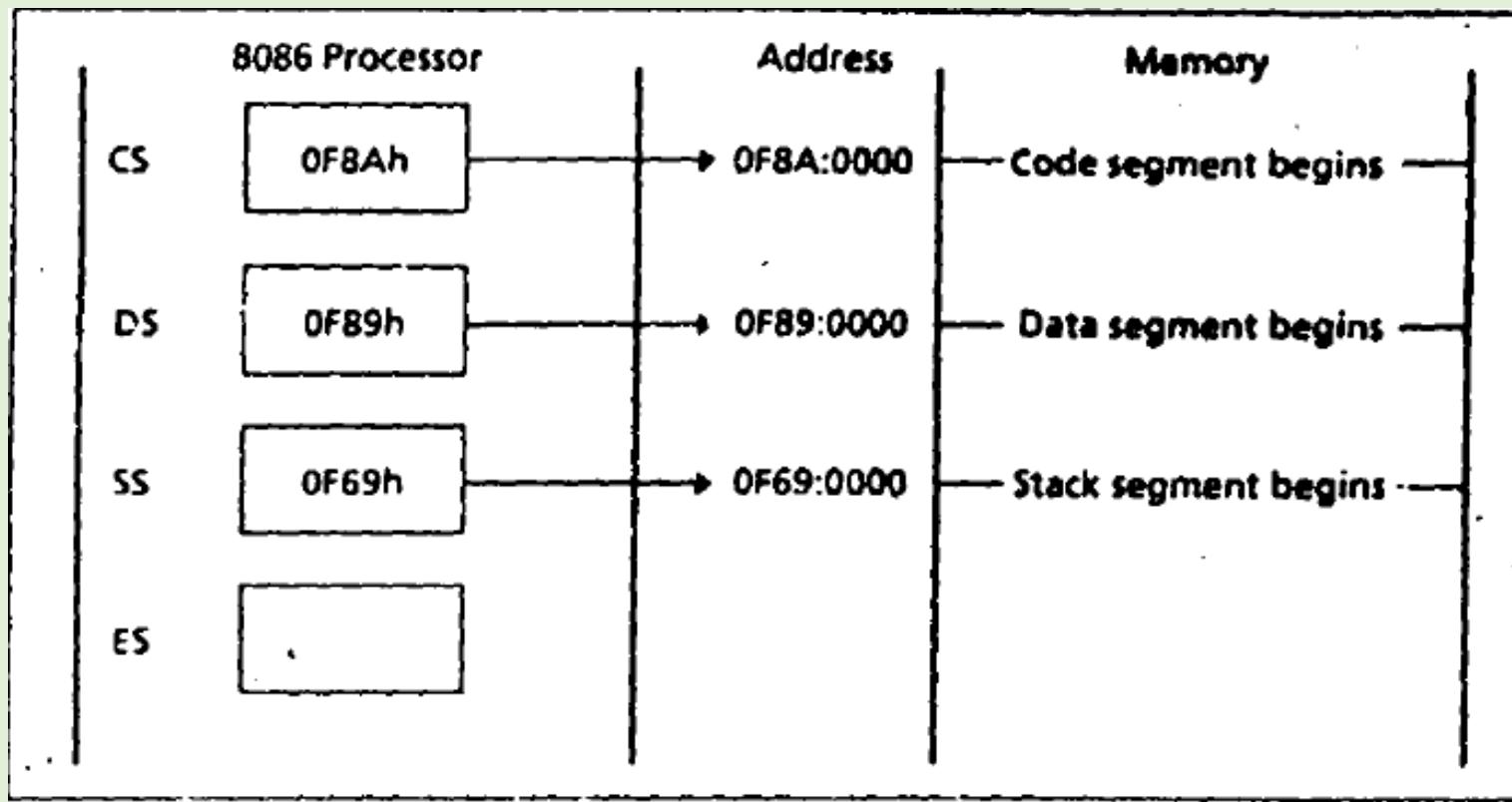
+ 4872h

A9822h

Overlapping of Segment Address

Address		
10021	11010101	
10020	01001001	
Segment 2 ends → 1001F	11110011	
1001E	10011100	
10010	01111001	
Segment 1 ends → 1000F	11101011	
1000E	10011101	
10000	01010001	
Segment 0 ends → 0FFFF	11111110	
0FFE	10011111	
00021	01000000	
Segment 2 begins → 00020	01101010	
0001F	10110101	
00011	01011001	
Segment 1 begins → 00010	11111111	
0000F	10001110	
00003	10101011	
00002	00000010	
- 00001	10101010	
Segment 0 begins → 00000	00111000	

Program Segments



Pointer and Index Registers

- SP (Stack Pointer) : is used in conjunction to SS
- BP (Base Pointer): is used to primarily to access data on the stack. However, unlike SP, we can also use BP to access data in the other segments.
- SI (Source Index) : used to point to memory locations in the data segment addressed by DS.
- DI (Destination Index) : performs the same functions as SI. There is a class of instructions, called string operations that use DI to access memory locations addressed by ES.

IP and Flags Register

- To access Instructions, the 8086 uses the registers CS and IP.
- The CS register contains the segment number of the next instruction, and the IP contains the offset. IP is updated each time an instruction is executed so that it will point to the next Instruction.
- Unlike the other registers, the IP cannot be directly manipulated by an instruction
- Flags Register is used to indicate the status of a microprocessor.

Chapter - 4

Introduction to IBM PC assembly language

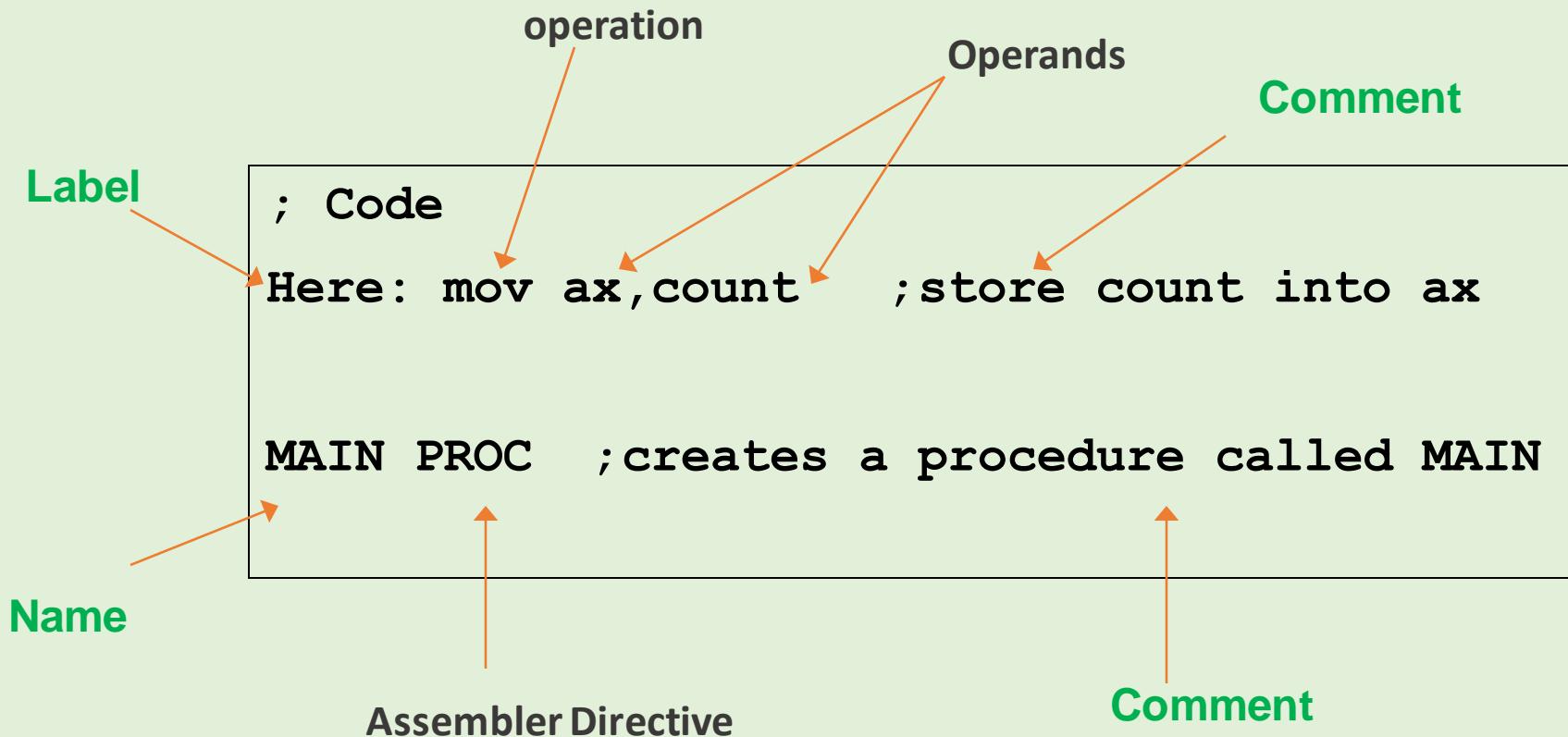
Statements

- Syntax:

name operation operand(s) comments

- name and comment are optional
 - Number of operands depend on the instruction
- One statement per line
 - At least one blank or tab character must separate the fields.
- Each statement is either:
 - Instruction (translated into machine code)
 - Assembler Directive (instructs the assembler to perform some specific task such as allocating memory space for a **variable** or creating a procedure)

Statement Example



Name/Label Field

- The assembler translates names into memory addresses.
- Names can be 1 to 31 character long and may consist of letter, digit or special characters. If period is used, it must be first character.
- Embedded blanks are not allowed.
- May not begin with a digit.
- Not case sensitive

Examples of legal names	Examples of illegal names
COUNTER_1	TWO WORDS
@character	2abc
.TEST	A45.28
DONE?	YOU&ME

Operation Field: Symbolic operation (Op code)

- Symbolic op code translated into Machine Language op code
- ***Examples:*** ADD, MOV, SUB
- In an assembler directive, the operation field represents Pseudo-op code
- Pseudo-op is not translated into Machine Language op code, it only tells assembler to do something.
- ***Example:*** PROC psuedo-op is used to create a procedure

Operand Field

- An instruction may have zero, one or more operands.
- In two-operand instruction, first operand is destination, second operand is source.
- For an assembler directive, operand field represents more information about the directive
- ***Examples***

NOP ;no operand, does nothing

INC AX ;one operand, adds 1 to the contents of AX

ADD AX, 2 ;two operands, adds value 2 to the contents of AX

Comments

- Optional
- Marked by semicolon in the beginning
- Ignored by assembler
- Good practice

Program Data

- Processor operates only on binary data.
- In assembly language, you can express data in:
 - Binary
 - Decimal
 - Hexadecimal
 - Characters
- Numbers
 - For Hexadecimal, the number must begin with a decimal digit. E.g.: write 0ABC not only ABC.
 - Cannot contain any non-digit character. E.g.: 1,234 not allowed
- Characters enclosed in single or double quotes.
 - ASCII codes can be used
 - No difference in “A” and 41h

Contd..

- Use a **radix symbol** (suffix) to select binary, octal, decimal, or hexadecimal

6A15h	; hexadecimal
0BAF1h	; leading zero required
32q	; octal
1011b	; binary
35d	; decimal (default)

Variables

- Each variable has a data type and is assigned a memory address by the program.
- Possible Values:
 - Numeric, String Constant, Constant Expression, ?
 - **8 Bit Number Range:** Signed (-128 to 127), Unsigned (0-255)
 - **16 Bit Number Range:** Signed (-32,678 to 32767), Unsigned (0-65,535)
 - ? To leave variable uninitialized

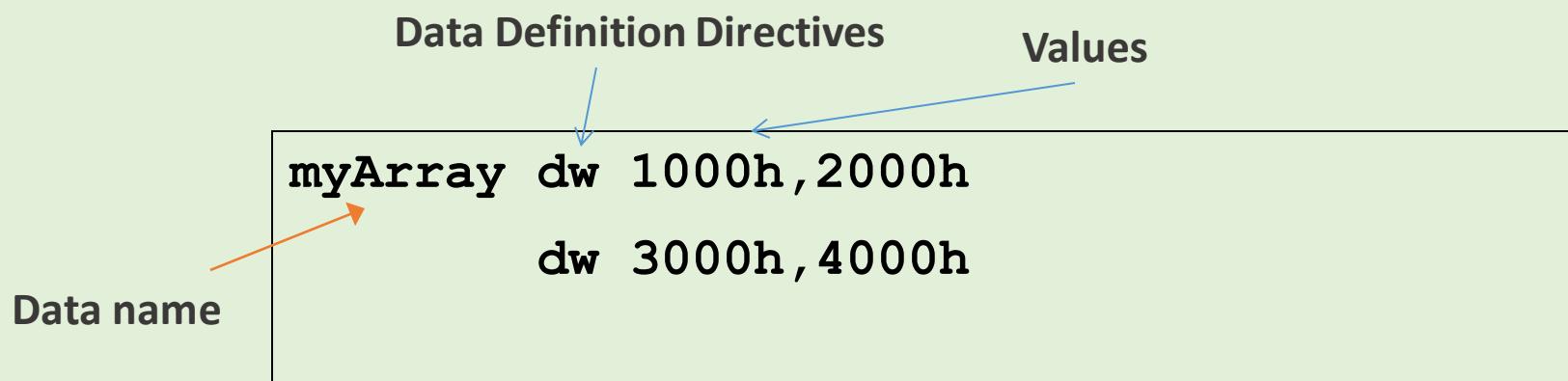
Contd..

- Syntax

```
variable_name type initial_value
```

```
variable_name type value1, value2, value3
```

- Data Definition Directives Or Data Defining Pseudo-ops
 - DB, DW, DD, DQ, DT



Remember: you can skip variable name!

Contd..

Examples	Bytes	Description	Pseudo-ops
var1 DB 'A' Var2 DB ? array1 DB 10, 20,30,40	1	Define Byte	DB
var2 DW 'AB' array2 DW 1000, 2000	2	Define Word	DW
Var3 DD -214743648	4	Define Double Word	DD

Note:

Consider

var2 DW 10h

Still in memory the value saved will be 0010h

Named Constants

- Use symbolic name for a constant quantity
- **Syntax:**

name EQU constant

- **Example:**

LF EQU 0Ah

- No memory allocated

MOV

- Transfer data
 - Between registers
 - Between register and a memory location
 - Move a no. directly to a register or a memory location
- Syntax

MOV *destination, source*

- Example

MOV AX, WORD1

- Difference?

- MOV AH, 'A'
- MOV AX, 'A'

	<i>Before</i>	<i>After</i>
AX	0006	0008
WORD1	0008	0008

Legal Combinations of Operands for MOV

Destination Operand	Source Operand	Legal
General Register	General Register	YES
General Register	Memory Location	YES
General Register	Segment Register	YES
General Register	Constant	YES
Memory Location	General Register	YES
Memory Location	Memory Location	NO
Memory Location	Segment Register	YES
Memory Location	Constant	YES

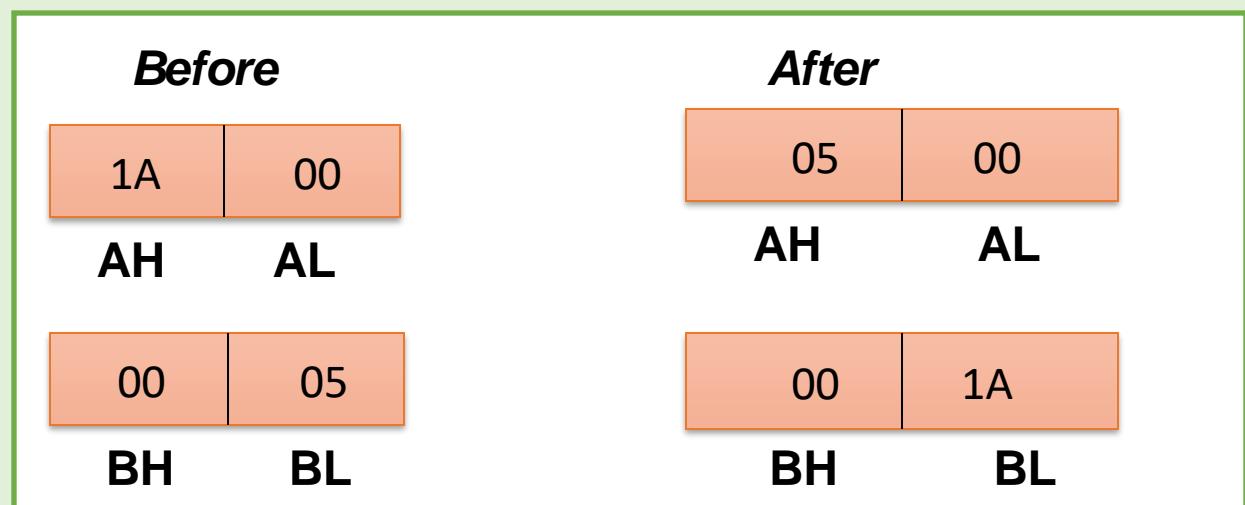
XCHG

- Exchange the contents of
 - Two registers
 - Register and a memory location
- Syntax

XCHG destination, source

- Example

XCHG AH, BL



Legal Combinations of Operands for XCHG

Destination Operand	Source Operand	Legal
General Register	General Register	YES
General Register	Memory Location	YES
Memory Location	General Register	YES
Memory Location	Memory Location	NO

ADD Instruction

- To add contents of:
 - Two registers
 - A register and a memory location
 - A number to a register
 - A number to a memory location
- Example

ADD WORD1, AX

	<i>Before</i>	<i>After</i>
AX	01BC	01BC
WORD1	0523	06DF

SUB Instruction

- To subtract the contents of:
 - Two registers
 - A register and a memory location
 - A number from a register
 - A number from a memory location
- Example

SUB AX, DX

	<i>Before</i>	<i>After</i>
AX	0000	FFFF
DX	0001	0001

Legal Combinations of Operands for ADD & SUB instructions

Destination Operand	Source Operand	Legal
General Register	General Register	YES
General Register	Memory Location	YES
General Register	Constant	YES
Memory Location	General Register	YES
Memory Location	Memory Location	NO
Memory Location	Constant	YES

INC & DEC

- **INC** (increment) instruction is used to add 1 to the contents of a register or memory location.
 - Syntax: INC *destination*
 - Example: INC WORD1
- **DEC** (decrement) instruction is used to subtract 1 from the contents of a register or memory location.
 - Syntax: DEC *destination*
 - Example: DEC BYTE1
- Destination can be 8-bit or 16-bits wide.
- Destination can be a register or a memory location.

Contd..

INC WORD1

	<i>Before</i>	<i>After</i>
WORD1	0002	0003

DEC BYTE1

	<i>Before</i>	<i>After</i>
BYTE1	FFFE	FFFFD

NEG

- Used to negate the contents of destination.
- Replace the contents by its 2's complement.
- Syntax

NEG *destination*

- Example

NEG BX

	<i>Before</i>	<i>After</i>
BX	0002	FFFE

How?

Examples

- Consider instructions: MOV, ADD, SUB, INC, DEC, NEG
- **A** and **B** are two word variables
- Translate statements into assembly language:

Statement	Translation
B = A	MOV AX, A MOV B, AX
A = 5 - A	MOV AX, 5 SUB AX, A MOV A, AX OR NEG A ADD A, 5

Contd..

Statement	Translation
A = B – 2 x A	MOV AX, B SUB AX, A SUB AX, A MOV A, AX

❑ Remember: Solution not unique!

❑ Be careful! Word variable or byte variable?

Program Segments

- Machine Programs consists of
 - Code
 - Data
 - Stack
- Each part occupies a memory segment.
- Same organization is reflected in an assembly language program as **Program Segments**.
- Each program segment is translated into a memory segment by the assembler.

Memory Models

- Determines the size of data and code a program can have.
- Syntax:

.MODEL memory_model

Model	Description
SMALL	code in one segment, data in one segment
MEDIUM	code in more than one segment, data in one segment
COMPACT	code in one segment, data in more than one segment
LARGE	Both code and data in more than one segments No array larger than 64KB
HUGE	Both code and data in more than one segments array may be larger than 64KB

Data Segment

- All variable definitions
- Use **.DATA** directive
- For Example:

```
.DATA  
WORD1 DW 2  
BYTE1 DB 10h
```

Stack Segment

- A block of memory to store stack
- Syntax
 - **.STACK size**
 - Where size is optional and specifies the stack area size in bytes
 - If size is omitted, 1 KB set aside for stack area
 - For example:
.STACK 100h

Code Segment

- Contains a program's instructions
- Syntax

.CODE name

- Where name is optional
- Do not write name when using SMALL as a memory model

Putting it Together!

.MODEL SMALL

.STACK 100h

.DATA

;data definition go here

.CODE

;instructions go here

The INT instruction

- INT interrupt_number
 - We will use INT 21H

How to use INT 21H

- Place function number in AH
- Call INT 21H

It will do the task specified by the function in AH

Function Number	Routine
1	Single key input
2	Single character output
9	Character string output

Function 1

- Single-Key Input
- Input: AH = 1
- Output: AL = ASCII code if character key is pressed
= 0 if non-character key is pressed

MOV AH, 1

INT 21H

Function 2

- Display a character or execute a control flow
- Input: AH = 2
 - DL = ASCII code of the display character or control character
- Output: AL = ASCII code of the display character or control character

MOV AH, 2

MOV DL, 'A'

INT 21H

Function 9

- Display a string
- Input: AH = 9

DX = offset address of the string

The string must end with '\$' character

LEA DX, MSG

MOV AH, 09H

INT 21H

The LEA instruction

- Load Effective Address
- *LEA destination, source*
- Puts a copy of the source offset address into the destination
- *LEA DX, MSG*

Initialize the Data Segment register

MOV AX,@DATA

MOV DS,AX

Function 4CH

Return control to DOS

MOVAH, 4CH

INT 21H

CSE 315

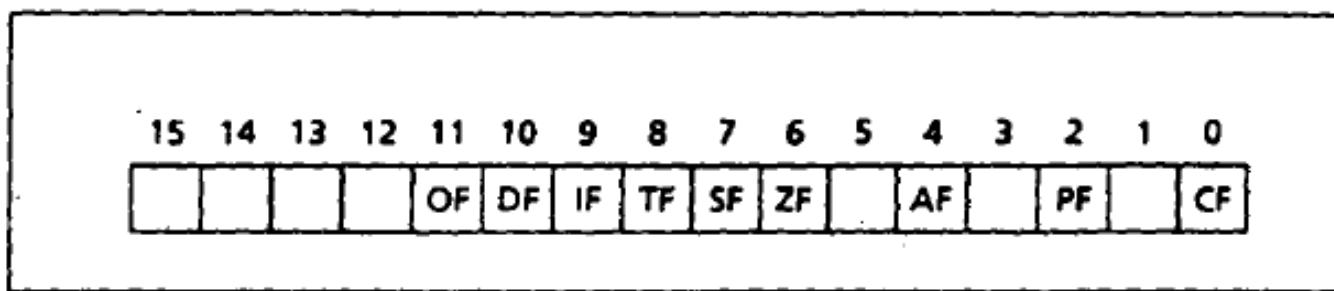
Microprocessors, Microcontrollers, and
Embedded Systems

Assembly Language:
Flow Control

Outline

- The Status Flags
- Control Flow Structure
 - Conditional Jump
 - Unconditional Jump
- Control Flow Structures
 - IF-THEN
 - IF-THEN-ELSE
 - CASE
- Branches with Compound Conditions
- Looping

The FLAGS Register



The FLAGS Register flags

Flag	Intel Mnemonic
Overflow	OF
Sign	SF
Zero	ZF
Auxiliary Carry	AF
Parity	PF
Carry	CF

Overflow Flag

- Unsigned Overflow (CF)

$$\begin{array}{r} \text{1111 1111 1111 1111} \\ + \text{0000 0000 0000 0001} \\ \hline \text{1 0000 0000 0000 0000} \end{array}$$

Overflow Flag

- Signed Overflow (OF)

$$\begin{array}{r} 0111\ 1111\ 1111\ 1111 \\ + 0111\ 1111\ 1111\ 1111 \\ \hline 1111\ 1111\ 1111\ 1110 = \text{FFFFh} \end{array}$$

OF & CF

- OF = 1 for signed overflow, CF =1 for unsigned overflow.
- CF = 1, if there is a carry out of the msb on addition. However, CF is not affected by INC or DEC.
- OF = 1, if we add two numbers of the same sign and the result is of different sign. If we add two numbers of the different sign, there is no way of overflow.
 - If the carry into and out of msb don't match OF = 1

ZF, SF and PF

- ZF = 1 if the result is zero.
- SF = 1, if the msb of result is 1.
- PF = 1, if the *low byte* of the result has an even number of 1s (even parity).

Effect of the flags

Example 5.3 SUB AX,BX, where AX contains 8000h and BX contains 0001h.

Solution:

$$\begin{array}{r} 8000h \\ - 0001h \\ \hline 7FFFh = 0111\ 1111\ 1111\ 1111 \end{array}$$

The result stored in AX is 7FFFh.

SF = 0 because the msb is 0.

PF = 1 because there are 8 (even number) one bits in the low byte of the result.

ZF = 0 because the result is nonzero.

CF = 0 because a smaller unsigned number is being subtracted from a larger one.

Now for OF. In a signed sense, we are subtracting a positive number from a negative one, which is like adding two negatives. Because the result is positive (the wrong sign), OF = 1.

Outline

- The Status Flags
- Control Flow Structure
 - Conditional Jump
 - Unconditional Jump
- Control Flow Structures
 - IF-THEN
 - IF-THEN-ELSE
 - CASE
- Branches with Compound Conditions
- Looping

An Example of Jump

Display the entire IBM character set

```
.MODEL SMALL
.CODE
.STARTUP
    MOV AH, 2          ; display char function
    MOV CX, 256        ; no. of chars to display
    MOV DL, 0          ; DL has ASCII code for null char
PRINT_LOOP:
    INT 21H            ; display a char
    INC DL              ; increment ASCII code
    DEC CX              ; decrement counter
    JNZ PRINT_LOOP      ; keep going if CX not 0
.EXIT
END
```

Section 6-1 of Assembly Language Programming Book

Conditional Jumps

- JNZ is an example of conditional jump instruction
 - Checks the Z flag. If Z = 0 then jump to the location
- Three categories of conditional jumps
 - Signed jumps, for signed interpretation
 - Unsigned jumps, for unsigned interpretation
 - Single-flag jumps, operates on settings of individual flags

The CMP Instruction

- The jump condition is often preceded by the CMP (compare) instruction

CMP destination, source

- It is like SUB, except that destination is not changed
- Destination may not be a constant
- The result is not stored but the flags are affected

CMP AX, BX
JG BELOW

If AX = 7FFFh, and BX = 0001h, the result is 7FFFh - 0001h = 7FFEh.

ZF = 0, SF = OF = 0, JG (jump if greater) is satisfied,
so control transfers to label BELOW

Signed Conditional Jumps

Symbol	Description	Condition for jumps
JG/JNLE	Jump if greater than Jump if not less than or equal to	ZF = 0 and SF = OF
JGE/JNL	Jump if greater than or equal to Jump if not less than	SF = OF
JL/JNGE	Jump if less than Jump if not greater than or equal to	SF <> OF
JLE/JNG	Jump if less than or equal Jump if not greater than	ZF = 1 or SF <> OF

Unsigned Conditional Jumps

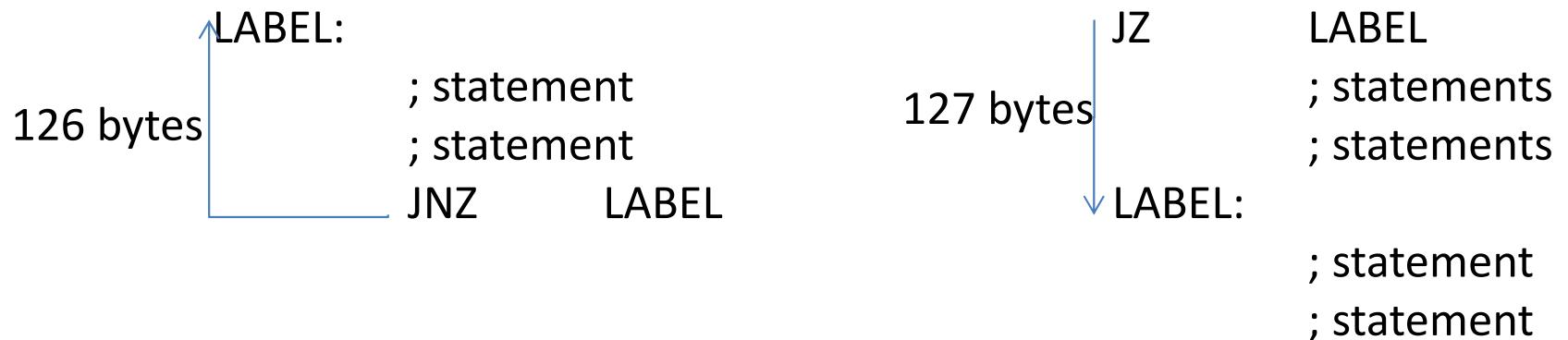
Symbol	Description	Condition for jumps
JA/JNBE	Jump if above Jump if not below or equal	CF = 0 and ZF = 0
JAE/JNB	Jump if above or equal Jump if not below	CF = 0
JB/JNAE	Jump if below Jump if not above or equal	CF = 1
JBE/JNA	Jump if below or equal Jump if not above	CF = 1 or ZF = 1

Single-Flag Jumps

Symbol	Description	Condition for jumps
JE/JZ	Jump if equal Jump if equal to zero	ZF = 1
JNE/JNZ	Jump if not equal Jump if not zero	ZF = 0
JC	Jump if carry	CF = 1
JNC	Jump if no carry	CF = 0
JO	Jump if overflow	OF = 1
JNO	Jump if no overflow	OF = 0
JS	Jump if sign negative	SF = 1
JNS	Jump if nonnegative sign	SF = 0
JP/JPE	Jump if parity even	PF = 1
JNP/JPO	Jump if parity odd	PF = 0

Range of a Conditional Jump

- The destination label must precede the jump instruction by no more than 126 bytes
- Or, follow by no more than 127 bytes



Signed vs. Unsigned Jumps

- Each signed jump corresponds to an analogous unsigned jump
 - e.g. signed JG corresponds to unsigned JA
 - Use depends on the interpretation
- The jumps operate on different flags

Symbol	Description	Condition for jumps
JG/JNLE	Jump if greater than Jump if not less than or equal to	ZF = 0 and SF = OF
JA/JNBE	Jump if above Jump if not below or equal	CF = 0 and ZF = 0

Signed vs. Unsigned Jumps cont.

- For signed interpretation, let us take
 - AX = 7FFFh, BX = 8000h and we execute

CMP	AX, BX
JA	BELOW

- Then, even though 7FFFh > 8000h in a signed sense, the program does not jump to BELOW
- Because 7FFFh < 8000h in an unsigned sense
- We used JA, which is the unsigned jump

The JMP Instruction

- JMP (jump) instruction causes an unconditional jump
- The syntax is: `JMP destination`
- JMP can be used to get around the range restriction

TOP:

```
; body of the loop, say 2 instructions  
DEC    CX      ; decrement counter  
JNZ    TOP     ; keep looping if CX > 0  
MOV    AX, BX
```

TOP:

```
; body of the loop contains many instructions  
DEC    CX  
JNZ    BOTTOM  
JMP    EXIT  
  
BOTTOM:  
      JMP    TOP  
  
EXIT:  
      MOV    AX, BX
```

Outline

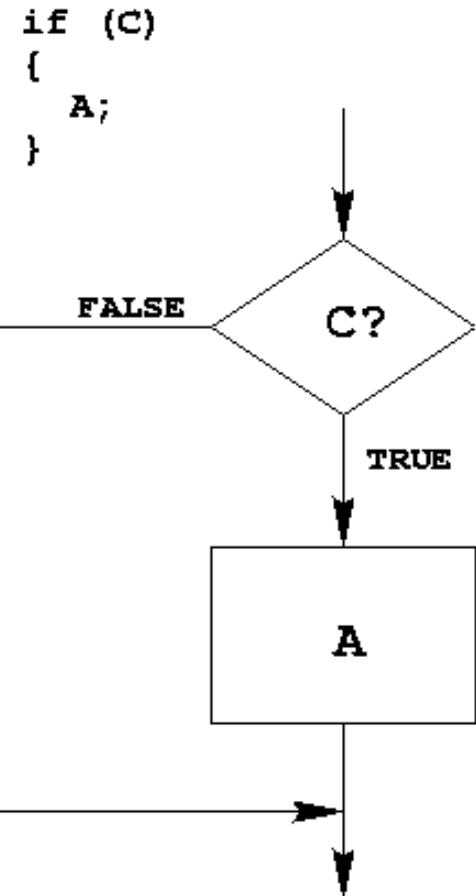
- Control Flow Structure
 - Conditional Jump
 - Unconditional Jump
- Control Flow Structures
 - IF-THEN
 - IF-THEN-ELSE
 - CASE
- Branches with Compound Conditions
- Looping

IF-THEN Structure

Replace the number in AX by its absolute value.

```
IF AX < 0 THEN  
    replace AX by -AX  
END_IF
```

```
CMP     AX, 0      ; AX < 0?  
JNL     END_IF  
NEG     AX  
END_IF:
```



Example 6-2: Assembly Language Programming

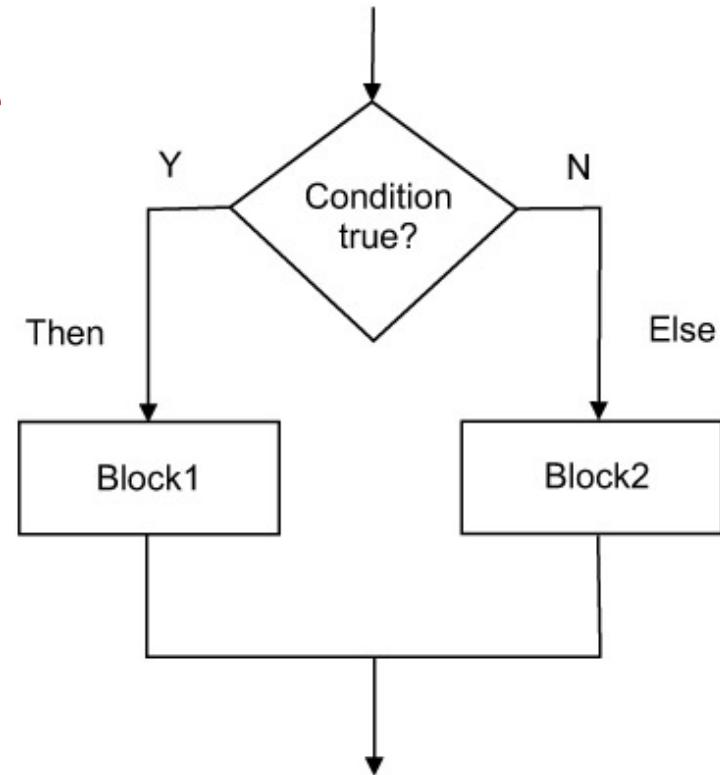
IF-THEN-ELSE Structure

Suppose AL and BL contains ASCII characters.

Display the one that comes first in the character sequence

```
IF AL <= BL THEN
    display the character in AL
ELSE
    display the character in BL
END_ID
```

```
MOV     AH, 2      ; prepare to display
CMP     AL, BL      ; AL <= BL?
JNBE   ELSE_
        MOV     DL, AL
        JMP     DISPLAY
ELSE_:
        MOV     DL, BL
DISPLAY:
        INT     21h
END_IF:
```



Example 6-3: Assembly Language Programming

CASE

- A CASE is a multi-way branch structure

CASE expression

1: statements_1

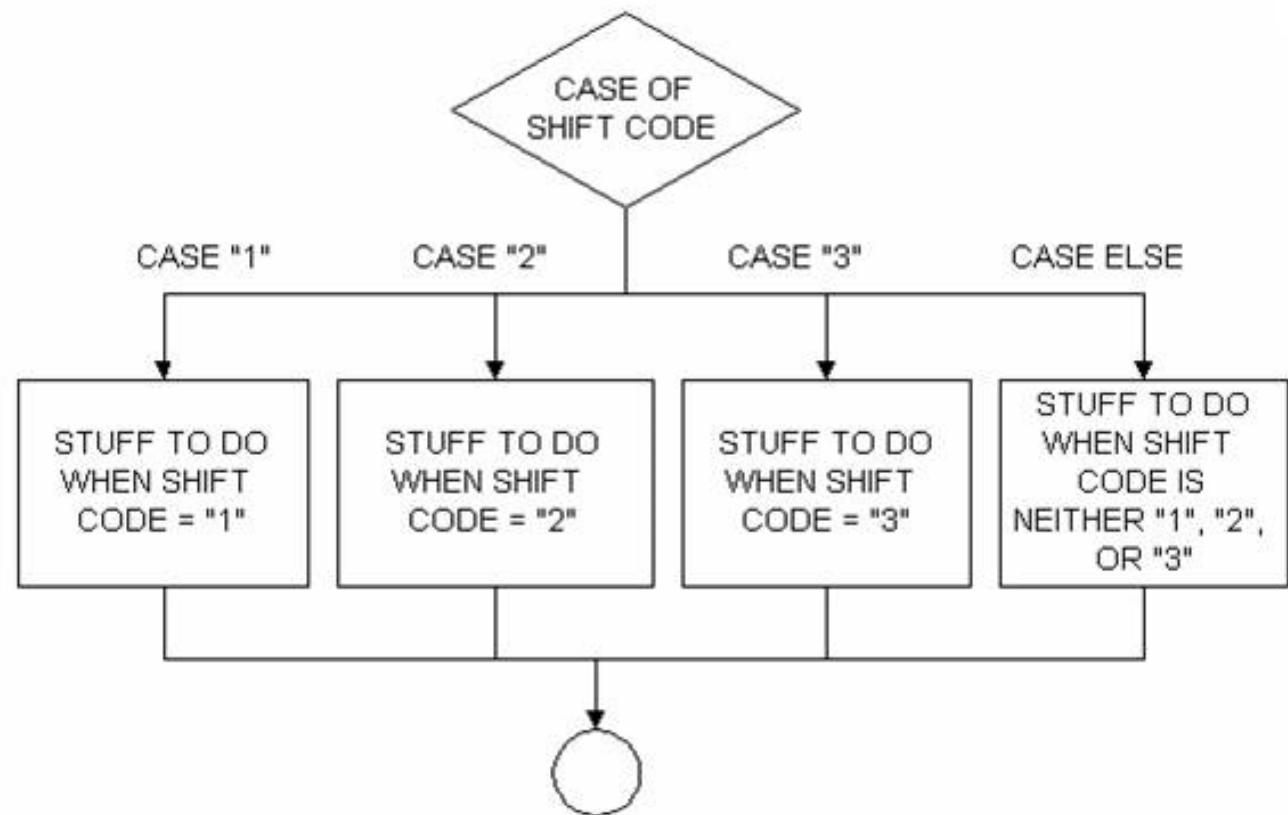
2: statements_2

*

*

n: statements_n

END_CASE



CASE Example

If AX contains a negative number, put -1 in BX;
If AX contains 0, put 0 in BX;
If AX contains a positive number, put 1 in BX.

CMP	AX, 0	; test AX
JL	NEGATIVE	; AX < 0
JE	ZERO	; AX = 0
JG	POSITIVE	; AX > 0

NEGATIVE:

MOV	BX, -1	; put -1 in BX
JMP	END_CASE	; and exit

ZERO:

MOV	BX, 0	; put 0 in BX
JMP	END_CASE	; and exit

POSITIVE:

MOV	BX, 1	; put 1 in BX
-----	-------	---------------

END_CASE:

CASE	AX
< 0:	put -1 in BX
= 0:	put 0 in BX
> 0:	put 1 in BX
END_CASE	

Example 6-4: Assembly Language Programming

More CASE Example

If AL contains 1 or 3, display “o” for odd;

If AL contains 2 or 4, display “e” for even;

```
CMP    AL, 1      ; AL = 1?  
JE     ODD       ; yes, display 'o'  
CMP    AL, 3      ; AL = 3?  
JE     ODD       ; yes, display 'o'  
CMP    AL, 2      ; AL = 2?  
JE     EVEN      ; yes, display 'e'  
CMP    AL, 4      ; AL = 4?  
JE     EVEN      ; yes, display 'e'  
JMP    END_CASE  
  
ODD:  
    MOV    DL, 'o'   ; get 'o'  
    JMP    DISPLAY   ; go to display  
  
EVEN:  
    MOV    DL, 'e'   ; get 'e'  
  
DISPLAY:  
    MOV    AH, 2     ; char display function  
    INT    21h       ; display character  
  
END_CASE
```

```
CASE AL  
    1, 3: display 'o'  
    2, 4: display 'e'  
END_CASE
```

Example 6-4: Assembly Language Programming

Outline

- Control Flow Structure
 - Conditional Jump
 - Unconditional Jump
- Control Flow Structures
 - IF-THEN
 - IF-THEN-ELSE
 - CASE
- Branches with Compound Conditions
- Looping

Branches with Compound Conditions

- Branching condition in an IF or CASE can be

condition_1	AND	condition_2
condition_1	OR	condition_2

- First one is **AND condition**
- Second one is **OR condition**

AND Conditions

Read a character, and if it's an uppercase letter, display it.

Read a character into AL

```
IF ('A' <= character ) and (character <= 'Z') THEN  
    display the character  
END_IF
```

```
MOV     AH, 1      ; read character function  
INT     21h        ; char in AL
```

```
CMP     AL, 'A'    ; char >= 'A'  
JNGE   END_IF    ; no, exit  
CMP     AL, 'Z'    ; char <= 'Z'  
JNLE   END_IF    ; no, exit
```

```
MOV     DL, AL    ; get char  
MOV     AH, 2      ; display character function  
INT     21h        ; display the character
```

END_IF:

Example 6-6: Assembly Language Programming

OR Conditions

Read a character, and if it's 'y' or 'Y', display it; otherwise, terminate the program

```
Read a character into AL  
IF (character = 'y') or (character = 'Y') THEN  
    display the character  
ELSE  
    terminate the program  
END_IF
```

```
MOV     AH, 1      ; read character function  
INT     21h        ; char in AL  
  
CMP     AL, 'Y'    ; char = 'Y'  
JE      THEN       ; yes, display the char  
CMP     AL, 'y'    ; char = 'y'  
JE      THEN       ; yes, display the char  
JMP     ELSE_  
  
THEN:  
    MOV     DL, AL    ; get the char  
    MOV     AH, 2      ; display character function  
    INT     21h        ; display the character  
  
ELSE_:
```

Example 6-7: Assembly Language Programming

Looping

Outline

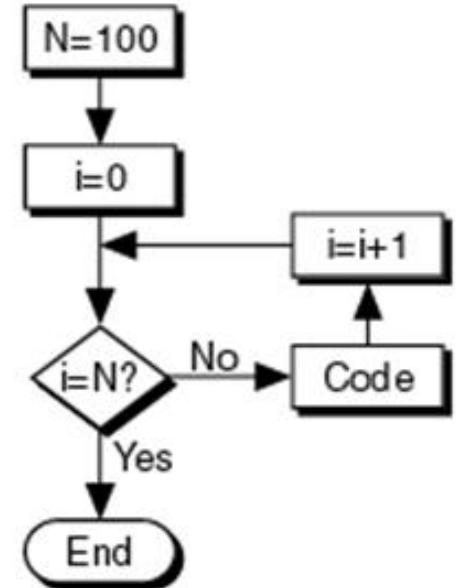
- Control Flow Structures
 - FOR Loop
 - WHILE Loop
 - REPEAT-UNTIL Loop
- Programming with Higher Level Structures

FOR Loop

Write a program to display a row of 80 stars '*'

```
FOR 80 times DO  
    display '*'  
END_FOR
```

```
MOV CX, 80 ; number of '*' to display  
MOV AH, 2 ; char display function  
MOV DL, '*' ; char to display  
TOP:  
    INT 21h ; display a star  
    LOOP TOP ; repeat 80 times
```



Example 6-8: Assembly Language Programming

Caution!

- ‘For loop’ implemented using LOOP is executed at least once
- If CX contains 0 when loop is entered, the LOOP instruction will decrement CX to FFFFh
- The loop will be executed 65535 more times
- JCXZ (jump if CX is zero) may be used

```
        JCXZ      SKIP
TOP:          ; body of the loop
              LOOP      TOP
SKIP:
```

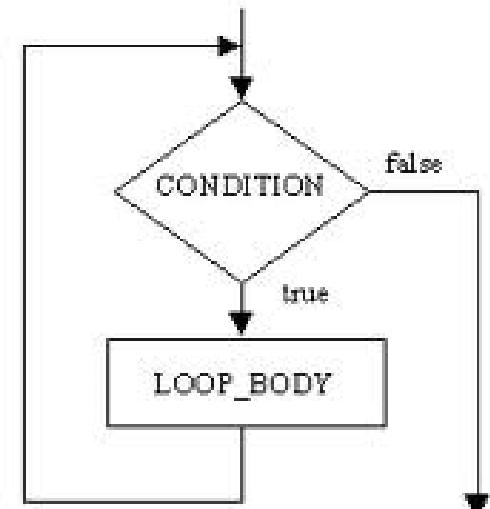
WHILE Loop

Write a program to count the characters in an input line

```
Initialize count to 0  
Read a character  
WHILE character <> carriage_return DO  
    count = count + 1  
    read a character  
END WHILE
```

```
WHILE condition DO  
    statements  
END WHILE
```

```
MOV DX, 0      ; DX counts the characters  
MOV AH, 1      ; read char function  
INT 21h        ; read a char in AL  
  
WHILE_  
    CMP AL, 0DH ; CR?  
    JE END WHILE  
    INC DX  
    INT 21h  
    JMP WHILE_  
  
END WHILE:
```



Example 6-9: Assembly Language Programming

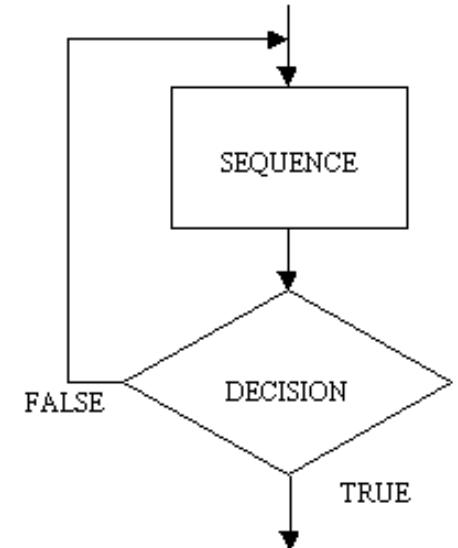
REPEAT Loop

Write a program to read characters until a blank/space is read

```
REPEAT  
    read a character  
UNTIL character is a blank
```

```
MOV     AH, 1      ; read char function  
REPEAT:  
    INT    21h      ; read a char in AL  
    CMP    AL, ' '  ; a blank?  
    JNE    REPEAT   ; no, keep reading
```

```
REPEAT  
    statements  
UNTIL condition
```



Example 6-10: Assembly Language Programming

Outline

- Control Flow Structures
 - IF-THEN
 - IF-THEN-ELSE
 - CASE
 - FOR Loop
 - WHILE Loop
 - REPEAT-UNTIL Loop
- Programming with Higher Level Structures

Programming with High Level Structures

- Problem
 - Prompt the user to enter a line of text. On the next line, display the capital letter entered that comes first alphabetically and the one that comes last. If no capital entered, display “No capital letters”.

Type a line of text:

THE QUICK **BROWN FOX** JUMPED

First capital = B Last capital = X

Top-down Program Design

- Divide the problem into sub-problems
 1. Display the opening message
 2. Read and process a line of text
 3. Display the results

Start the Program

Type a line of text:

THE QUICK **B**ROWN FOX JUMPED

First capital = B Last capital = X

```
.MODEL SMALL
.STACK 100H
.DATA
PROMPT      DB      'Type a line of text', 0DH, 0AH, '$'
NOCAP_MSG   DB      0DH, 0AH, 'No capitals $'
CAP_MSG     DB      0DH, 0AH, 'First capital = '
FIRST       DB      ']' ← Follows 'Z' in ASCII sequence
              DB      ' Last capital = '
LAST        DB      '@ $'
.CODE
.STARTUP
```

Precedes 'A' in ASCII sequence

@ABCDE.....XYZ]
LAST FIRST

Step 1. Display the opening message

```
.DATA  
PROMPT      DB      'Type a line of text', 0DH, 0AH, '$'
```

```
; initialize DS  
MOV     AX, @DATA  
MOV     DS, AX  
; display opening message  
MOV     AH, 9          ; display string function  
LEA     DX, PROMPT    ; get opening message  
INT     21h            ; display it
```

Step 2: Read and Process a Line of Text

```
Read a character
WHILE character is not carriage return DO
    IF character is a capital letter (*) THEN
        IF character precedes first capital THEN
            first capital = character
        END_IF
        IF character follows last capital THEN
            last capital = character
        END_IF
    END_IF
    Read a character
END WHILE
```

Line (*) is actually an AND condition:
IF ('A' <= character) AND (character <= 'Z')

Step 2: Read and Process a Line of Text

```
Read a character
WHILE character is not carriage return DO
    IF character is a capital letter (*) THEN
        IF character precedes first capital THEN
            first capital = character
        END_IF
        IF character follows last capital THEN
            last capital = character
        END_IF
    END_IF
    Read a character
END WHILE
```

Line (*) is actually an AND condition:
IF ('A' <= character) AND (character <= 'Z')

@ABCDE.....XYZ]
LAST FIRST

```
MOV      AH, 1
INT      21h

WHILE_:
        CMP      AL, 0DH
        JE       END_WHILE
        CMP      AL, 'A'
        JNGE    END_IF
        CMP      AL, 'Z'
        JNLE    END_IF
        CMP      AL, FIRST      ; char < FIRST or ']'
        JNL     CHECK_LAST
        MOV      FIRST, AL

CHECK_LAST:
        CMP      AL, LAST      ; char > LAST or '@'
        JNG    END_IF
        MOV      LAST, AL

END_IF:
        INT 21H
        JMP      WHILE_

END WHILE:
```

Step 3: Display The Results

```
IF no capitals were typed THEN  
    display "no capitals"  
ELSE  
    display first capital and last capital  
END_ID
```

```
MOV     AH, 9      ; display string function  
CMP     FIRST, ']'  
JNE     CAPS      ; no, display results  
LEA     DX, NOCAP_MSG  
JMP     DISPLAY  
  
CAPS:  
    LEA     DX, CAP_MSG  
  
DISPLAY:  
    INT     21H  
  
.EXIT  
END
```

@ABCDE.....XYZ]
LAST FIRST

References

- Ch 5, 6 Assembly Language Programming – by Yu and Marut

CSE 315

Microprocessors, Microcontrollers, and
Embedded Systems

Assembly Language:
Arithmetic and logic instructions

AND, OR AND XOR

- Syntax
 - AND destination, source
 - OR destination, source
 - XOR destination, source
- The restrictions of destination and source are the same as ADD or SUB instructions

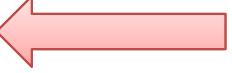
AND, OR AND XOR

- Effect on flags
 - ✓ SF, ZF, PF reflect the result
 - ✓ AF is undefined
 - ✓ CF, OF = 0

AND, OR AND XOR

- Example

✓ XOR AX, AX  Clearing a Register

✓ OR AL, 81h  Set the MSB and LSB while preserving other bits

✓ AND AL, 7Fh  Clear the Sign Bit

NOT Instruction

- Works on a single operand
- Performs one's complement operation on the destination
- Syntax

NOT destination

TEST Instruction

- Performs similarly to the AND instruction
- Except doesn't write the output on the destination.
- Only sets or resets the flags
- Syntax

TEST destination, source

- Usually used for flow control

TEST Instruction

- Example

✓ TEST AL,1  Testing a number is even or not

Shift / Rotate Instructions

- Has two possible formats:

Opcode destination, 1

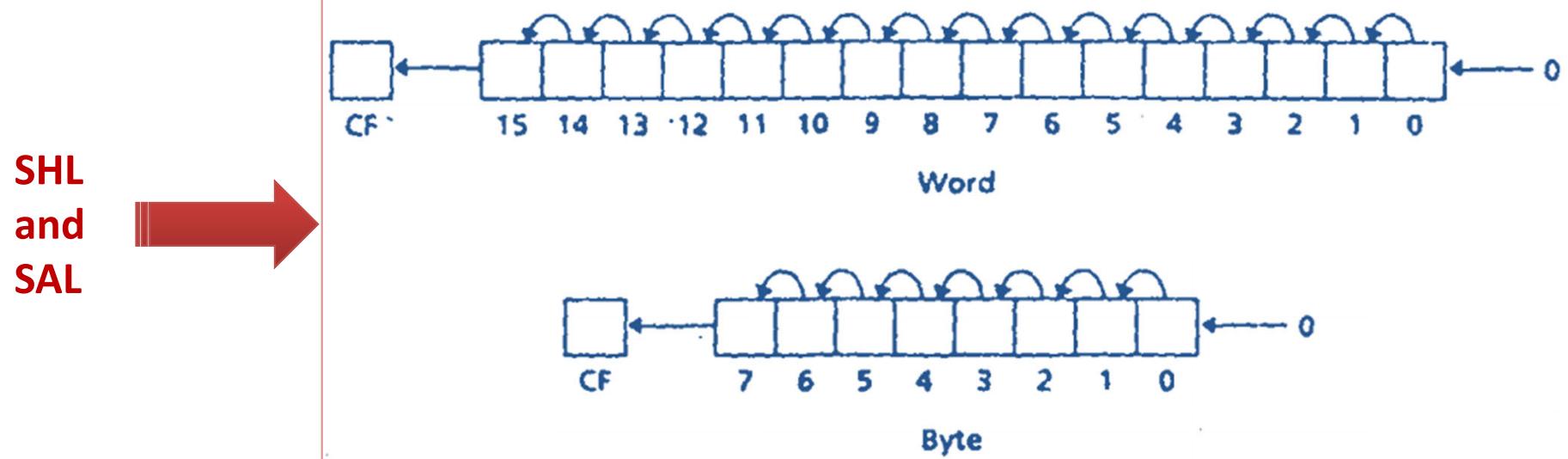
Opcode destination, CL

- Effect on flags

- ✓ SF, ZF, PF reflect the result
- ✓ AF is undefined
- ✓ CF value changes according to Shift / Rotate Type
- ✓ OF =1 if result changes sign on last Shift / Rotation

SHL and SAL instructions

- Shifts the bits in the destination to the left
- The MSB is shifted into CF
- A 0 is shifted to LSB
- SAL and SHL generate the same machine code



SHL and SAL instructions

- Example

- ✓ If DH=8AH, CL=3

- What is the result of
SHL DH, CL



- DH=50H
CF=0
OF=0

- ✓ If DX=8AH, CX=3

- What is the result of
SHL DX, CX



- DX=450H
CF=0
OF=0

Multiplication Using SHL and SAL instructions

- A Left shift on a binary number multiplies it by 2
- Example
 - ✓ If DH=2H, CL=1



DH=4H
CF=0
OF=0

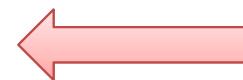
What is the result of

SHL DH, CL

Overflow Flag Untrustworthy During Multiplication using SHL or SAL

- Example

- ✓ If DH=80H, CL=2



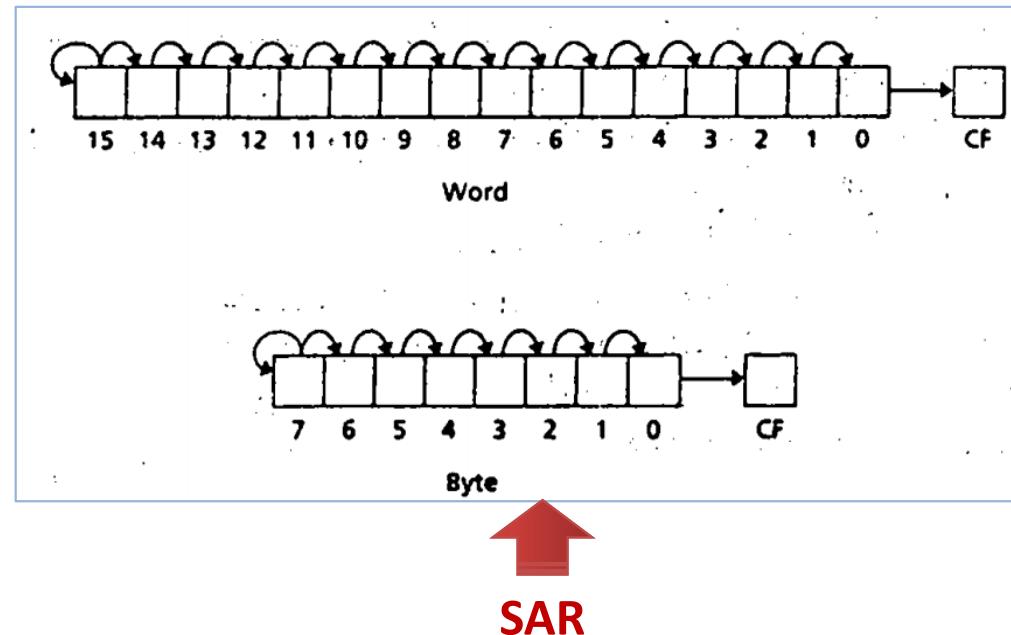
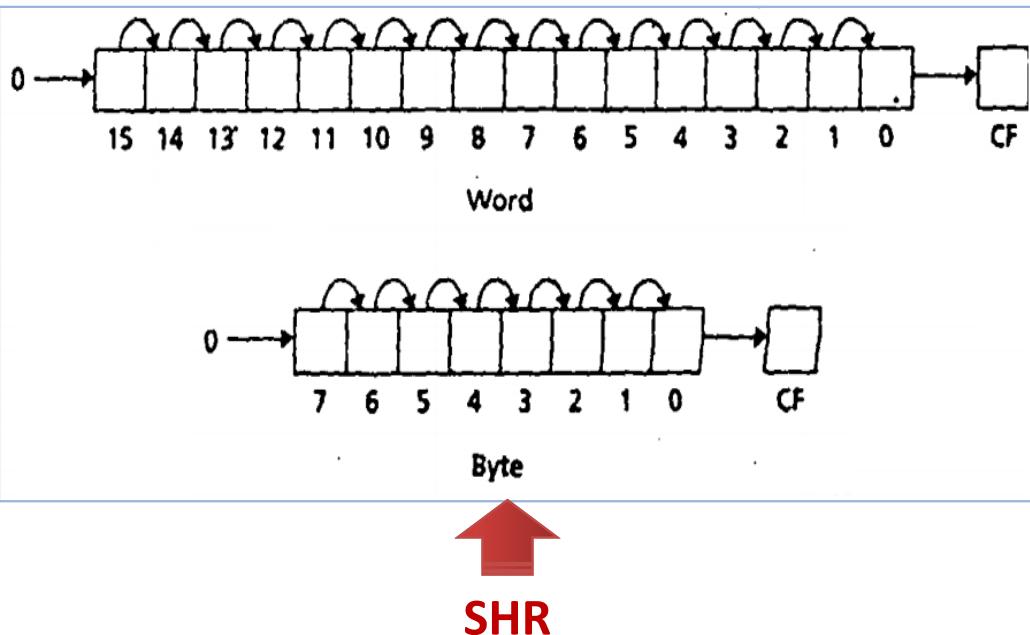
DH=00H
CF=0
OF=0

What is the result of
SHL DH, CL

CF and OF are not reliable for multiple shifts because multiple shifts are just a series of single shifts and CF, OF only reflect the result of the last shift.

SHR and SAR instructions

- Shifts the bits in the destination to the right
- The LSB is shifted into CF
- In case of SAR MSB retains its original value



SHR and SAR instructions

- SHR should be used for unsigned interpretation as it does not preserve sign
- SAR should be used for signed interpretation as it preserves sign

SHR and SAR instructions

- Example
 - ✓ If AL=-15, CL=1
What is the result of SHR AL, CL
 - ✓ If AL=-15, CL=1
What is the result of SAR AL, CL
- AL=120H
CF=1
OF=1
- AL=-8
CF=1
OF=1

Multiplication instructions

- **imul source**
 - Signed multiplication
- **mul source**
 - Unsigned multiplication
- **Byte and Word Multiplication (A X B)**
 - If two **bytes** are multiplied, the result is a 16-bit **word**
 - A: **source**
 - B: **al**
 - product: **ax**
 - If two **words** are multiplied, the result is a 32-bit **doubleword**
 - A: **source**
 - B: **ax**
 - Product (ms 16 bits): **dx**
 - Product (ls 16 bits): **ax**

Multiplication instructions

- *source* can be a register or memory location (not a constant)
- Byte form
 - AX=AL**source*
- Word form
 - DX:AX=AX**source*
 - If **ax** contains **0001h** and **bx** contains **FFFFh**
 - **mul bx;** **dx = 0000h** **ax = FFFFh**
 - **imul bx ;** **dx = FFFFh** **ax = FFFFh (-1)**

Signed Multiplication

C

Label1:

```
short int x=0x8000;  
short int y=0xFFFF;  
x=x*y
```

Assembly

Label1:

```
MOV x, 8000H  
MOV y, FFFFH  
MOV AX, x  
MOV BX, y  
IMUL BX
```

Unsigned Multiplication

C

Label1:

```
unsigned short int x=0x8000;  
unsigned short int y=0xFFFF;  
x=x*y
```

Assembly

Label1:

```
MOV x, 8000H  
MOV y, FFFFH  
MOV AX, x  
MOV BX, y  
MUL BX
```

Multiplication instructions

- Effect on flags
 - ✓ SF, ZF, AF, and PF Undefined
 - CF/OF
 - MUL
 - 0: if upper half result 0
 - 1: Otherwise
 - IMUL
 - 0: if upper half is sign extension of lower half.
 - 1: Otherwise

More Examples

- AX=FFFFh, BX=FFFFh

Instruction	Hex Product	DX	AX	CF/OF
MUL BX	FFFE0001 (4294836225)	FFFE (!zero)	0001	1
IMUL BX	1	0000	0001	0

- AX=80h, BX=FFh

Instruction	Hex Product	AH	AL	CF/OF
MUL BX	7F80 (128)	7F(!zero)	80	1
IMUL BX	0080	00 (no sign extension)	80	1

References

- Ch 7, 9 Assembly Language Programming – by Yu and Marut

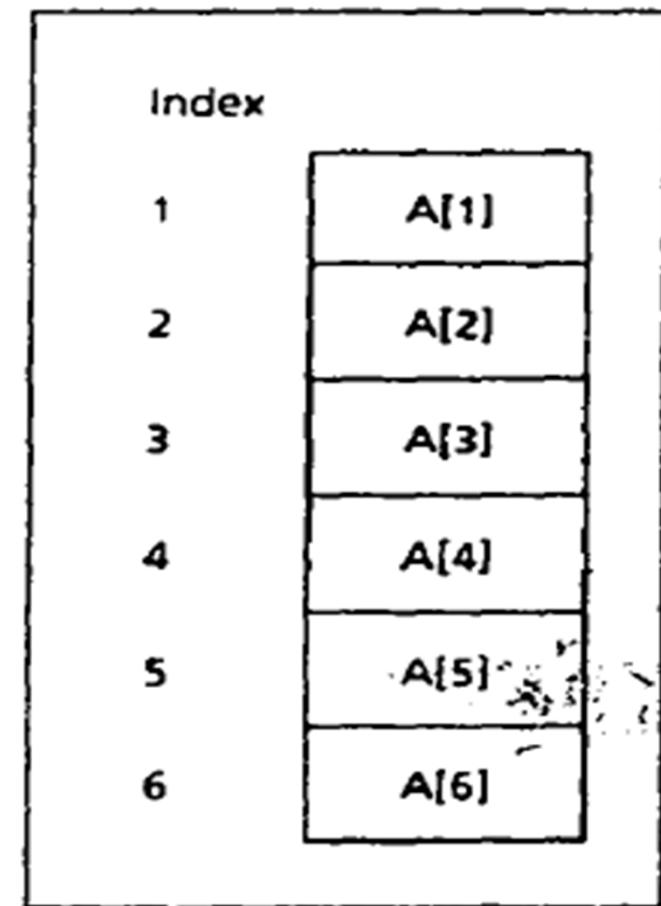
CSE 315

Microprocessors, Microcontrollers, and
Embedded Systems

Arrays and Addressing Modes

One Dimensional Array

- A one dimensional array is an ordered list of elements
- A 5 character string named MSG
MSG DB 'abcde'
- A word array W of 6 integers
W DW 10, 20, 30, 40, 50, 60



One Dimensional Array

- The address of the array variable is called the base address of the array
- The array W is like this

<i>Offset address</i>	<i>Symbolic address</i>	<i>Decimal content</i>
0200h	W	10
0202h	W+2h	20
0204h	W+4h	30
0206h	W+6h	40
0208h	W+8h	50
020Ah	W+Ah	60

DUP operator

repeat_count DUP (value)

- An array of 100 words, each initialized to 0

 GAMMA DW 100 DUP (0)

- An array of 212 uninitialized bytes.

 DELTA DB 212 DUP (?)

- Nested DUP operator

 LINE DB 5, 4, 3 DUP (2, 3 DUP (0), 1)

is equivalent to

 LINE DB 5, 4, 2, 0, 0, 0, 1, 2, 0, 0, 0, 1, 2, 0,
0, 0, 1

Location of Array Elements

- A is an array of N elements
- S is the number of bytes in an element of the array
 - S = 1 (for byte array)
 - = 2 (for word array)
- So, the location of the elements of A can be determined as

<i>Position</i>	<i>Location</i>
1	A
2	$A + 1 \times S$
3	$A + 2 \times S$
.	.
.	.
N	$A + (N - 1) \times S$

Addressing Modes

- So far we have used
 1. Register mode operand is a register
 2. Immediate mode operand is constant
 3. Direct mode operand is a variable
- MOV AX, 0 ; Destination AX is register mode, source 0 is immediate mode
- ADD X, AX ; Destination X is direct mode, source AX is register mode
- There are four additional addressing modes
 1. Register indirect mode
 2. Based mode
 3. Indexed mode
 4. Based indexed mode

Register Indirect Mode

- The offset address of the operand is contained in a register.
 - Operand format [register]
 - Register can be BX, SI, DI, or BP.
-
- If BX, SI, or DI contains the offset of the operand,
DS contains the segment number
 - If BP contains the offset of the operand,
SS contains the segment number

Register Indirect Mode

Write a code to sum in AX the elements of the 10-element array W defined by

W DW 10, 20, 30, 40, 50, 60, 70, 80, 90, 100

XOR	AX, AX	; AX holds sum
LEA	SI, W	; SI points to array W
MOV	CX, 10	; CX has number of elements
ADDNOS:		
ADD	AX, [SI]	; sum=sum+element
ADD	SI, 2	; move pointer to the next element
LOOP	ADDNOS	; loop until done

Based Mode

- Operand format
 - [register + displacement]
 - [displacement + register]
 - [register] + displacement
 - displacement + [register]
 - displacement [register]
- Register can be BX (base register), or BP (base pointer)
- Displacement can be
 - the offset address of a variable (e.g., A)
 - a constant (positive or negative) (e.g., -2)
 - the offset address of a variable plus or minus a constant (A + 2)
- If BX is used as register, DS contains the segment number
- If BP is used as register, SS contains the segment number

Based Mode

- Suppose W is a word array and BX contains 4
- Displacement is the offset address of variable W.
- So, to move the 3rd element of the array to AX, we can write

MOV AX, [BX + W]

;[register + displacement]

MOV AX, [W + BX]

;[displacement + register]

MOV AX, [BX] + W

;[register] + displacement

MOV AX, W + [BX]

;displacement + [register]

MOV AX, W [BX]

;displacement [register]

Indexed Mode

- Operand format
 - [register + displacement]
 - [displacement + register]
 - [register] + displacement
 - displacement + [register]
 - displacement [register]
- Register can be SI (source index), or DI (destination index)
- Displacement can be
 - the offset address of a variable (e.g., A)
 - a constant (positive or negative) (e.g., -2)
 - the offset address of a variable plus or minus a constant (A + 2)
- If SI, or DI is used as register, DS contains the segment number

PTR operator

- Instructions such as

MOV [BX], 1

is not allowed since whether the destination is byte or word is unknown

- The PTR operator can be used for type casting

MOV BYTE PTR [BX], 1

MOV WORD PTR [BX], 1

CSE 315

Microprocessors, Microcontrollers, and
Embedded Systems

Assembly Language:
Stack and Procedures

Stack Segment

- A block of memory to store stack
- Syntax
 - **.STACK size**
 - Where size is optional and specifies the stack area size in bytes
 - If size is omitted, 1 KB set aside for stack area
- For example:
.STACK 100h
- SS contains segment number of stack segment
- SP contains offset address of the top of the stack

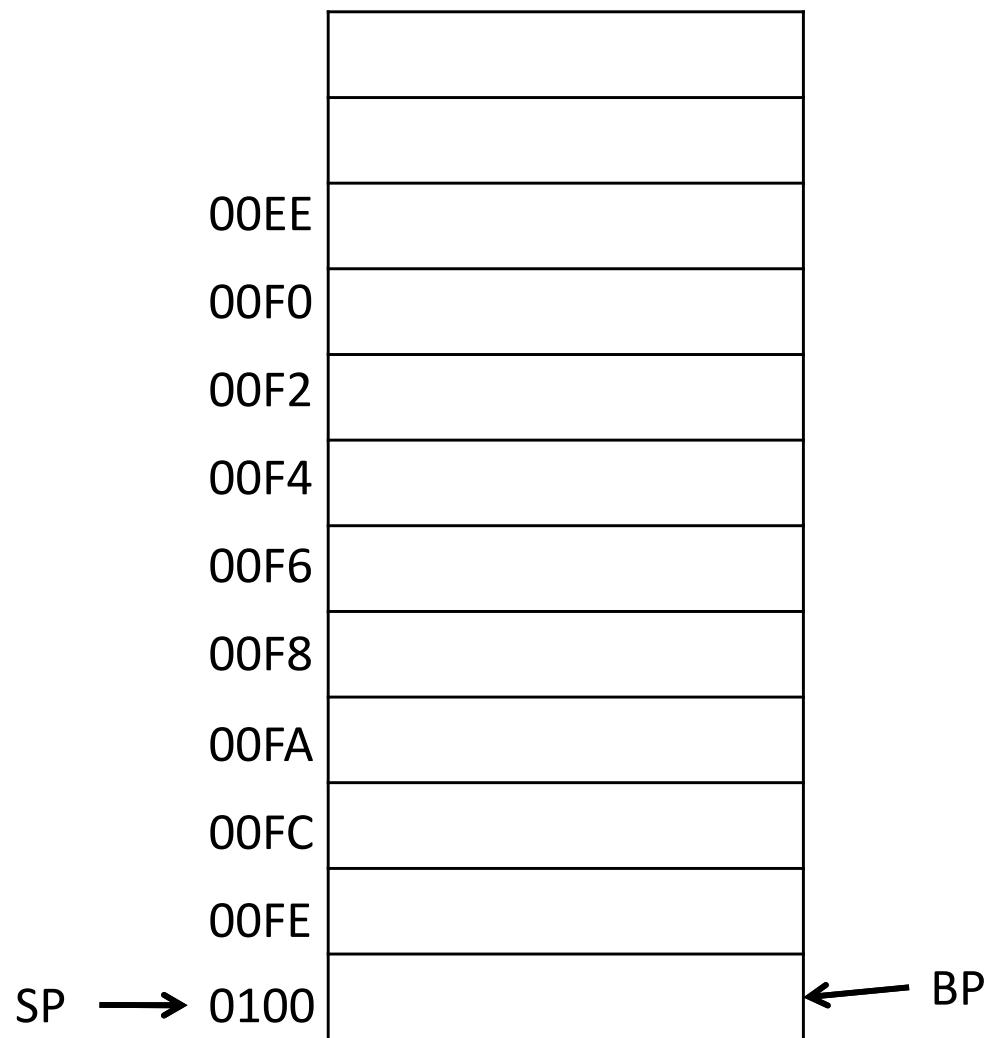
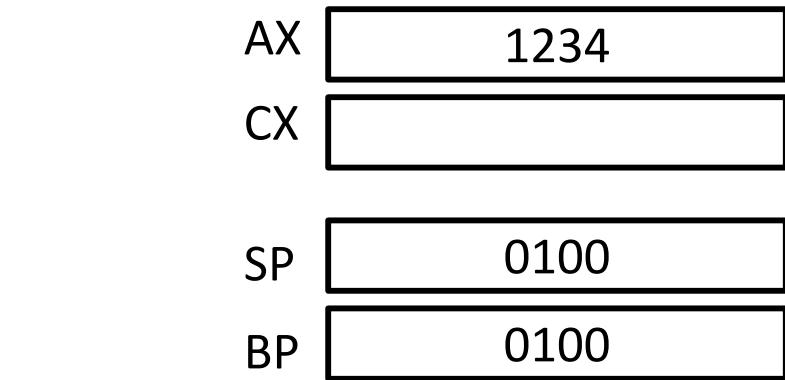
PUSH

PUSH source

- source is a 16-bit register / memory word

PUSH AX

- SP is decreased by 2
- Source content is copied to the address SS:SP



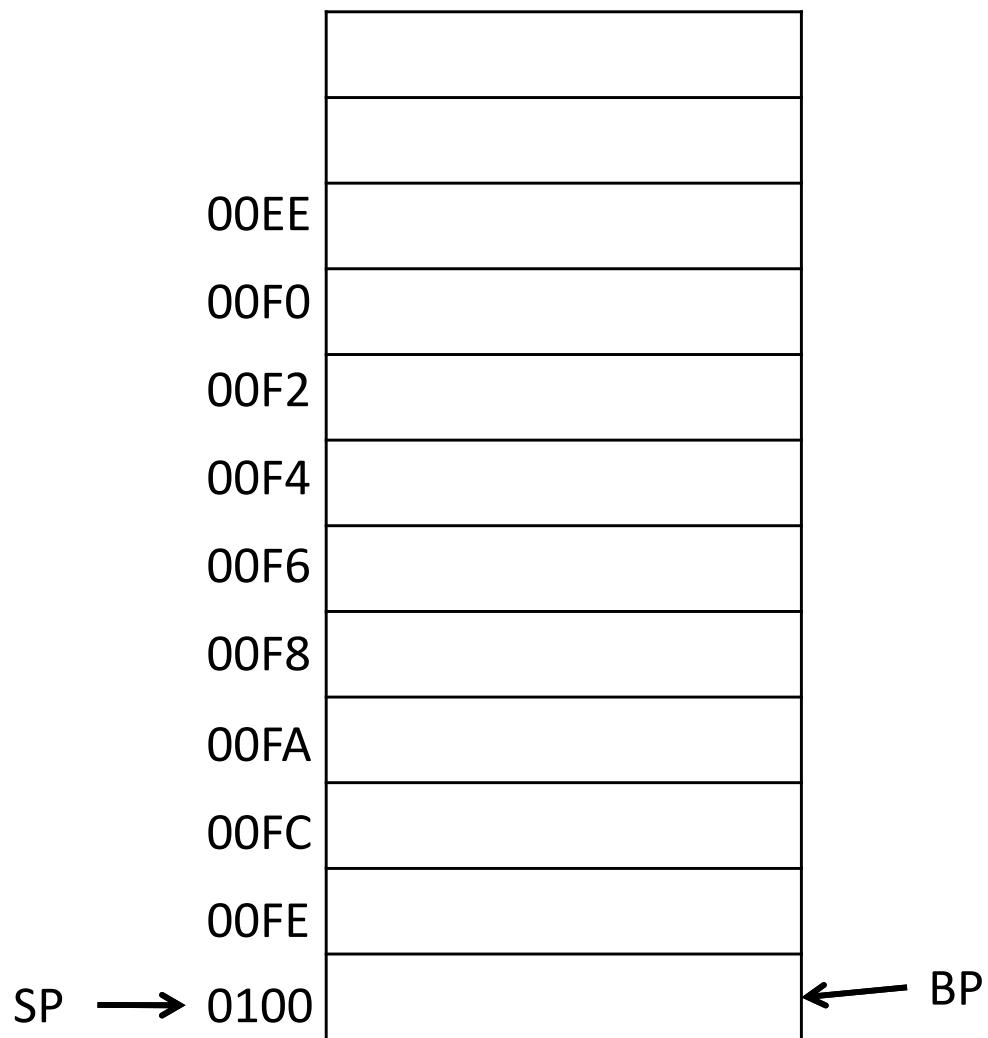
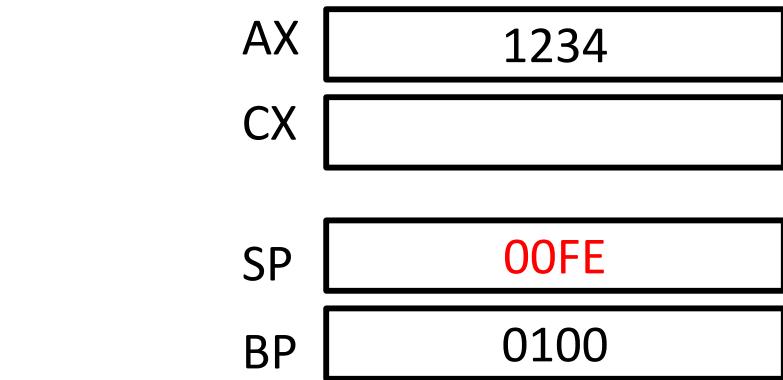
PUSH

PUSH source

- source is a 16-bit register / memory word

PUSH AX

- SP is decreased by 2
- Source content is copied to the address SS:SP



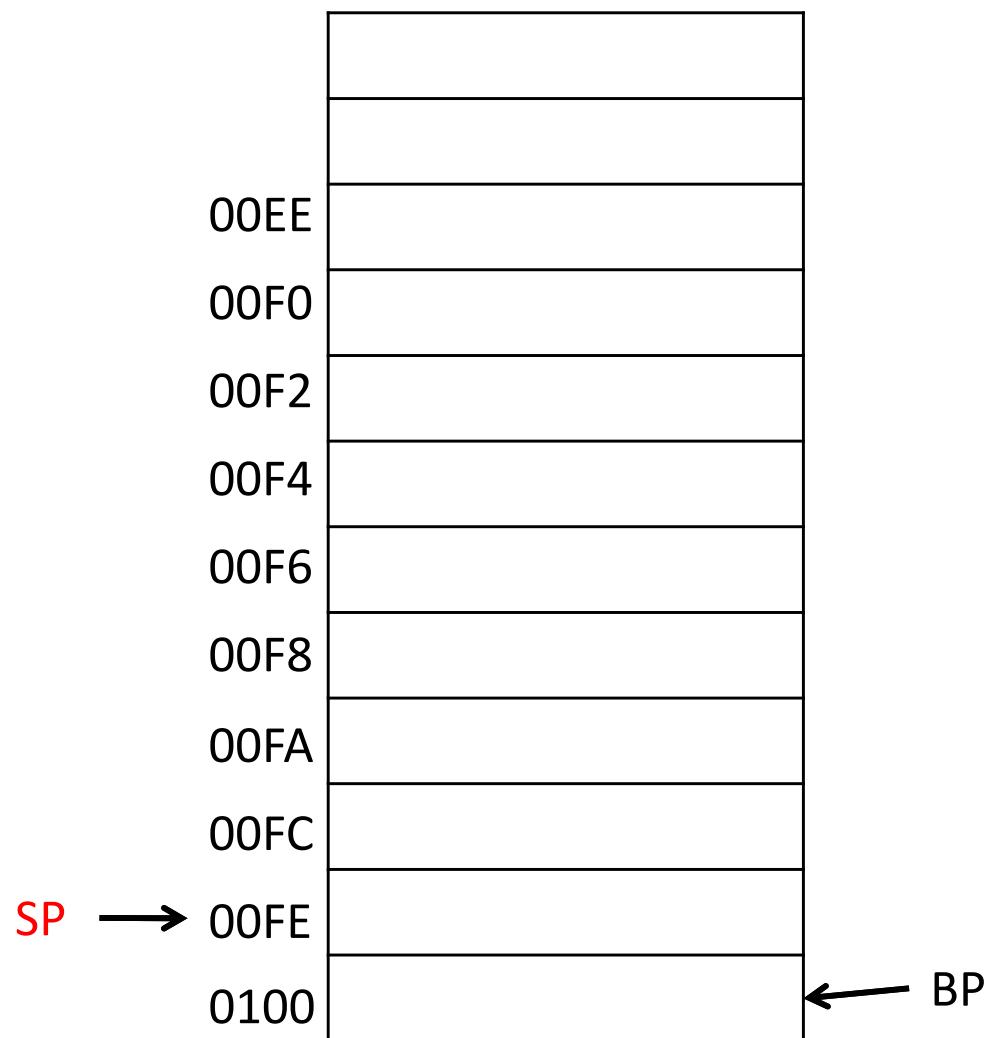
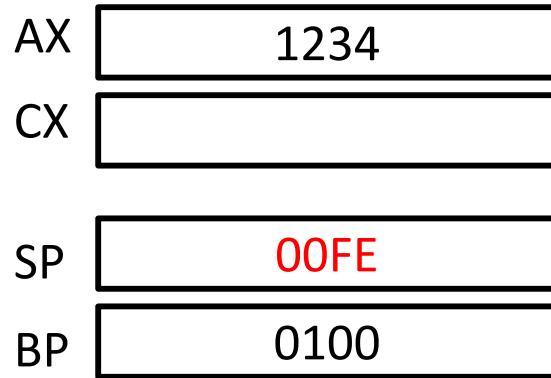
PUSH

PUSH source

- source is a 16-bit register / memory word

PUSH AX

- SP is decreased by 2
- Source content is copied to the address SS:SP



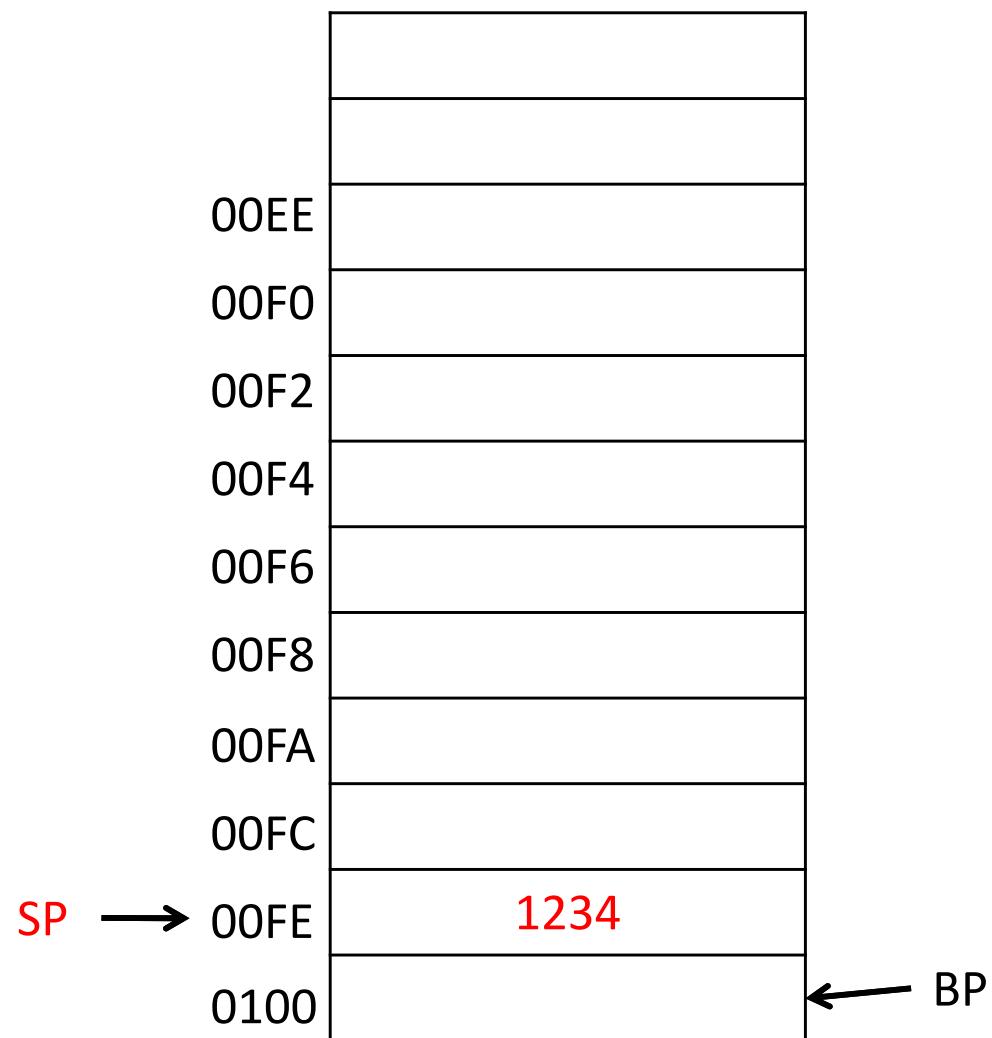
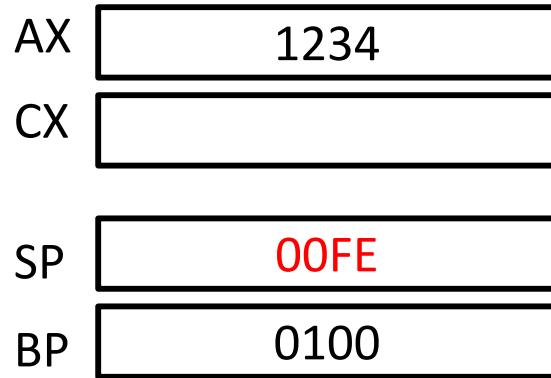
PUSH

PUSH source

- source is a 16-bit register / memory word

PUSH AX

- SP is decreased by 2
- Source content is copied to the address SS:SP



POP

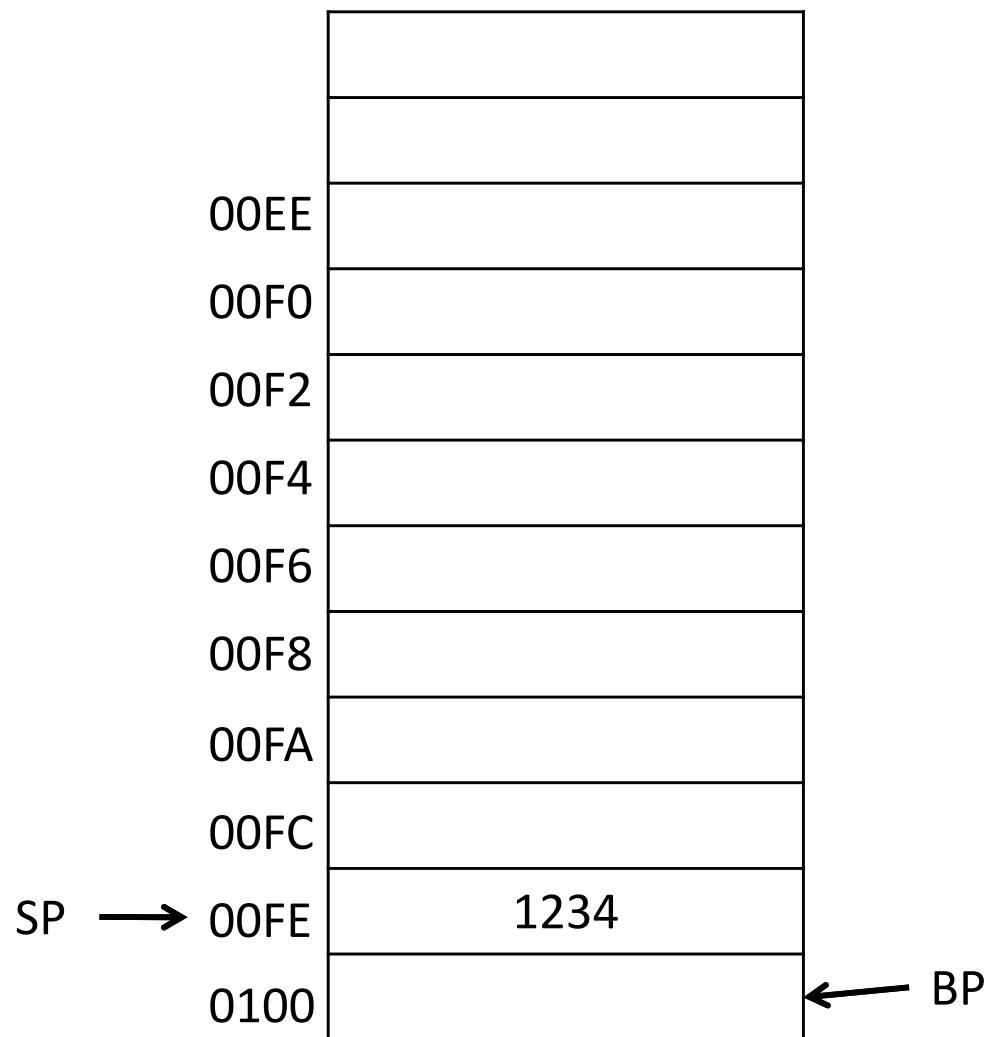
POP destination

AX	1234
CX	
SP	00FE
BP	0100

- destination is a 16-bit register(except IP register) / memory word

POP CX

1. The content of SS:SP is moved to the destination
2. SP is increased by 2



POP

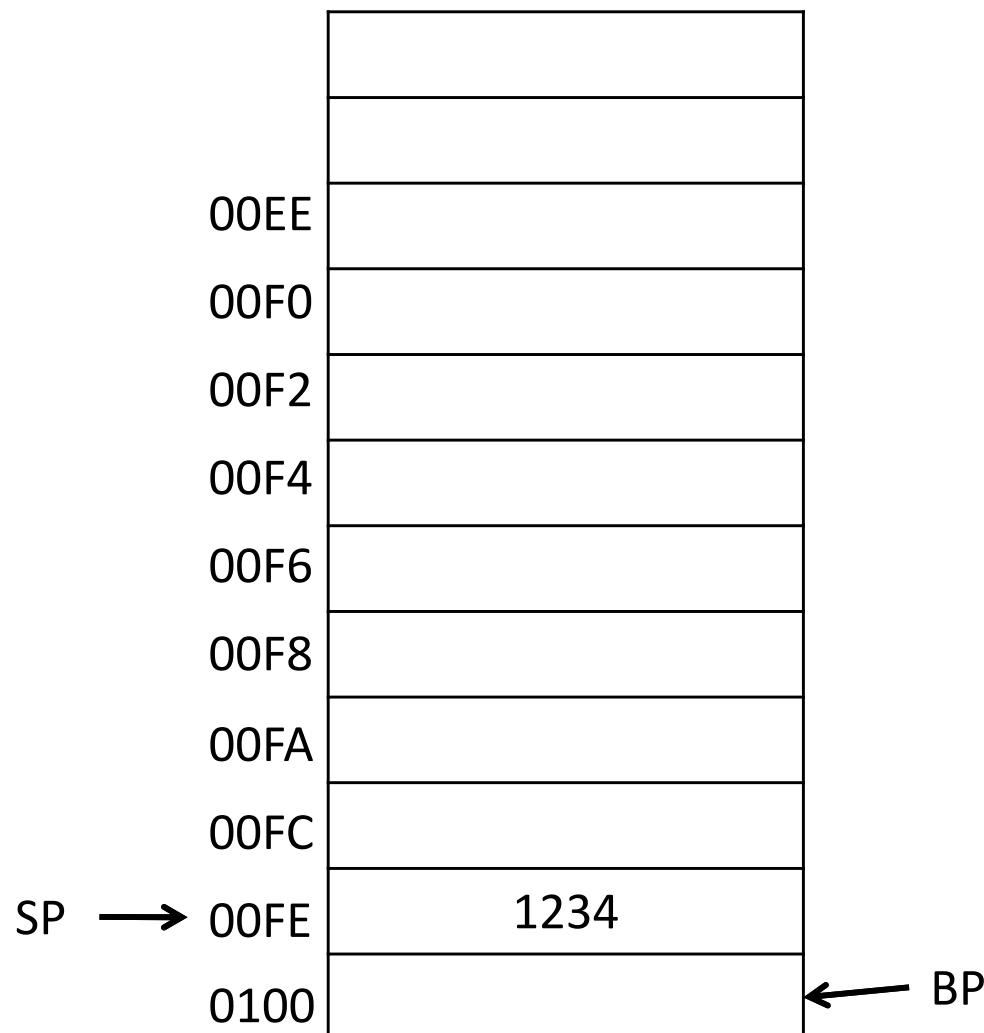
POP destination

- destination is a 16-bit register(except IP register) / memory word

POP CX

1. The content of SS:SP is moved to the destination
2. SP is increased by 2

AX	1234
CX	1234
SP	00FE
BP	0100



POP

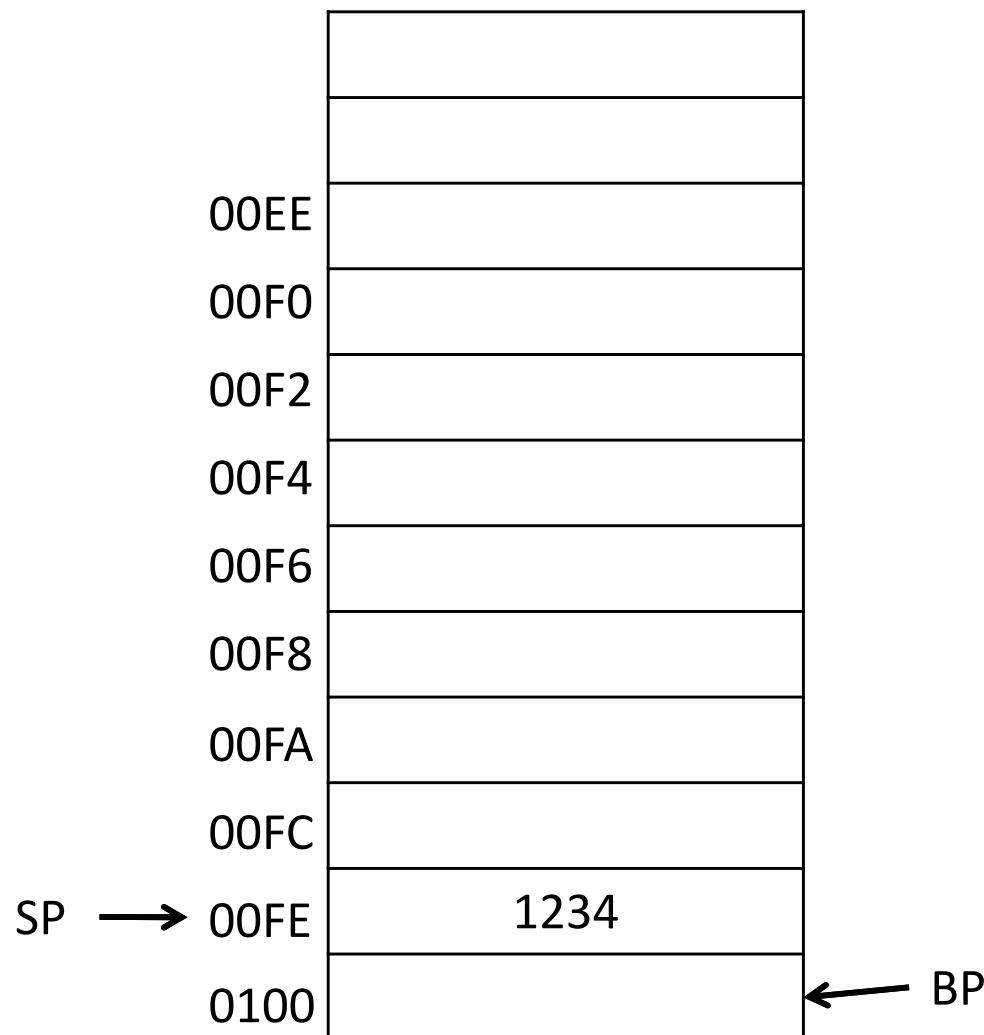
POP destination

- destination is a 16-bit register(except IP register) / memory word

POP CX

1. The content of SS:SP is moved to the destination
2. SP is increased by 2

AX	1234
CX	1234
SP	0100
BP	0100



POP

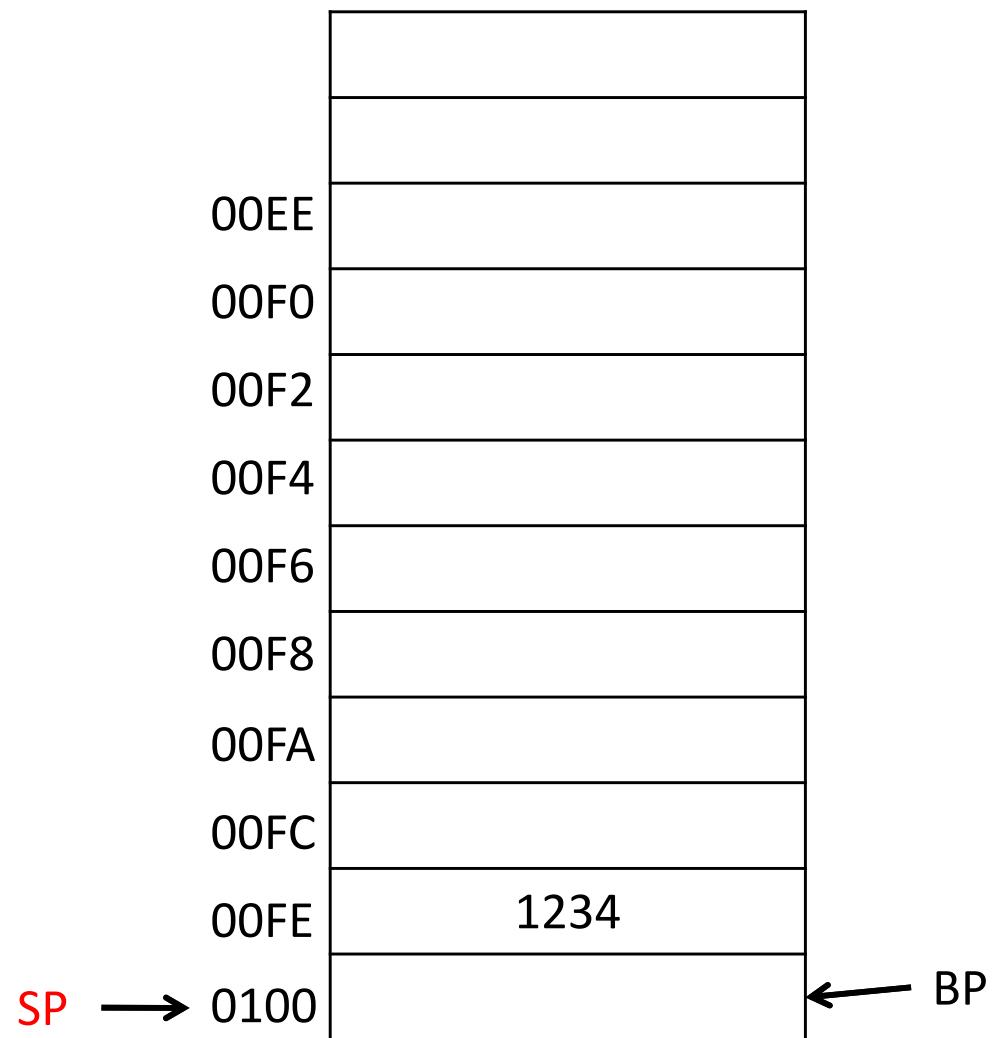
POP destination

- destination is a 16-bit register(except IP register) / memory word

POP CX

1. The content of SS:SP is moved to the destination
2. SP is increased by 2

AX	1234
CX	1234
SP	0100
BP	0100



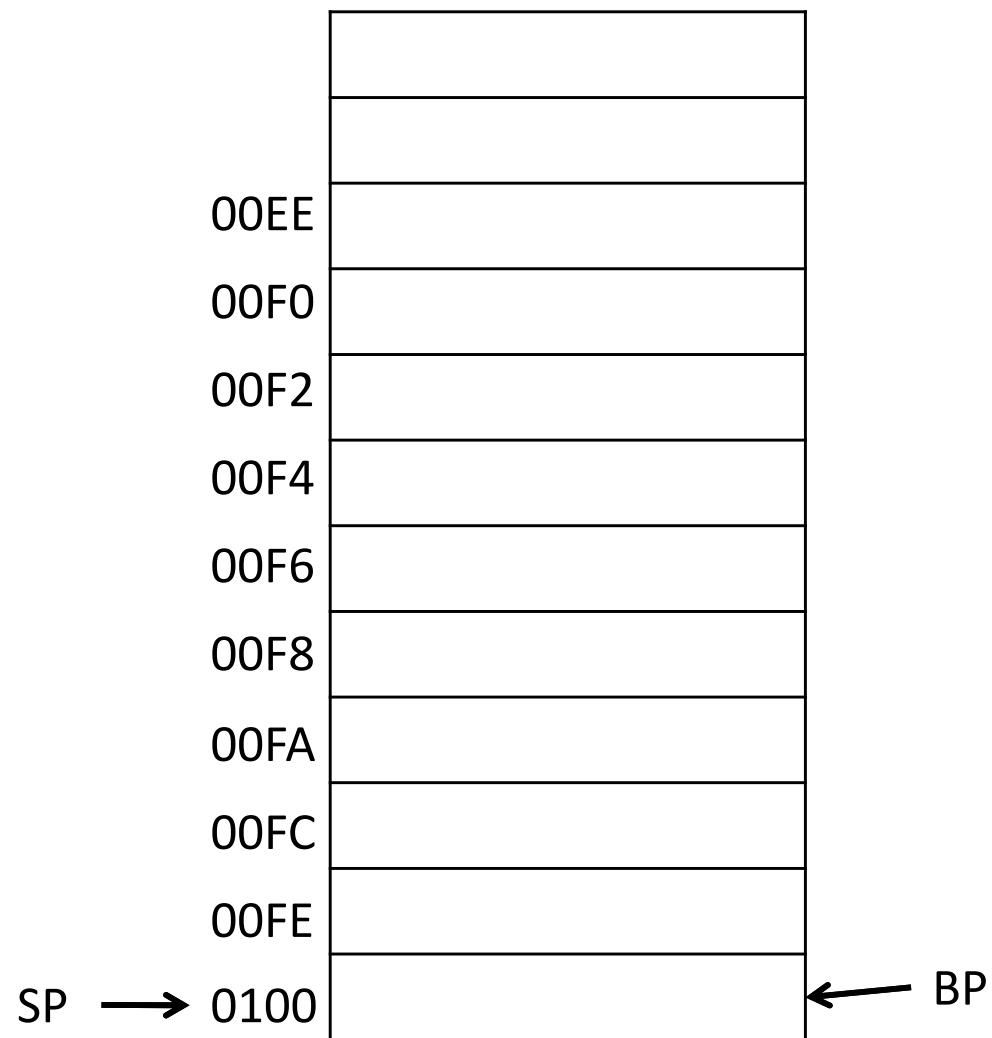
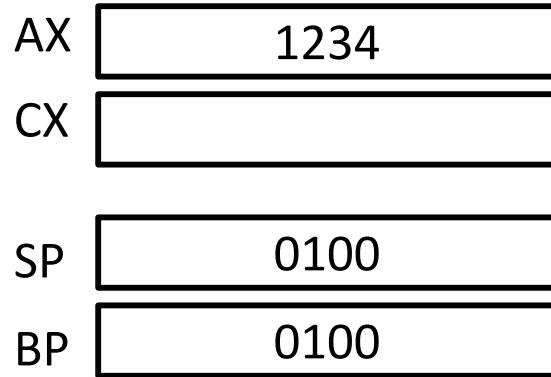
PUSHF and POPF

PUSHF

- No source operand is allowed here
- Pushes the contents of FLAGS register onto the stack

POPF

- Reverse of PUSHF
- Pops the top of the stack into FLAGS register



Order of PUSH and POP

PUSH A

PUSH B

PUSH C

...

...

POP C

POP B

POP A

Procedure Declaration

```
name    PROC   type  
; body of the procedure  
        RET  
name    ENDP
```

“name” is the user-defined name of the procedure

“type” is optional, can be “FAR”/“NEAR”

“RET” causes control back to the calling procedure

```
name PROC type  
; body of the procedure  
RET  
name ENDP
```

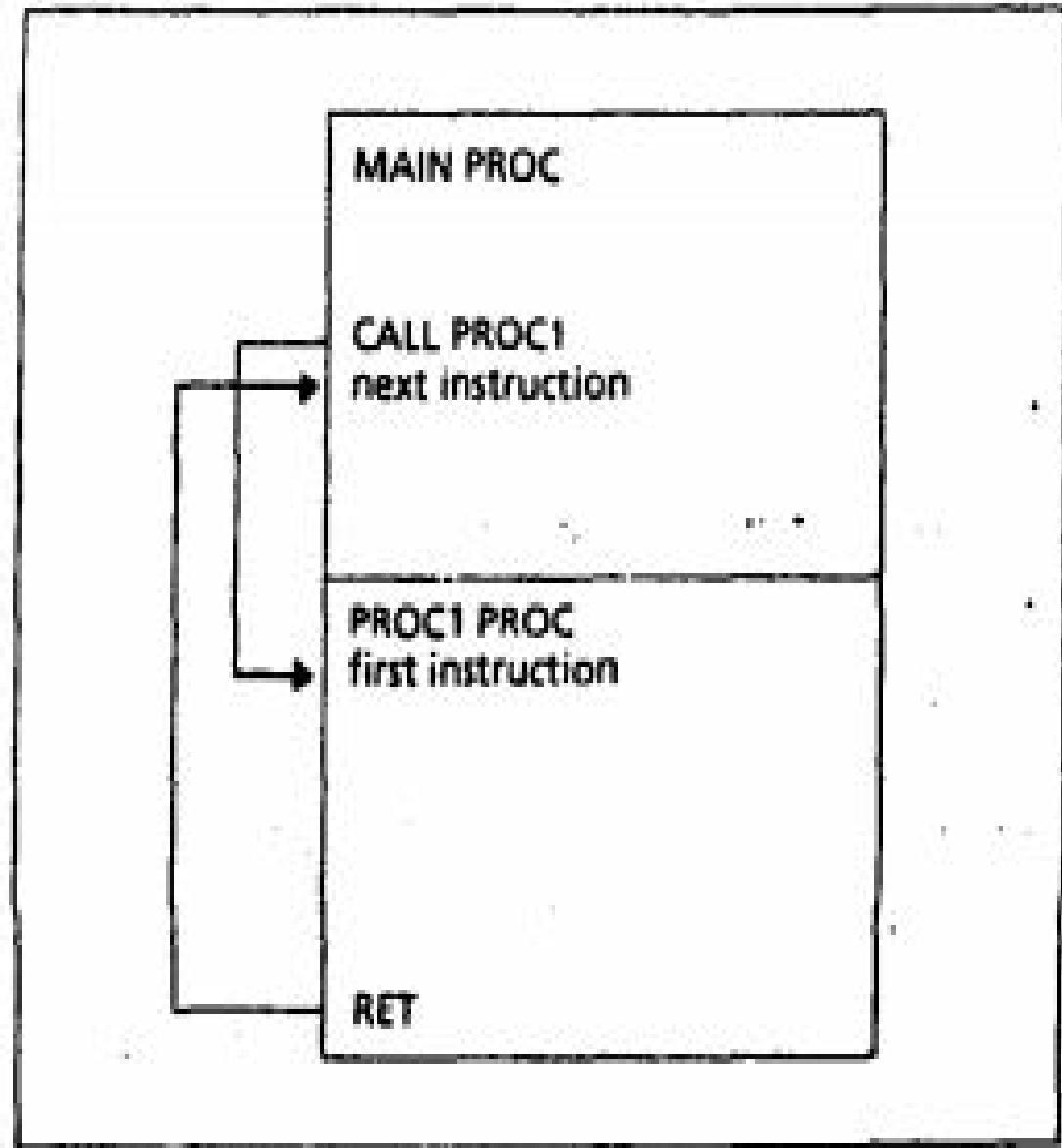


Fig: Procedure CALL and RET

CALL

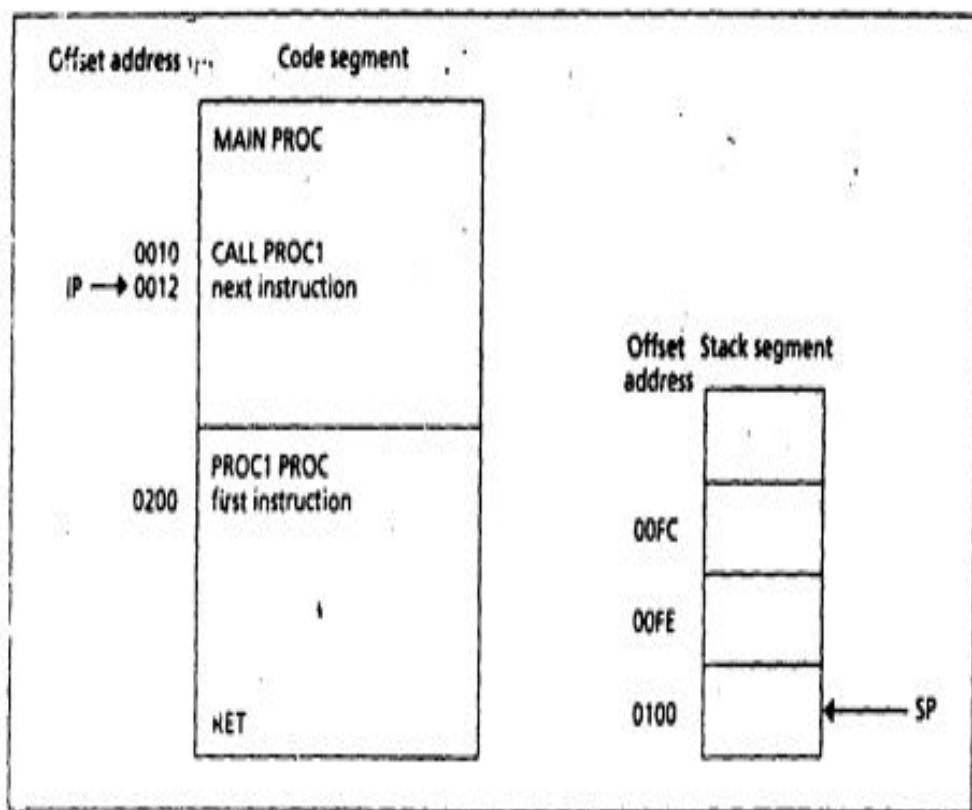


Figure 8.4A Before CALL

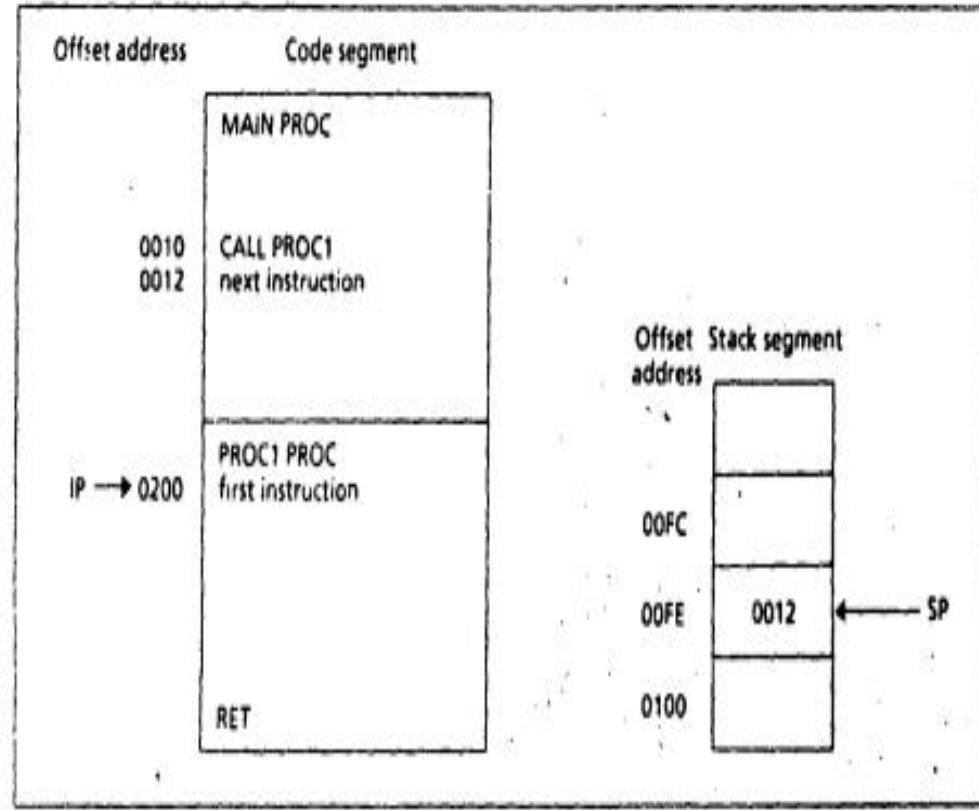


Figure 8.4B After CALL

RET

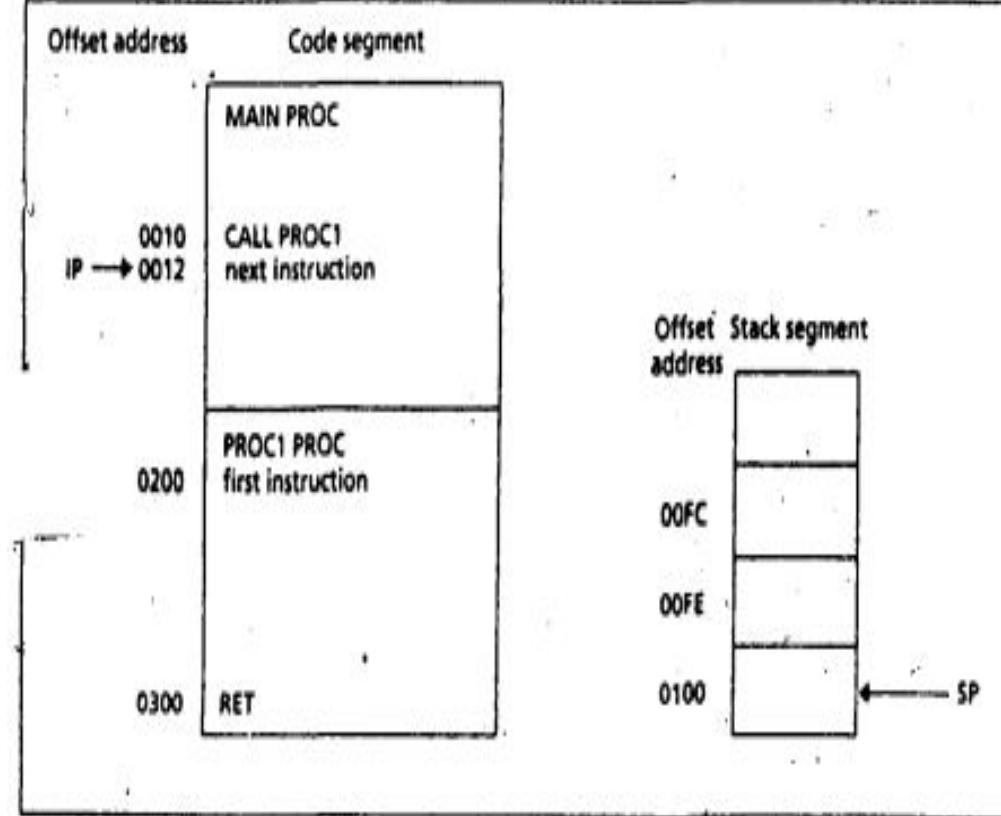
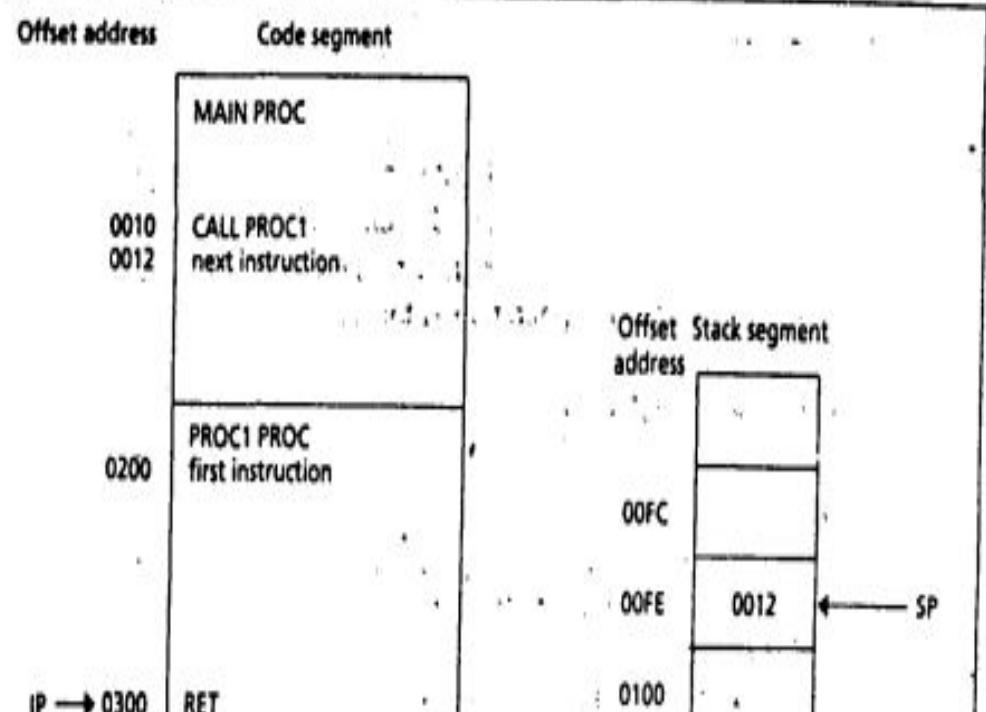


Figure 8.5A Before RET

Figure 8.5B After RET

Example of a procedure

Unsigned multiplication of two positive integers A and B by addition and bit shifting

```
Product = 0
REPEAT
    IF lsb of B is 1
    THEN
        Product= Product+A
    END_IF
    Shift left A
    Shift right B
UNTIL B=0
```

$$\begin{array}{r} 111b \\ \times 1101b \\ \hline 111 \\ 000 \\ 111 \\ \hline 1011011b \end{array}$$

```
Product = 0
Since lsb of B is 1, Product = 0 + 111b = 111b
Shift left A: A = 1110b
Shift right B: B = 110b

Since lsb of B is 0,
Shift left A: A = 11100b
Shift right B: B = 11b
.

Since lsb of B is 1
Product = 111b + 11100b = 100011b
Shift left A: A = 111000b
Shift right B: B = 1

Since lsb of B is 0
Product = 100011b + 111000b = 1011011b
Shift left A: A = 1110000b
Shift right B: B = 0

Since lsb of B = 0
Return Product = 1011011b = 91d
```

Example of a procedure

Unsigned multiplication of two positive integers A and B by addition and bit shifting

```
Product = 0
REPEAT
    IF lsb of B is 1
    THEN
        Product= Product+A
    END_IF
    Shift left A
    Shift left B
UNTIL B=0
```

```
.MODEL SMALL
.STACK 100H
.CODE
MAIN PROC
    MOV AX, @DATA
    MOV DS, AX
    MOV AX, 10
    MOV BX, 6
    CALL MULTIPLY
    ...
    MOV AH, 4CH
    INT 21H
MAIN ENDP

MULTIPLY PROC
    PUSH AX
    PUSH BX
    XOR DX, DX
    REPEAT:
        TEST BX, 1
        JZ END_IF
        ADD DX, AX
        END_IF:
        SHL AX, 1
        SHR BX, 1
        JNZ REPEAT
    POP BX
    POP AX
    RET
MULTIPLY ENDP
END MAIN
```

Chapter 17

Recursion

Parameter passing

- We can push the parameters into the stack
- Use BP to access the parameters
 - BP is saved in the stack for later
- RET n
 - Return and pop *n* additional bytes

Activation Record

- Record to be stored before CALLing a recursive function

SP and BP →

saved BP value (from second call)
return addr (in second call)
parameters and local variables

original BP value (from first call)
return addr (in first procedure)
parameters and local variables

original BP
return addr (in main procedure)
parameters

} activation
record
third call

} activation
record
second call

} activation
record
first call

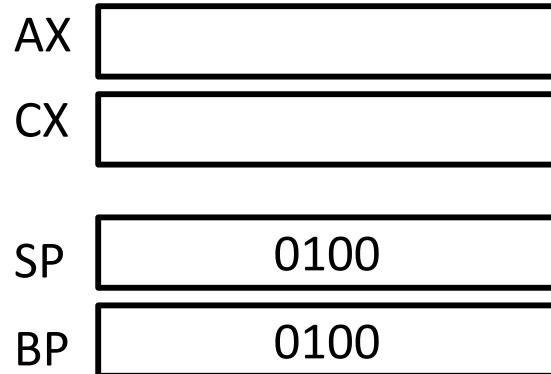
Factorial

- $\text{Factorial}(n) = n! = n * (n-1) * (n-2) * \dots * 3 * 2 * 1$

$$\text{Factorial}(n) = n * \text{Factorial}(n-1)$$

MAIN PROC

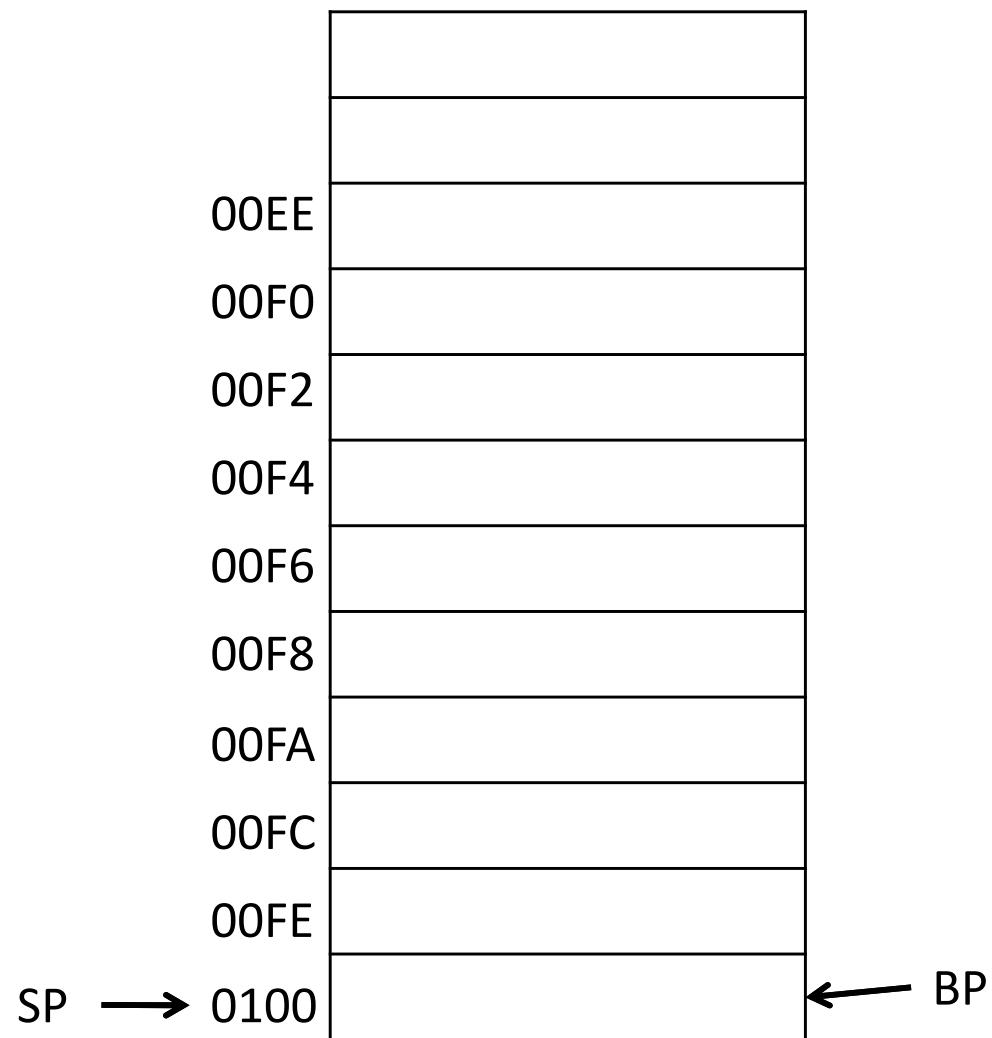
0000	MOV	AX,3	
0002	PUSH	AX	
0004	CALL	FACTORIAL	
0006	MOV	AH,4CH	
0008	INT	21H	



MAIN ENDP

FACTORIAL PROC NEAR

000A	PUSH	BP	
000C	MOV	BP,SP	
000E	CMP	WORD PTR[BP+4],1	
0010	JG	END_IF	
0012	MOV	AX,1	
0014	JMP	RETURN	
0016	END_IF:	MOV CX, [BP+4]	
0018	DEC	CX	
001A	PUSH	CX	
001C	CALL	FACTORIAL	
001E	MUL	WORD PTR[BP+4]	
0020	RETURN :	POP BP	
0022		RET 2	
	END	MAIN	



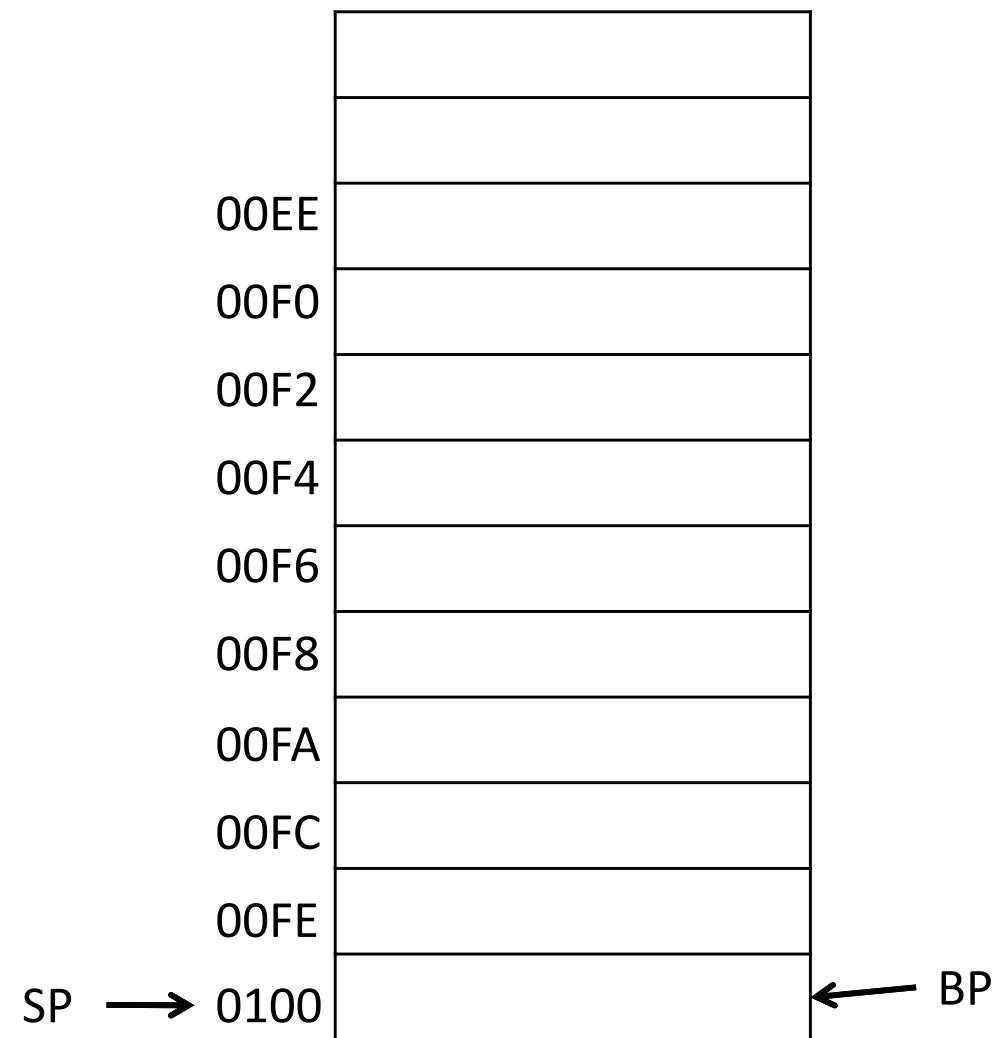
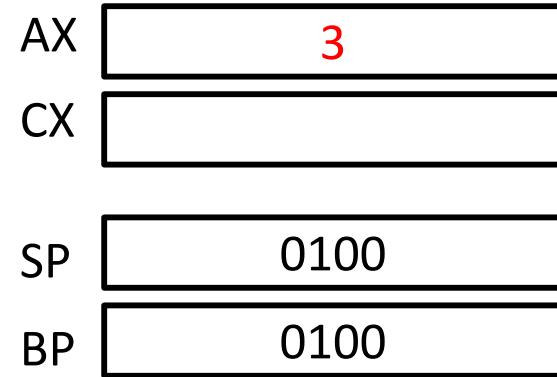
MAIN PROC

0000	MOV	AX,3
0002	PUSH	AX
0004	CALL	FACTORIAL
0006	MOV	AH,4CH
0008	INT	21H

MAIN ENDP

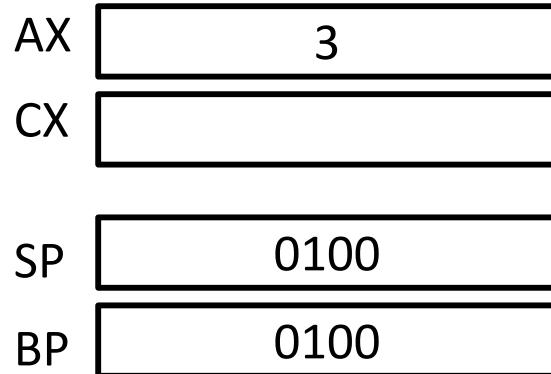
FACTORIAL PROC NEAR

000A	PUSH	BP
000C	MOV	BP,SP
000E	CMP	WORD PTR[BP+4],1
0010	JG	END_IF
0012	MOV	AX,1
0014	JMP	RETURN
0016	END_IF:	MOV CX, [BP+4]
0018	DEC	CX
001A	PUSH	CX
001C	CALL	FACTORIAL
001E	MUL	WORD PTR[BP+4]
0020	RETURN :	POP BP
0022		RET 2
	END	MAIN



MAIN PROC

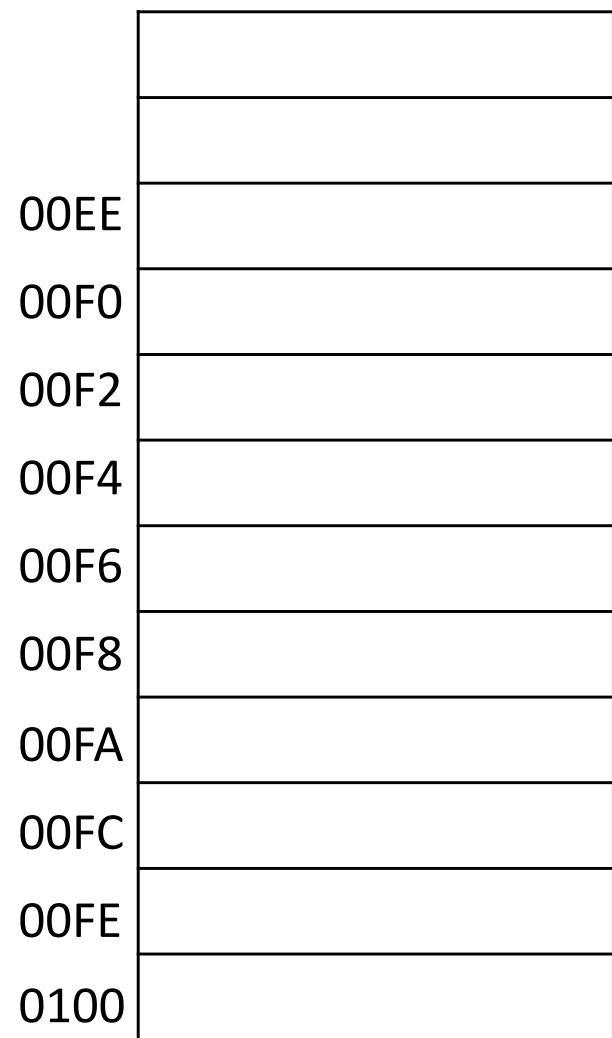
0000	MOV	AX,3	
0002	PUSH	AX	
0004	CALL	FACTORIAL	
0006	MOV	AH,4CH	
0008	INT	21H	



MAIN ENDP

FACTORIAL PROC NEAR

000A	PUSH	BP	
000C	MOV	BP,SP	
000E	CMP	WORD PTR[BP+4],1	
0010	JG	END_IF	
0012	MOV	AX,1	
0014	JMP	RETURN	
0016	END_IF:	MOV CX, [BP+4]	
0018	DEC	CX	
001A	PUSH	CX	
001C	CALL	FACTORIAL	
001E	MUL	WORD PTR[BP+4]	
0020	RETURN :	POP BP	
0022		RET 2	
	END	MAIN	



SP → 0100 ← BP

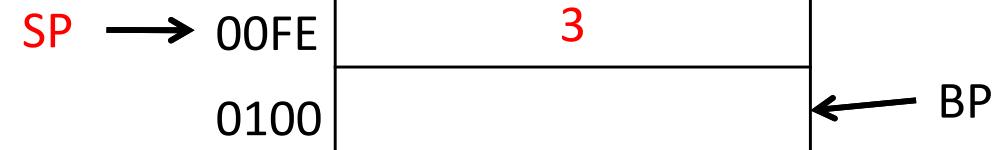
MAIN PROC

0000	MOV	AX,3	AX	3
0002	PUSH	AX	CX	
0004	CALL	FACTORIAL	SP	00FE
0006	MOV	AH,4CH	BP	0100
0008	INT	21H		

MAIN ENDP

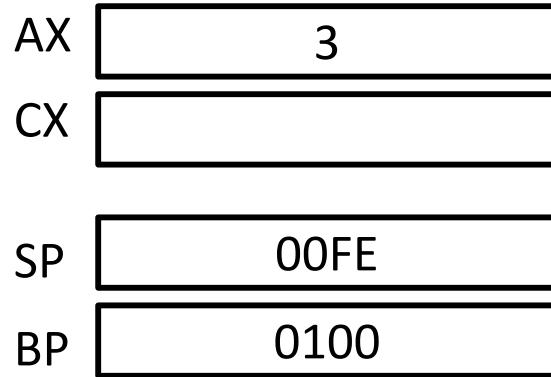
FACTORIAL PROC NEAR

000A	PUSH	BP		
000C	MOV	BP,SP		
000E	CMP	WORD PTR[BP+4],1	00EE	
0010	JG	END_IF	00F0	
0012	MOV	AX,1	00F2	
0014	JMP	RETURN	00F4	
0016	END_IF:	MOV CX, [BP+4]	00F6	
0018	DEC	CX	00F8	
001A	PUSH	CX	00FA	
001C	CALL	FACTORIAL	00FC	
001E	MUL	WORD PTR[BP+4]	00FE	3
0020	RETURN :	POP BP		
0022		RET 2		
	END	MAIN		



MAIN PROC

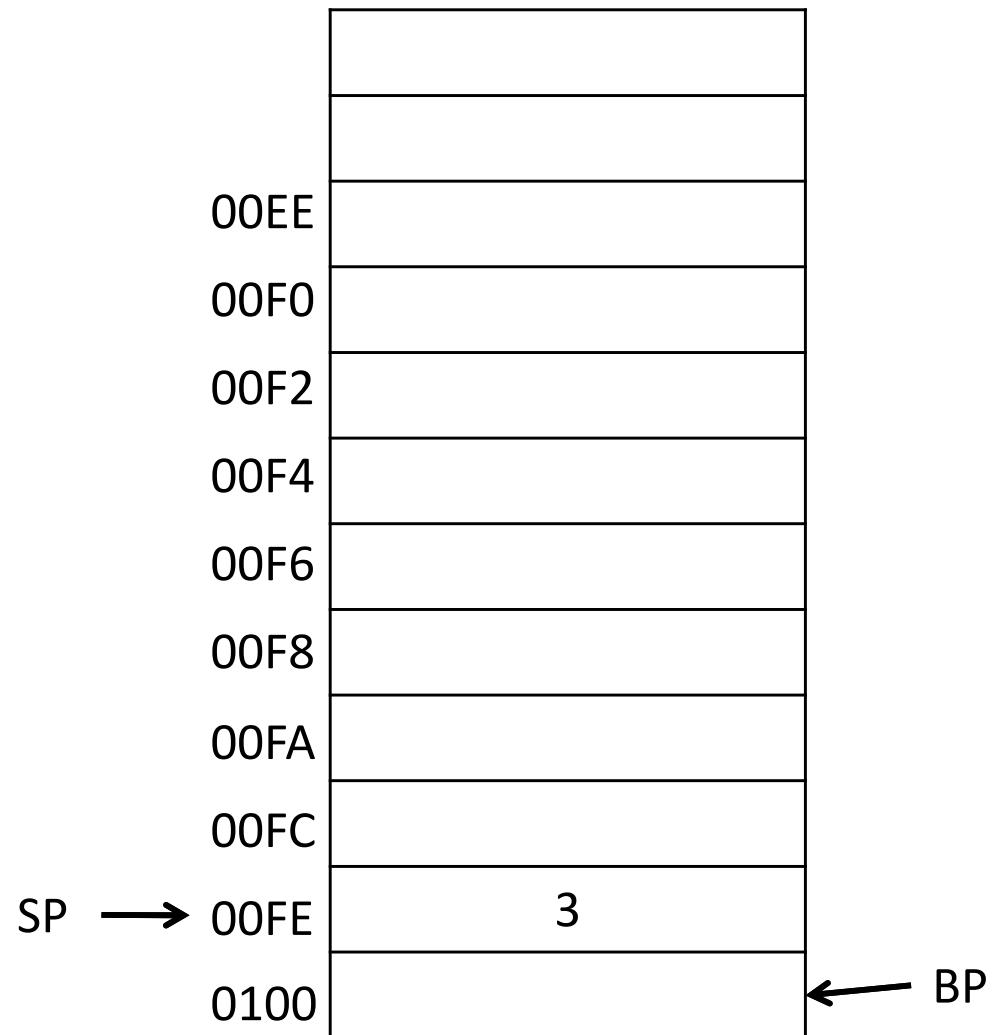
0000	MOV	AX,3
0002	PUSH	AX
0004	CALL	FACTORIAL
0006	MOV	AH,4CH
0008	INT	21H



MAIN ENDP

FACTORIAL PROC NEAR

000A	PUSH	BP
000C	MOV	BP,SP
000E	CMP	WORD PTR[BP+4],1
0010	JG	END_IF
0012	MOV	AX,1
0014	JMP	RETURN
0016	END_IF:	MOV CX, [BP+4]
0018	DEC	CX
001A	PUSH	CX
001C	CALL	FACTORIAL
001E	MUL	WORD PTR[BP+4]
0020	RETURN :	POP BP
0022		RET 2
	END	MAIN



MAIN PROC

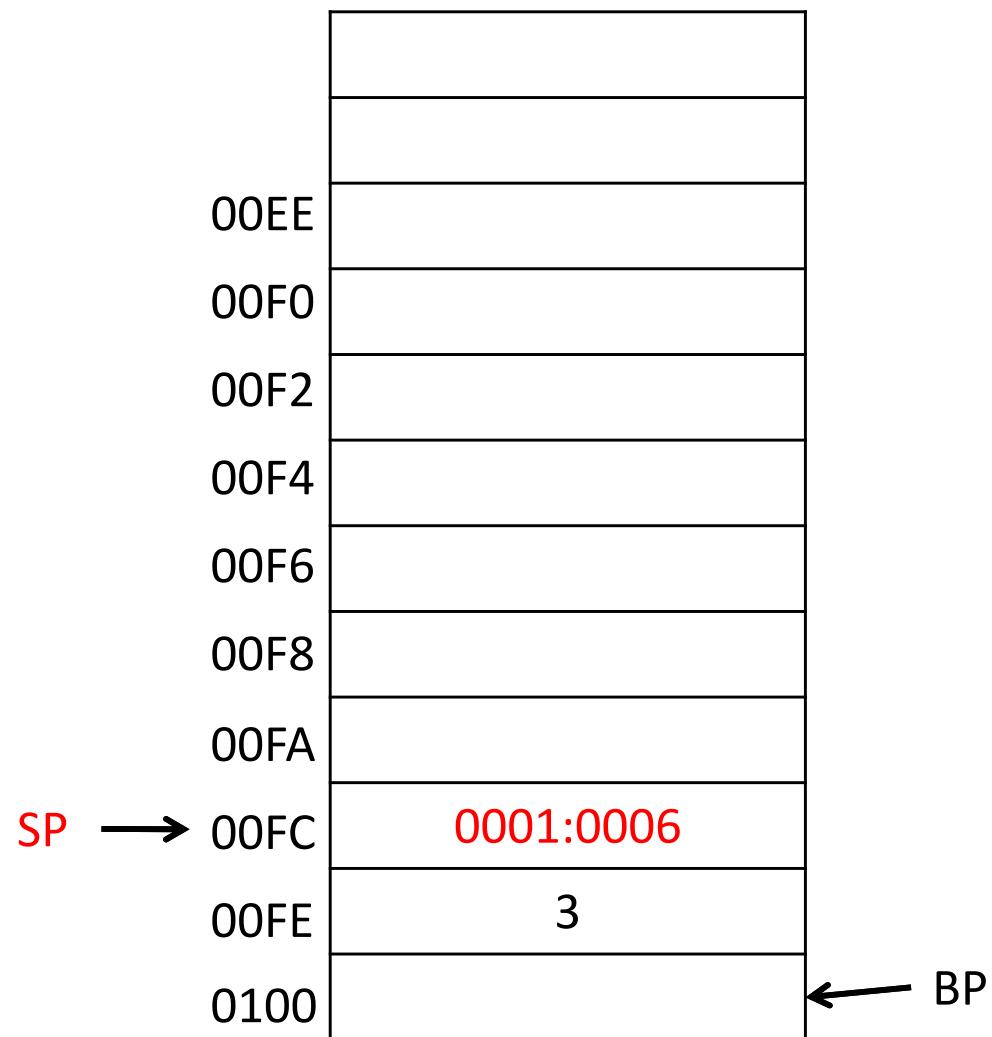
0000	MOV	AX,3
0002	PUSH	AX
0004	CALL	FACTORIAL
0006	MOV	AH,4CH
0008	INT	21H

AX	3
CX	
SP	00FC
BP	0100

MAIN ENDP

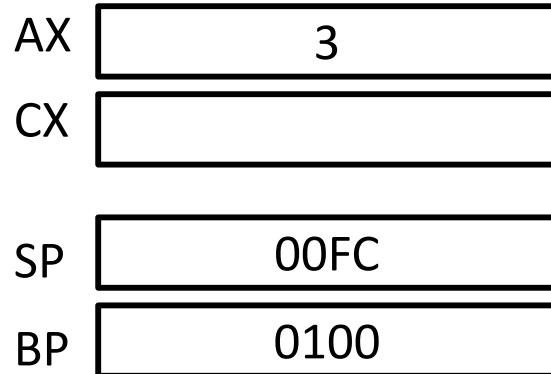
FACTORIAL PROC NEAR

000A	PUSH	BP
000C	MOV	BP,SP
000E	CMP	WORD PTR[BP+4],1
0010	JG	END_IF
0012	MOV	AX,1
0014	JMP	RETURN
0016	END_IF:	MOV CX, [BP+4]
0018	DEC	CX
001A	PUSH	CX
001C	CALL	FACTORIAL
001E	MUL	WORD PTR[BP+4]
0020	RETURN :	POP BP
0022	RET	2
	END	MAIN



MAIN PROC

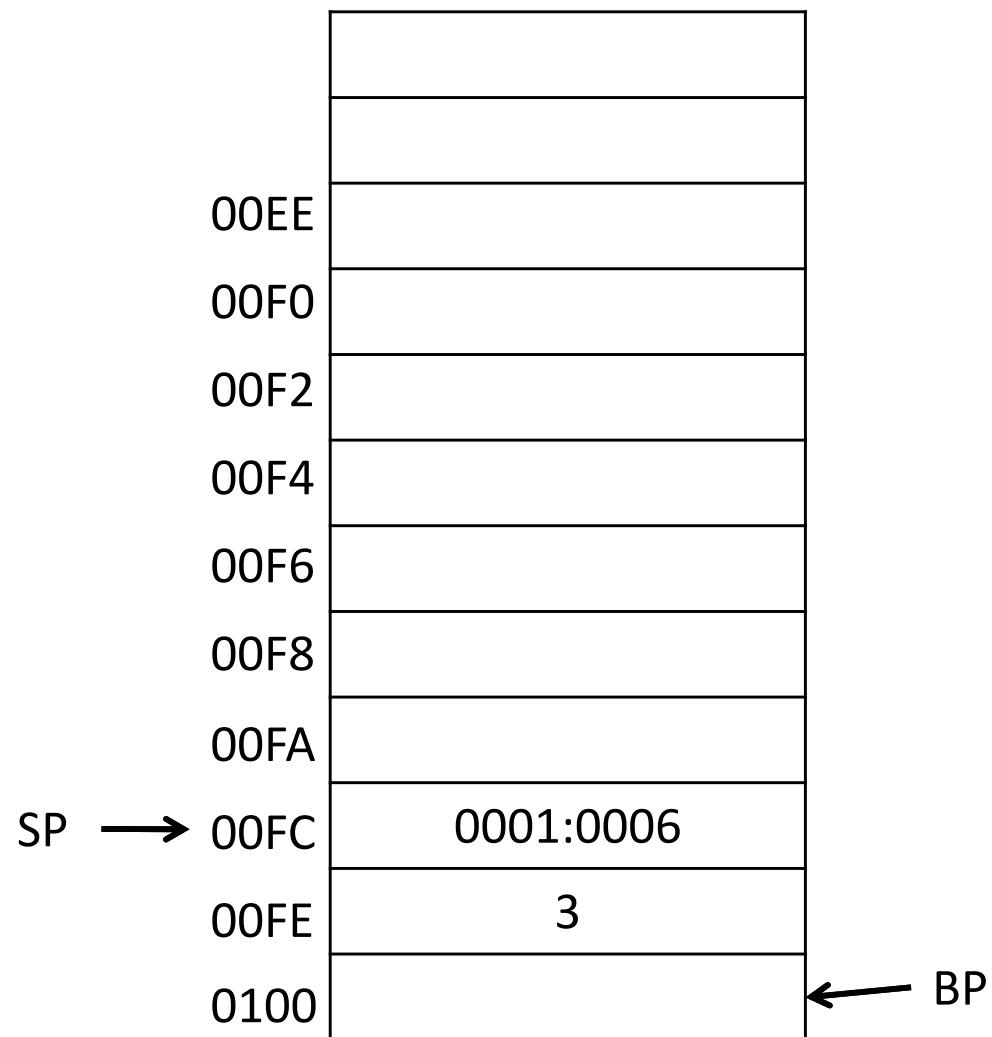
0000	MOV	AX,3
0002	PUSH	AX
0004	CALL	FACTORIAL
0006	MOV	AH,4CH
0008	INT	21H



MAIN ENDP

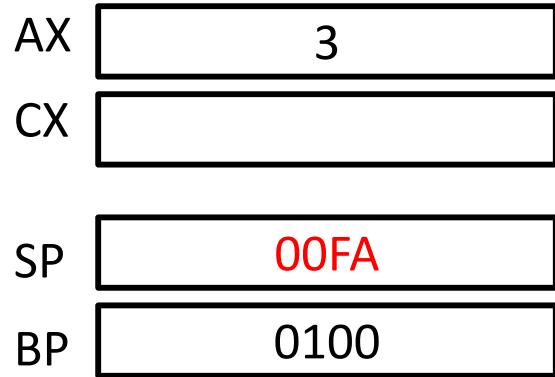
FACTORIAL PROC NEAR

000A	PUSH	BP
000C	MOV	BP,SP
000E	CMP	WORD PTR[BP+4],1
0010	JG	END_IF
0012	MOV	AX,1
0014	JMP	RETURN
0016	END_IF:	MOV CX, [BP+4]
0018	DEC	CX
001A	PUSH	CX
001C	CALL	FACTORIAL
001E	MUL	WORD PTR[BP+4]
0020	RETURN :	POP BP
0022	RET	2
	END	MAIN



MAIN PROC

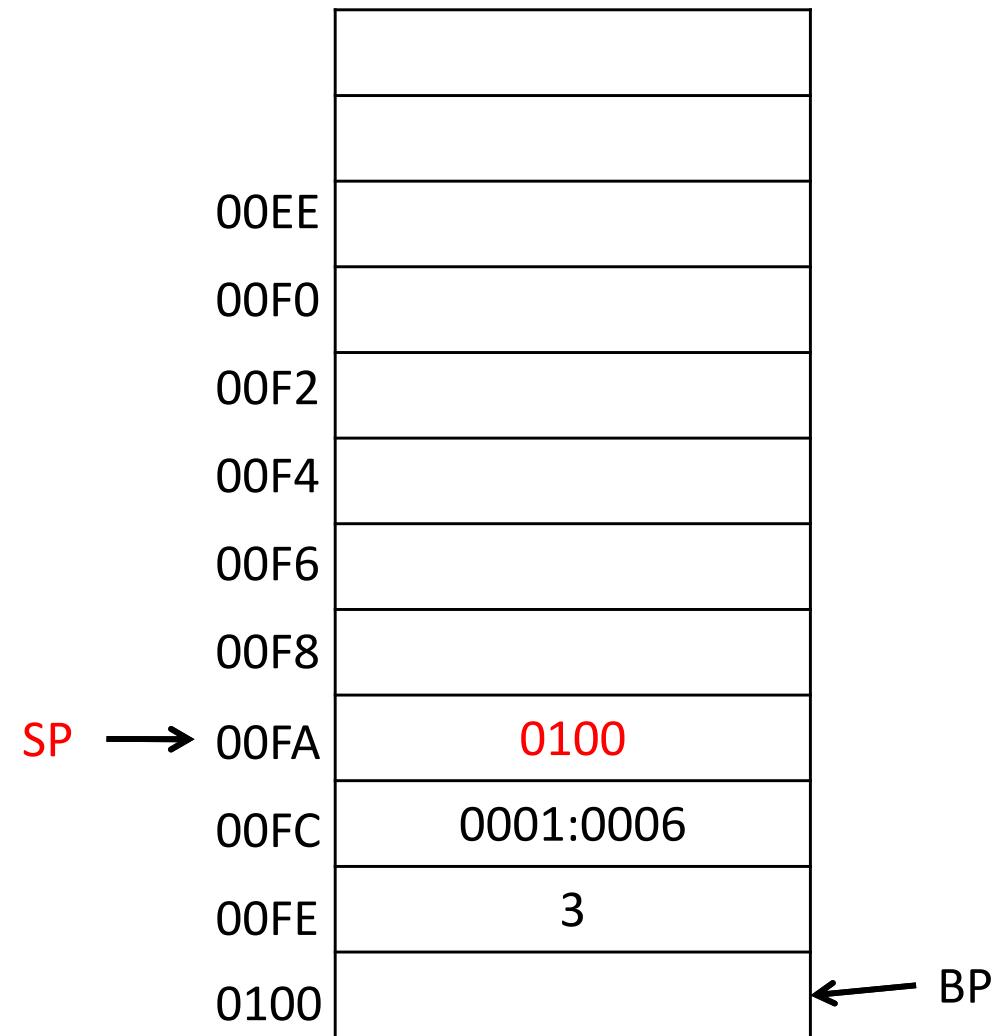
0000	MOV	AX,3
0002	PUSH	AX
0004	CALL	FACTORIAL
0006	MOV	AH,4CH
0008	INT	21H



MAIN ENDP

FACTORIAL PROC NEAR

000A	PUSH	BP
000C	MOV	BP,SP
000E	CMP	WORD PTR[BP+4],1
0010	JG	END_IF
0012	MOV	AX,1
0014	JMP	RETURN
0016	END_IF:	MOV CX, [BP+4]
0018	DEC	CX
001A	PUSH	CX
001C	CALL	FACTORIAL
001E	MUL	WORD PTR[BP+4]
0020	RETURN :	POP BP
0022	RET	2
	END	MAIN



MAIN PROC

0000	MOV	AX,3
0002	PUSH	AX
0004	CALL	FACTORIAL
0006	MOV	AH,4CH
0008	INT	21H

AX	3
CX	
SP	00FA
BP	0100

MAIN ENDP

FACTORIAL PROC NEAR

000A	PUSH	BP
000C	MOV	BP,SP
000E	CMP	WORD PTR[BP+4],1
0010	JG	END_IF
0012	MOV	AX,1
0014	JMP	RETURN
0016	END_IF:	MOV CX, [BP+4]
0018	DEC	CX
001A	PUSH	CX
001C	CALL	FACTORIAL
001E	MUL	WORD PTR[BP+4]
0020	RETURN :	POP BP
0022	END	RET 2
		MAIN

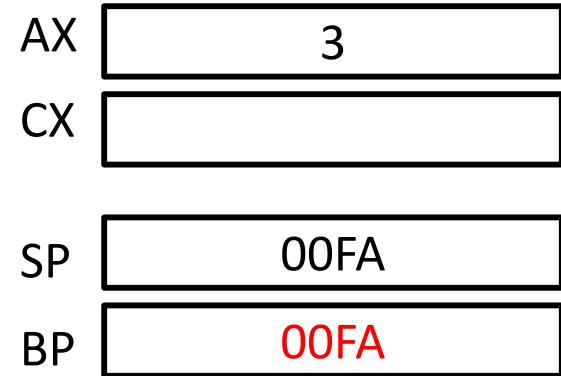
00EE	
00F0	
00F2	
00F4	
00F6	
00F8	
00FA	0100
00FC	0001:0006
00FE	3
0100	

SP →

← BP

MAIN PROC

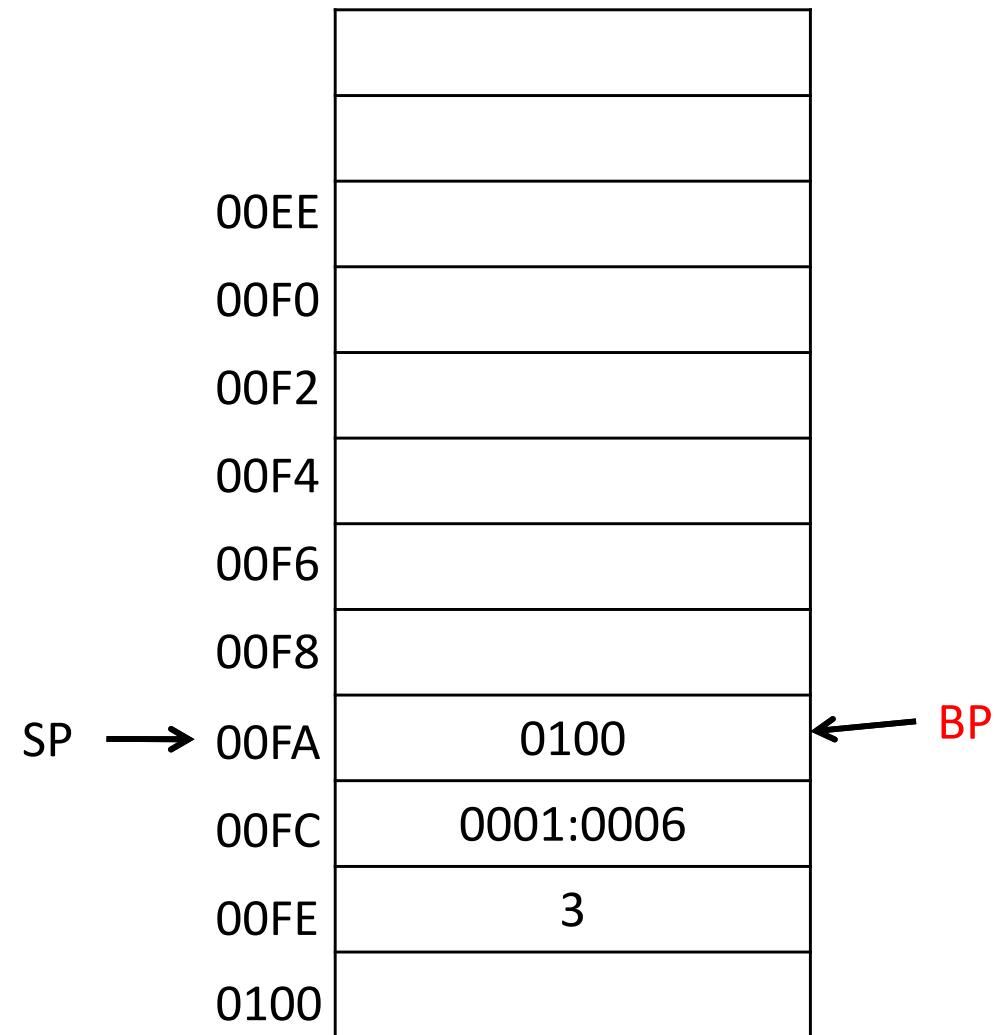
0000	MOV	AX,3
0002	PUSH	AX
0004	CALL	FACTORIAL
0006	MOV	AH,4CH
0008	INT	21H



MAIN ENDP

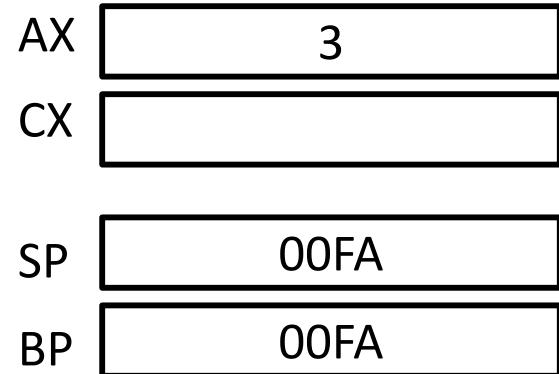
FACTORIAL PROC NEAR

000A	PUSH	BP
000C	MOV	BP,SP
000E	CMP	WORD PTR[BP+4],1
0010	JG	END_IF
0012	MOV	AX,1
0014	JMP	RETURN
0016	END_IF:	MOV CX, [BP+4]
0018	DEC	CX
001A	PUSH	CX
001C	CALL	FACTORIAL
001E	MUL	WORD PTR[BP+4]
0020	RETURN :	POP BP
0022	END	RET 2
		MAIN



MAIN PROC

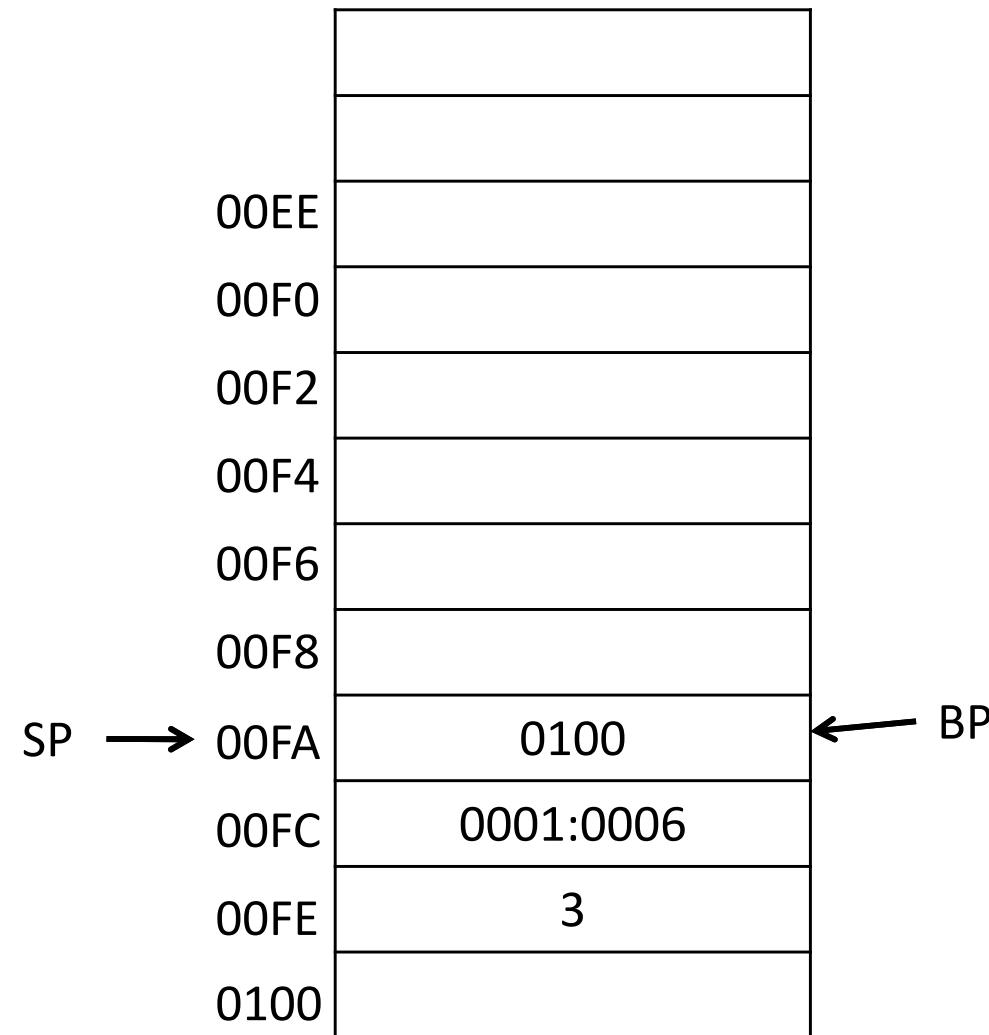
0000	MOV	AX,3
0002	PUSH	AX
0004	CALL	FACTORIAL
0006	MOV	AH,4CH
0008	INT	21H



MAIN ENDP

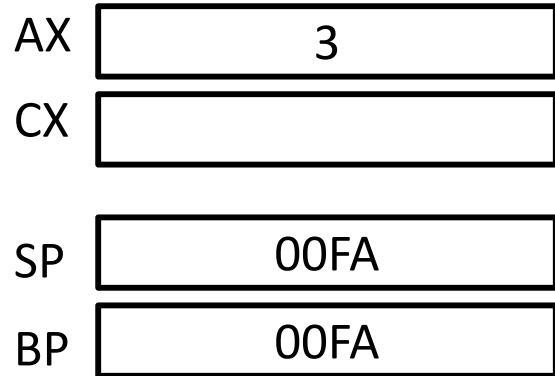
FACTORIAL PROC NEAR

000A	PUSH	BP
000C	MOV	BP,SP
000E	CMP	WORD PTR[BP+4],1
0010	JG	END_IF
0012	MOV	AX,1
0014	JMP	RETURN
0016	END_IF:	MOV CX, [BP+4]
0018	DEC	CX
001A	PUSH	CX
001C	CALL	FACTORIAL
001E	MUL	WORD PTR[BP+4]
0020	RETURN :	POP BP
0022	END	RET 2
		MAIN



MAIN PROC

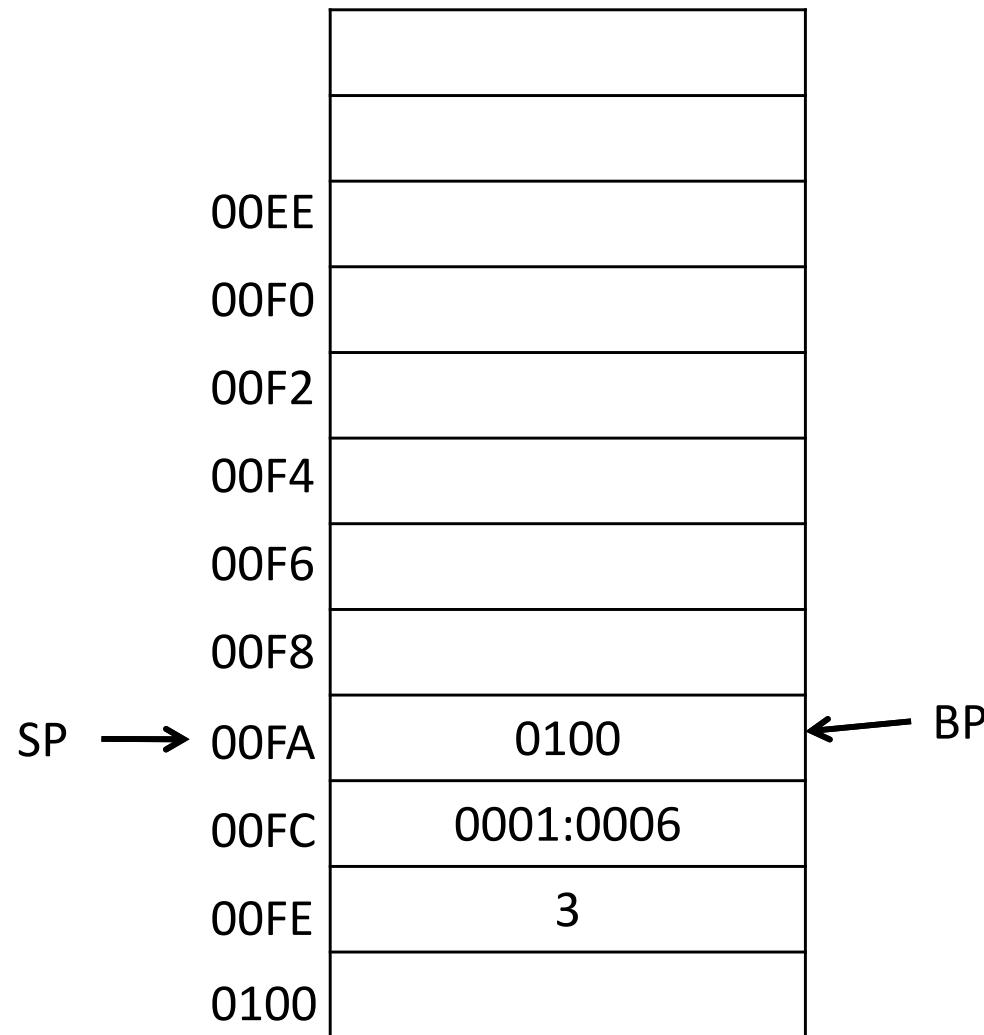
0000	MOV	AX,3
0002	PUSH	AX
0004	CALL	FACTORIAL
0006	MOV	AH,4CH
0008	INT	21H



MAIN ENDP

FACTORIAL PROC NEAR

000A	PUSH	BP
000C	MOV	BP,SP
000E	CMP	WORD PTR[BP+4],1
0010	JG	END_IF
0012	MOV	AX,1
0014	JMP	RETURN
0016	END_IF:	MOV CX, [BP+4]
0018	DEC	CX
001A	PUSH	CX
001C	CALL	FACTORIAL
001E	MUL	WORD PTR[BP+4]
0020	RETURN :	POP BP
0022	END	RET 2
		MAIN



MAIN PROC

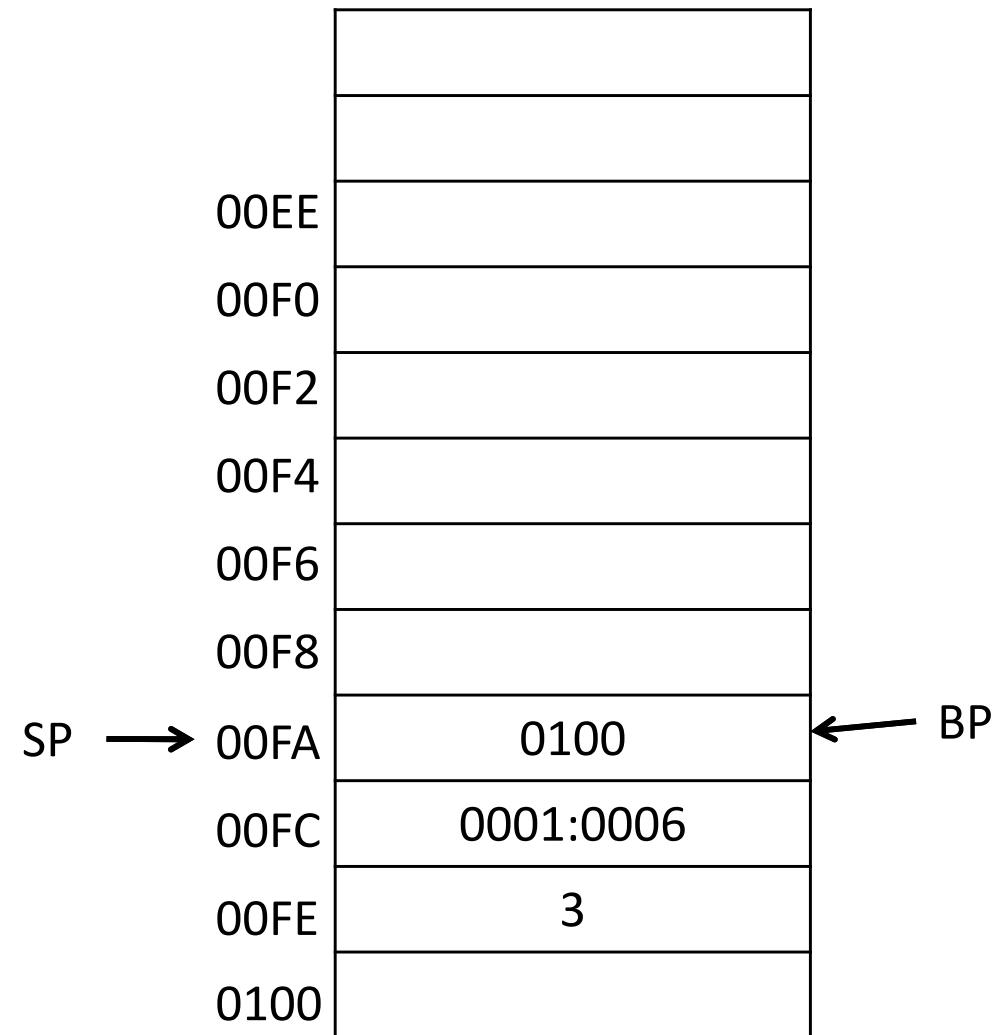
0000	MOV	AX,3
0002	PUSH	AX
0004	CALL	FACTORIAL
0006	MOV	AH,4CH
0008	INT	21H

AX	3
CX	
SP	00FA
BP	00FA

MAIN ENDP

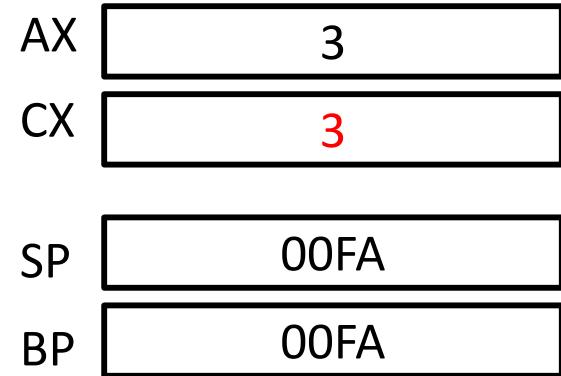
FACTORIAL PROC NEAR

000A	PUSH	BP
000C	MOV	BP,SP
000E	CMP	WORD PTR[BP+4],1
0010	JG	END_IF
0012	MOV	AX,1
0014	JMP	RETURN
0016	END_IF:	MOV CX, [BP+4]
0018	DEC	CX
001A	PUSH	CX
001C	CALL	FACTORIAL
001E	MUL	WORD PTR[BP+4]
0020	RETURN :	POP BP
0022	END	RET 2
		MAIN



MAIN PROC

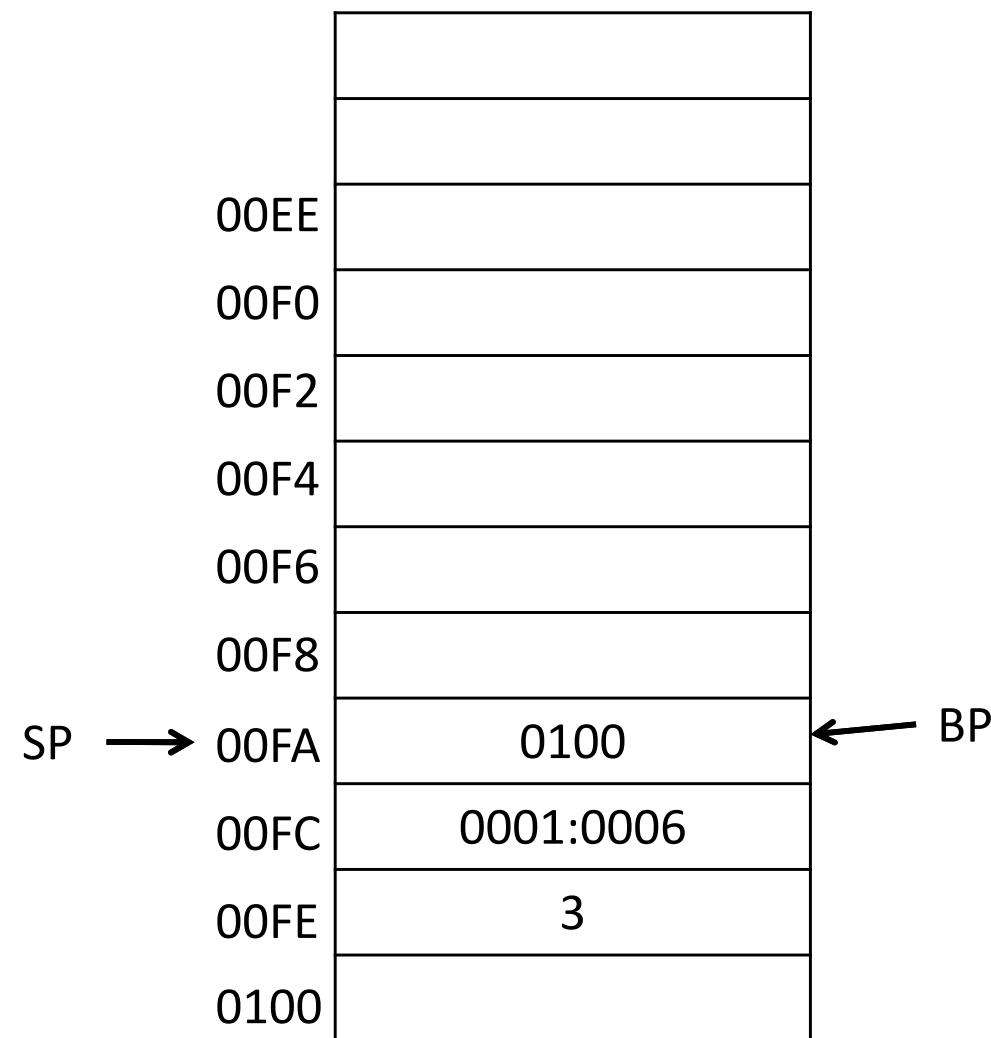
0000	MOV	AX,3
0002	PUSH	AX
0004	CALL	FACTORIAL
0006	MOV	AH,4CH
0008	INT	21H



MAIN ENDP

FACTORIAL PROC NEAR

000A	PUSH	BP
000C	MOV	BP,SP
000E	CMP	WORD PTR[BP+4],1
0010	JG	END_IF
0012	MOV	AX,1
0014	JMP	RETURN
0016	END_IF:	MOV CX, [BP+4]
0018	DEC	CX
001A	PUSH	CX
001C	CALL	FACTORIAL
001E	MUL	WORD PTR[BP+4]
0020	RETURN :	POP BP
0022		RET 2
	END	MAIN



MAIN PROC

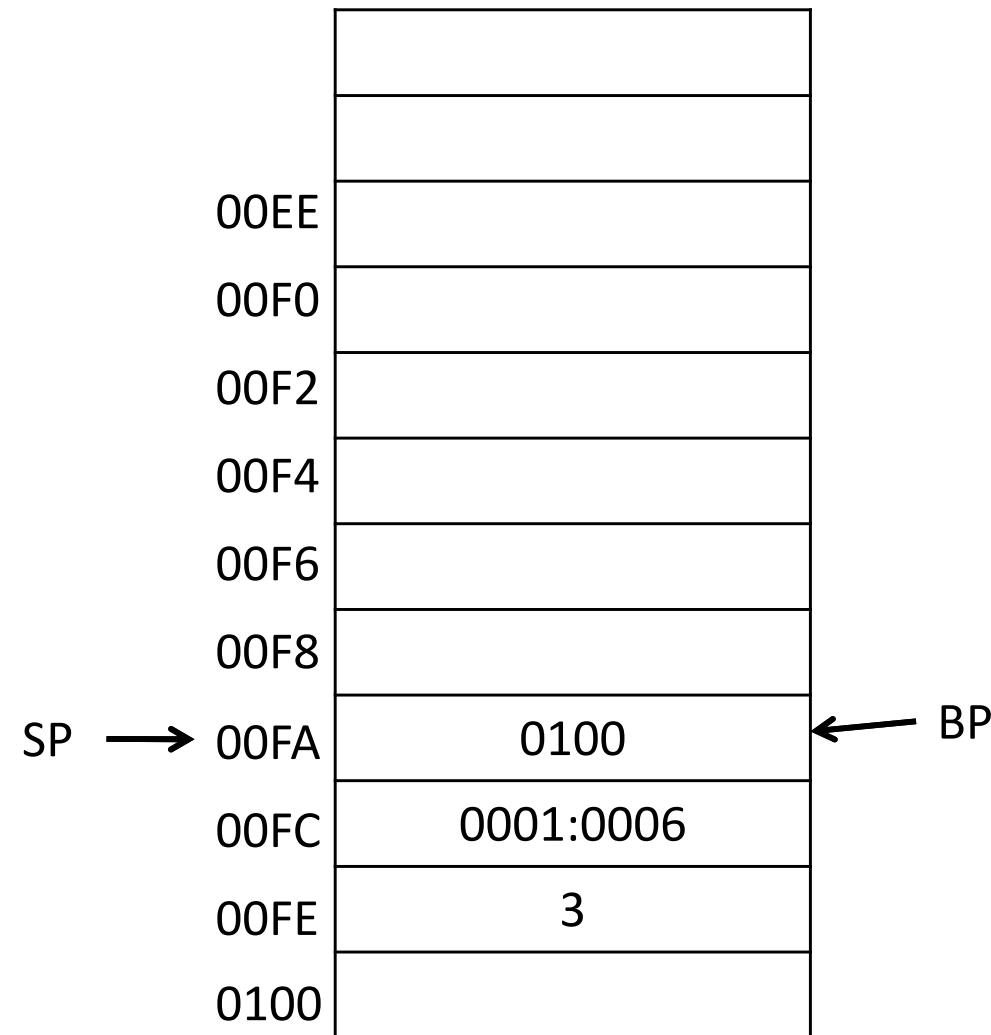
0000	MOV	AX,3
0002	PUSH	AX
0004	CALL	FACTORIAL
0006	MOV	AH,4CH
0008	INT	21H

AX	3
CX	3
SP	00FA
BP	00FA

MAIN ENDP

FACTORIAL PROC NEAR

000A	PUSH	BP
000C	MOV	BP,SP
000E	CMP	WORD PTR[BP+4],1
0010	JG	END_IF
0012	MOV	AX,1
0014	JMP	RETURN
0016	END_IF:	MOV CX, [BP+4]
0018	DEC	CX
001A	PUSH	CX
001C	CALL	FACTORIAL
001E	MUL	WORD PTR[BP+4]
0020	RETURN :	POP BP
0022	END	RET 2
		MAIN



MAIN PROC

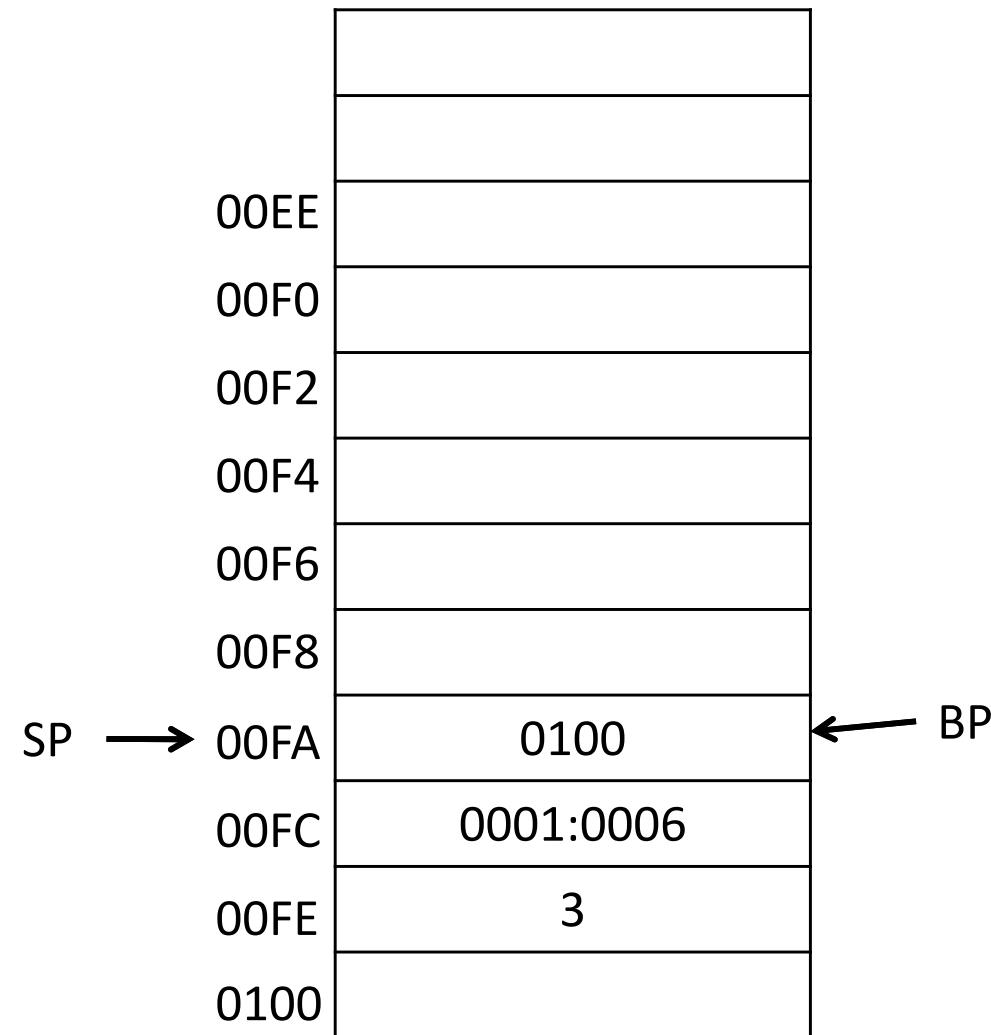
0000	MOV	AX,3
0002	PUSH	AX
0004	CALL	FACTORIAL
0006	MOV	AH,4CH
0008	INT	21H

AX	3
CX	2
SP	00FA
BP	00FA

MAIN ENDP

FACTORIAL PROC NEAR

000A	PUSH	BP
000C	MOV	BP,SP
000E	CMP	WORD PTR[BP+4],1
0010	JG	END_IF
0012	MOV	AX,1
0014	JMP	RETURN
0016	END_IF:	MOV CX, [BP+4]
0018	DEC	CX
001A	PUSH	CX
001C	CALL	FACTORIAL
001E	MUL	WORD PTR[BP+4]
0020	RETURN :	POP BP
0022	END	RET 2
		MAIN



MAIN PROC

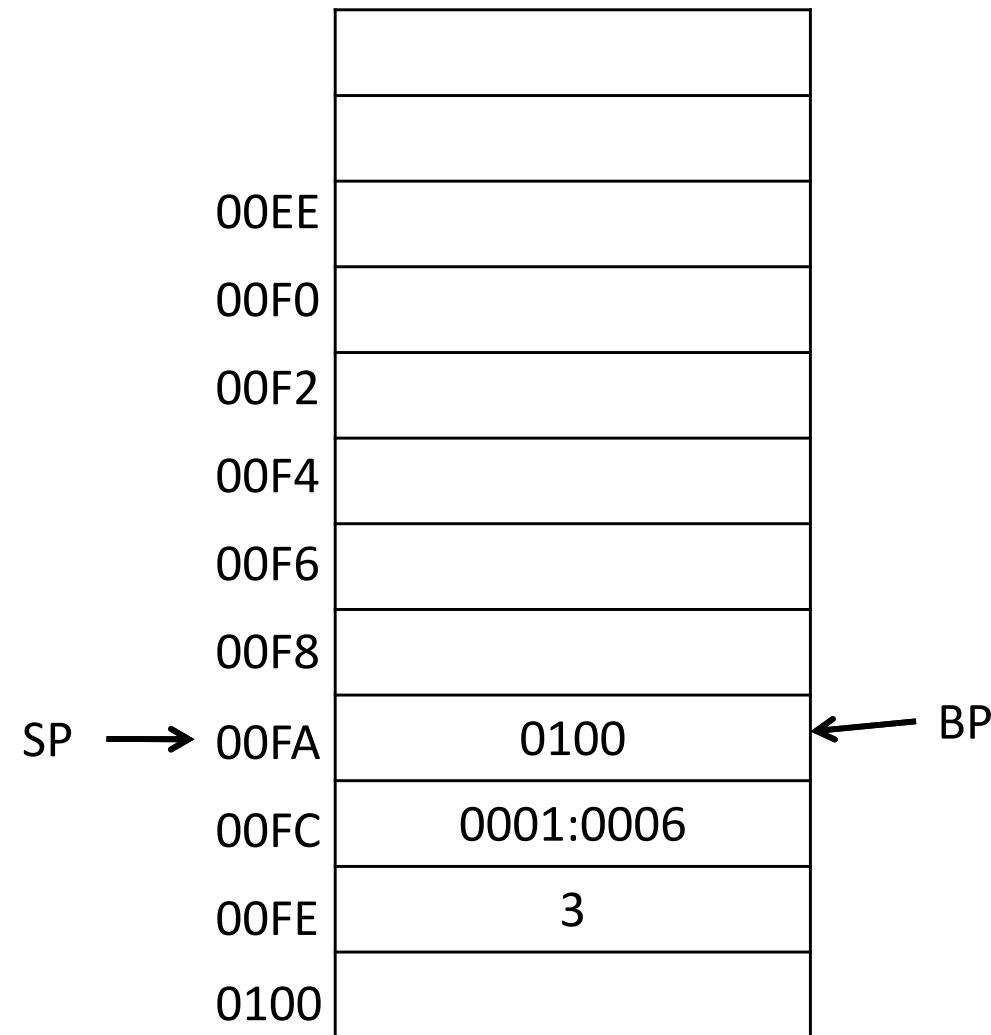
0000	MOV	AX,3
0002	PUSH	AX
0004	CALL	FACTORIAL
0006	MOV	AH,4CH
0008	INT	21H

AX	3
CX	2
SP	00FA
BP	00FA

MAIN ENDP

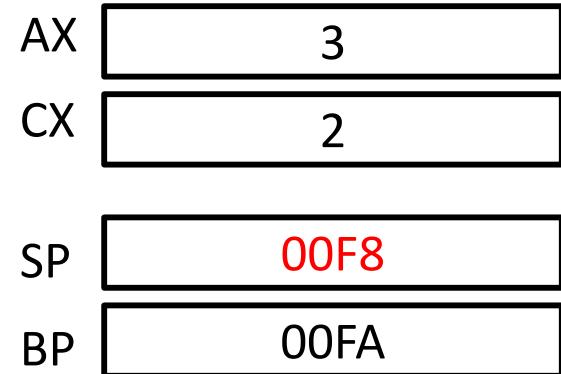
FACTORIAL PROC NEAR

000A	PUSH	BP
000C	MOV	BP,SP
000E	CMP	WORD PTR[BP+4],1
0010	JG	END_IF
0012	MOV	AX,1
0014	JMP	RETURN
0016	END_IF:	MOV CX, [BP+4]
0018	DEC	CX
001A	PUSH	CX
001C	CALL	FACTORIAL
001E	MUL	WORD PTR[BP+4]
0020	RETURN :	POP BP
0022	END	RET 2
		MAIN



MAIN PROC

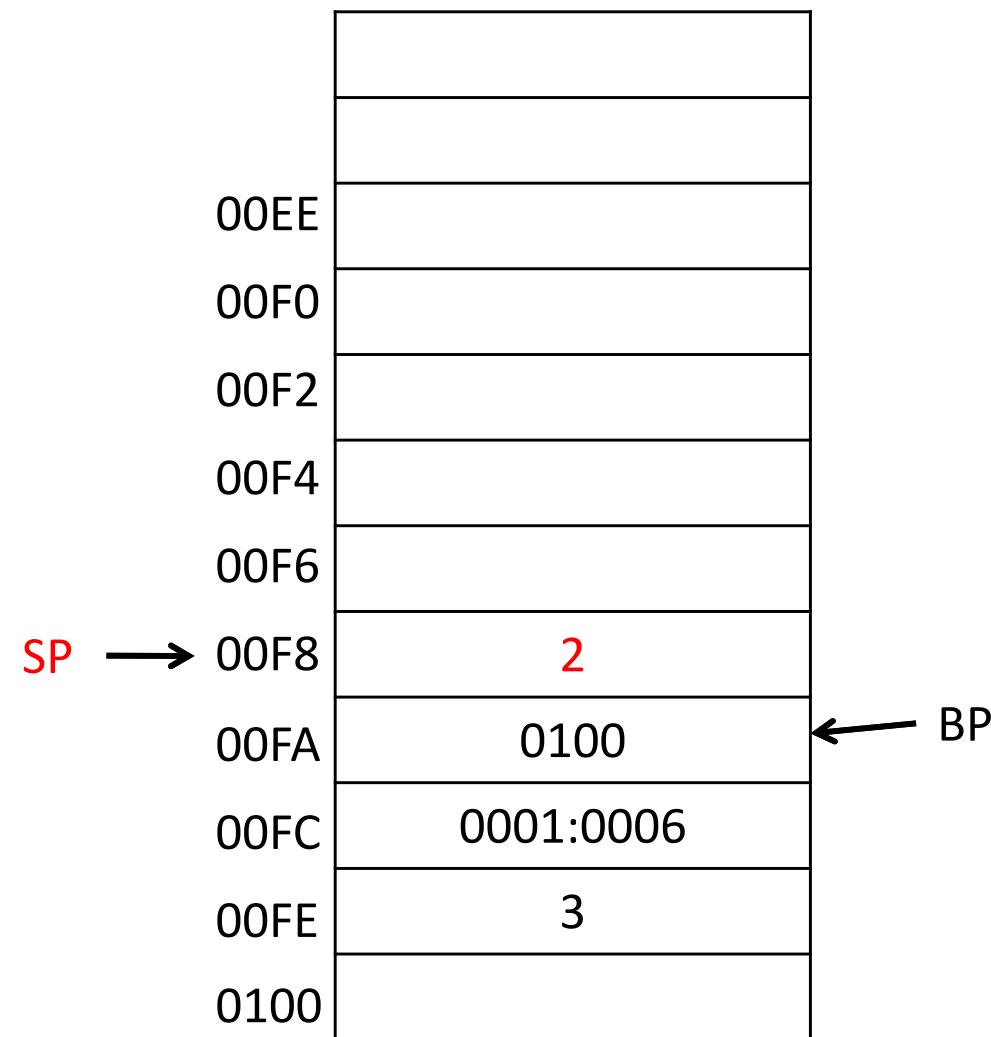
0000	MOV	AX,3
0002	PUSH	AX
0004	CALL	FACTORIAL
0006	MOV	AH,4CH
0008	INT	21H



MAIN ENDP

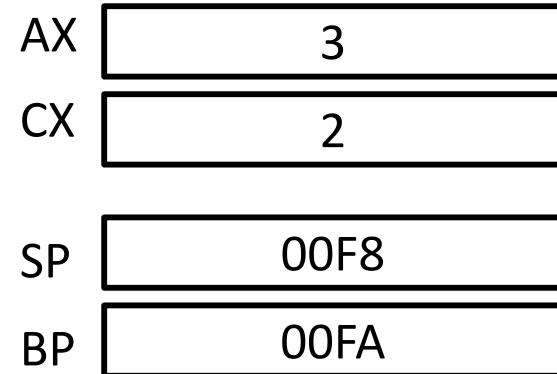
FACTORIAL PROC NEAR

000A	PUSH	BP
000C	MOV	BP,SP
000E	CMP	WORD PTR[BP+4],1
0010	JG	END_IF
0012	MOV	AX,1
0014	JMP	RETURN
0016	END_IF:	MOV CX, [BP+4]
0018	DEC	CX
001A	PUSH	CX
001C	CALL	FACTORIAL
001E	MUL	WORD PTR[BP+4]
0020	RETURN :	POP BP
0022	RET	2
	END	MAIN



MAIN PROC

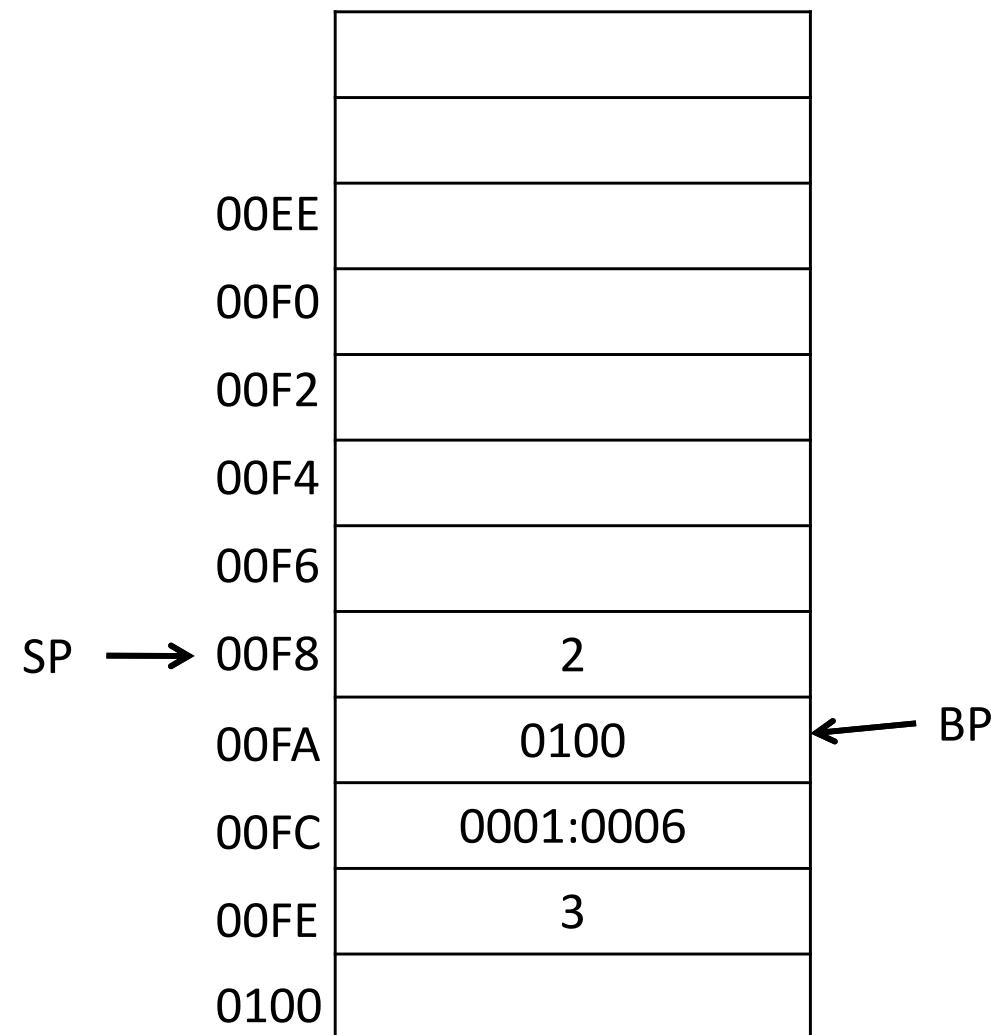
0000	MOV	AX,3
0002	PUSH	AX
0004	CALL	FACTORIAL
0006	MOV	AH,4CH
0008	INT	21H



MAIN ENDP

FACTORIAL PROC NEAR

000A	PUSH	BP
000C	MOV	BP,SP
000E	CMP	WORD PTR[BP+4],1
0010	JG	END_IF
0012	MOV	AX,1
0014	JMP	RETURN
0016	END_IF:	MOV CX, [BP+4]
0018	DEC	CX
001A	PUSH	CX
001C	CALL	FACTORIAL
001E	MUL	WORD PTR[BP+4]
0020	RETURN :	POP BP
0022		RET 2
	END	MAIN



MAIN PROC

0000	MOV	AX,3
0002	PUSH	AX
0004	CALL	FACTORIAL
0006	MOV	AH,4CH
0008	INT	21H

AX	3
CX	2
SP	00F6
BP	00FA

MAIN ENDP

FACTORIAL PROC NEAR

000A	PUSH	BP
000C	MOV	BP,SP
000E	CMP	WORD PTR[BP+4],1
0010	JG	END_IF
0012	MOV	AX,1
0014	JMP	RETURN
0016	END_IF:	MOV CX, [BP+4]
0018	DEC	CX
001A	PUSH	CX
001C	CALL	FACTORIAL
001E	MUL	WORD PTR[BP+4]
0020	RETURN :	POP BP
0022		RET 2
	END	MAIN

00EE	
00F0	
00F2	
00F4	
00F6	0001:001E
00F8	2
00FA	0100
00FC	0001:0006
00FE	3
0100	

SP → 00F6 0001:001E

BP ←

MAIN PROC

0000	MOV	AX,3
0002	PUSH	AX
0004	CALL	FACTORIAL
0006	MOV	AH,4CH
0008	INT	21H

AX	3
CX	2
SP	00F6
BP	00FA

MAIN ENDP

FACTORIAL PROC NEAR

000A	PUSH	BP
000C	MOV	BP,SP
000E	CMP	WORD PTR[BP+4],1
0010	JG	END_IF
0012	MOV	AX,1
0014	JMP	RETURN
0016	END_IF:	MOV CX, [BP+4]
0018	DEC	CX
001A	PUSH	CX
001C	CALL	FACTORIAL
001E	MUL	WORD PTR[BP+4]
0020	RETURN :	POP BP
0022	RET	2
	END	MAIN

00EE	
00F0	
00F2	
00F4	
00F6	0001:001E
00F8	2
00FA	0100
00FC	0001:0006
00FE	3
0100	

SP → 00F6 BP ←

MAIN PROC

0000	MOV	AX,3
0002	PUSH	AX
0004	CALL	FACTORIAL
0006	MOV	AH,4CH
0008	INT	21H

AX	3
CX	2
SP	00F4
BP	00FA

MAIN ENDP

FACTORIAL PROC NEAR

000A	PUSH	BP
000C	MOV	BP,SP
000E	CMP	WORD PTR[BP+4],1
0010	JG	END_IF
0012	MOV	AX,1
0014	JMP	RETURN
0016	END_IF:	MOV CX, [BP+4]
0018	DEC	CX
001A	PUSH	CX
001C	CALL	FACTORIAL
001E	MUL	WORD PTR[BP+4]
0020	RETURN :	POP BP
0022	END	RET 2
		MAIN

00EE	
00F0	
00F2	
00F4	00FA
00F6	0001:001E
00F8	2
00FA	0100
00FC	0001:0006
00FE	3
0100	

SP →

BP ←

MAIN PROC

0000	MOV	AX,3
0002	PUSH	AX
0004	CALL	FACTORIAL
0006	MOV	AH,4CH
0008	INT	21H

AX	3
CX	2
SP	00F4
BP	00FA

MAIN ENDP

FACTORIAL PROC NEAR

000A	PUSH	BP
000C	MOV	BP,SP
000E	CMP	WORD PTR[BP+4],1
0010	JG	END_IF
0012	MOV	AX,1
0014	JMP	RETURN
0016	END_IF:	MOV CX, [BP+4]
0018	DEC	CX
001A	PUSH	CX
001C	CALL	FACTORIAL
001E	MUL	WORD PTR[BP+4]
0020	RETURN :	POP BP
0022		RET 2
	END	MAIN

00EE	
00F0	
00F2	
00F4	00FA
00F6	0001:001E
00F8	2
00FA	0100
00FC	0001:0006
00FE	3
0100	

SP → 00F4

BP ← 00FA

MAIN PROC

0000	MOV	AX,3
0002	PUSH	AX
0004	CALL	FACTORIAL
0006	MOV	AH,4CH
0008	INT	21H

AX	3
CX	2
SP	00F4
BP	00F4

MAIN ENDP

FACTORIAL PROC NEAR

000A	PUSH	BP
000C	MOV	BP,SP
000E	CMP	WORD PTR[BP+4],1
0010	JG	END_IF
0012	MOV	AX,1
0014	JMP	RETURN
0016	END_IF:	MOV CX, [BP+4]
0018	DEC	CX
001A	PUSH	CX
001C	CALL	FACTORIAL
001E	MUL	WORD PTR[BP+4]
0020	RETURN :	POP BP
0022		RET 2
	END	MAIN

00EE	
00F0	
00F2	
00F4	00FA
00F6	0001:001E
00F8	2
00FA	0100
00FC	0001:0006
00FE	3
0100	

SP → 00F4

BP ← 00FA

MAIN PROC

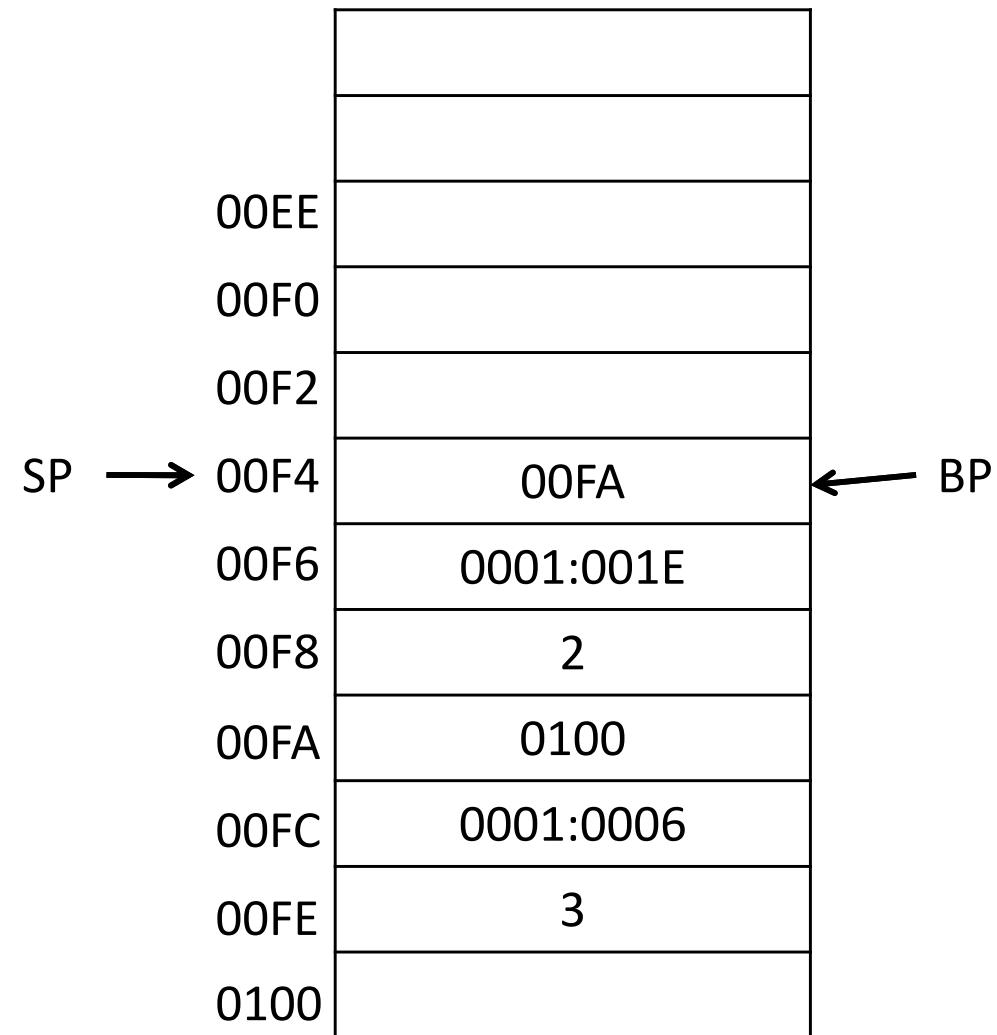
0000	MOV	AX,3
0002	PUSH	AX
0004	CALL	FACTORIAL
0006	MOV	AH,4CH
0008	INT	21H

AX	3
CX	2
SP	00F4
BP	00F4

MAIN ENDP

FACTORIAL PROC NEAR

000A	PUSH	BP
000C	MOV	BP,SP
000E	CMP	WORD PTR[BP+4],1
0010	JG	END_IF
0012	MOV	AX,1
0014	JMP	RETURN
0016	END_IF:	MOV CX, [BP+4]
0018	DEC	CX
001A	PUSH	CX
001C	CALL	FACTORIAL
001E	MUL	WORD PTR[BP+4]
0020	RETURN :	POP BP
0022		RET 2
	END	MAIN



MAIN PROC

0000	MOV	AX,3
0002	PUSH	AX
0004	CALL	FACTORIAL
0006	MOV	AH,4CH
0008	INT	21H

AX	3
CX	2
SP	00F4
BP	00F4

MAIN ENDP

FACTORIAL PROC NEAR

000A	PUSH	BP
000C	MOV	BP,SP
000E	CMP	WORD PTR[BP+4],1
0010	JG	END_IF
0012	MOV	AX,1
0014	JMP	RETURN
0016	END_IF:	MOV CX, [BP+4]
0018	DEC	CX
001A	PUSH	CX
001C	CALL	FACTORIAL
001E	MUL	WORD PTR[BP+4]
0020	RETURN :	POP BP
0022		RET 2
	END	MAIN

SP → 00F4 ← BP

00EE	
00F0	
00F2	
00F4	00FA
00F6	0001:001E
00F8	2
00FA	0100
00FC	0001:0006
00FE	3
0100	

MAIN PROC

0000	MOV	AX,3
0002	PUSH	AX
0004	CALL	FACTORIAL
0006	MOV	AH,4CH
0008	INT	21H

AX	3
CX	2
SP	00F4
BP	00F4

MAIN ENDP

FACTORIAL PROC NEAR

000A	PUSH	BP
000C	MOV	BP,SP
000E	CMP	WORD PTR[BP+4],1
0010	JG	END_IF
0012	MOV	AX,1
0014	JMP	RETURN
0016	END_IF:	MOV CX, [BP+4]
0018	DEC	CX
001A	PUSH	CX
001C	CALL	FACTORIAL
001E	MUL	WORD PTR[BP+4]
0020	RETURN :	POP BP
0022		RET 2
	END	MAIN

SP → 00F4 ← BP

00EE	
00F0	
00F2	
00F4	00FA
00F6	0001:001E
00F8	2
00FA	0100
00FC	0001:0006
00FE	3
0100	

MAIN PROC

0000	MOV	AX,3
0002	PUSH	AX
0004	CALL	FACTORIAL
0006	MOV	AH,4CH
0008	INT	21H

AX	3
CX	2
SP	00F4
BP	00F4

MAIN ENDP

FACTORIAL PROC NEAR

000A	PUSH	BP
000C	MOV	BP,SP
000E	CMP	WORD PTR[BP+4],1
0010	JG	END_IF
0012	MOV	AX,1
0014	JMP	RETURN
0016	END_IF:	MOV CX, [BP+4]
0018	DEC	CX
001A	PUSH	CX
001C	CALL	FACTORIAL
001E	MUL	WORD PTR[BP+4]
0020	RETURN :	POP BP
0022		RET 2
	END	MAIN

00EE	
00F0	
00F2	
00F4	00FA
00F6	0001:001E
00F8	2
00FA	0100
00FC	0001:0006
00FE	3
0100	

SP → 00F4 ← BP

MAIN PROC

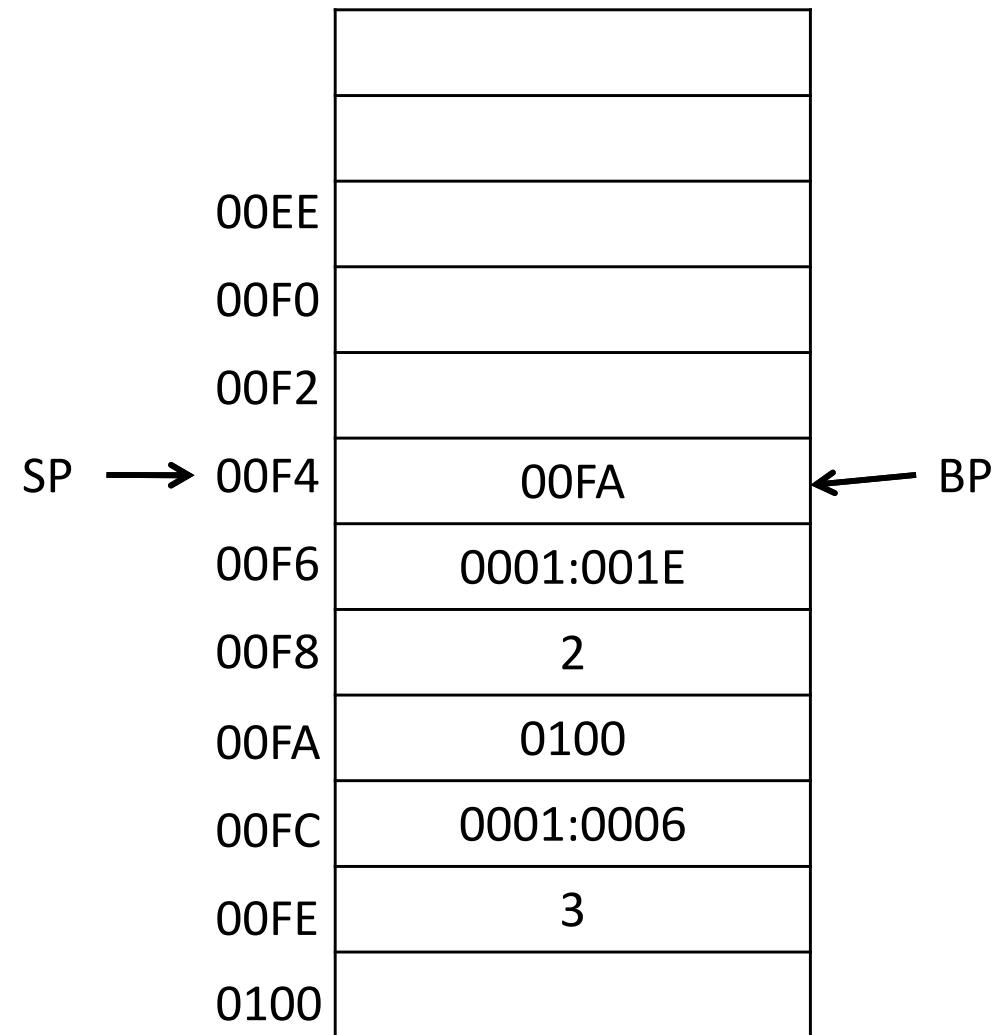
0000	MOV	AX,3
0002	PUSH	AX
0004	CALL	FACTORIAL
0006	MOV	AH,4CH
0008	INT	21H

AX	3
CX	2
SP	00F4
BP	00F4

MAIN ENDP

FACTORIAL PROC NEAR

000A	PUSH	BP
000C	MOV	BP,SP
000E	CMP	WORD PTR[BP+4],1
0010	JG	END_IF
0012	MOV	AX,1
0014	JMP	RETURN
0016	END_IF:	MOV CX, [BP+4]
0018	DEC	CX
001A	PUSH	CX
001C	CALL	FACTORIAL
001E	MUL	WORD PTR[BP+4]
0020	RETURN :	POP BP
0022		RET 2
	END	MAIN



MAIN PROC

0000	MOV	AX,3	
0002	PUSH	AX	
0004	CALL	FACTORIAL	
0006	MOV	AH,4CH	
0008	INT	21H	

AX	3
CX	1
SP	00F4
BP	00F4

MAIN ENDP

FACTORIAL PROC NEAR

000A	PUSH	BP	
000C	MOV	BP,SP	
000E	CMP	WORD PTR[BP+4],1	
0010	JG	END_IF	
0012	MOV	AX,1	
0014	JMP	RETURN	
0016	END_IF:	MOV CX, [BP+4]	
0018	DEC	CX	
001A	PUSH	CX	
001C	CALL	FACTORIAL	
001E	MUL	WORD PTR[BP+4]	
0020	RETURN :	POP BP	
0022		RET 2	
	END	MAIN	

00EE	
00F0	
00F2	
00F4	00FA
00F6	0001:001E
00F8	2
00FA	0100
00FC	0001:0006
00FE	3
0100	

SP → 00F4 BP ← 00FA

MAIN PROC

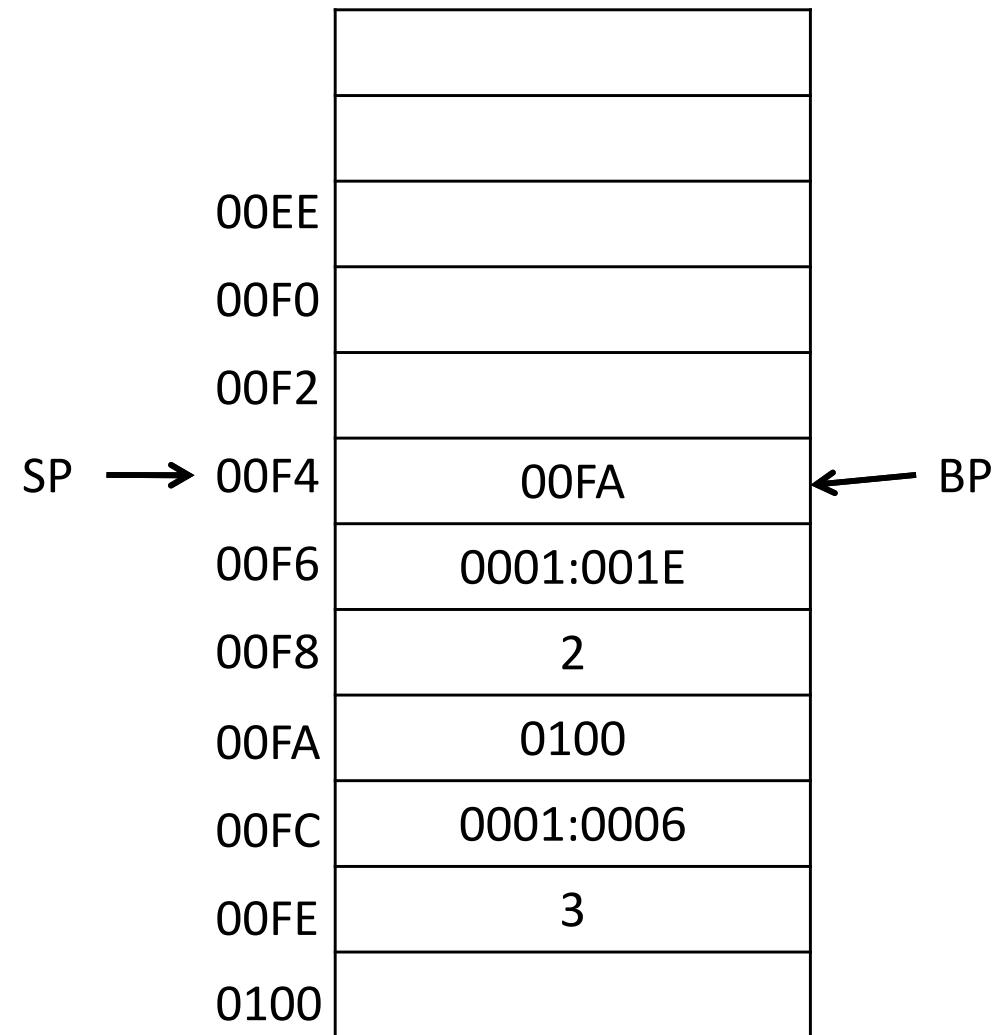
0000	MOV	AX,3	
0002	PUSH	AX	
0004	CALL	FACTORIAL	
0006	MOV	AH,4CH	
0008	INT	21H	

AX	3
CX	1
SP	00F4
BP	00F4

MAIN ENDP

FACTORIAL PROC NEAR

000A	PUSH	BP	
000C	MOV	BP,SP	
000E	CMP	WORD PTR[BP+4],1	
0010	JG	END_IF	
0012	MOV	AX,1	
0014	JMP	RETURN	
0016	END_IF:	MOV CX, [BP+4]	
0018	DEC	CX	
001A	PUSH	CX	
001C	CALL	FACTORIAL	
001E	MUL	WORD PTR[BP+4]	
0020	RETURN :	POP BP	
0022		RET 2	
	END	MAIN	



MAIN PROC

0000	MOV	AX,3
0002	PUSH	AX
0004	CALL	FACTORIAL
0006	MOV	AH,4CH
0008	INT	21H

AX	3
CX	1
SP	00F2
BP	00F4

MAIN ENDP

FACTORIAL PROC NEAR

000A	PUSH	BP
000C	MOV	BP,SP
000E	CMP	WORD PTR[BP+4],1
0010	JG	END_IF
0012	MOV	AX,1
0014	JMP	RETURN
0016	END_IF:	MOV CX, [BP+4]
0018	DEC	CX
001A	PUSH	CX
001C	CALL	FACTORIAL
001E	MUL	WORD PTR[BP+4]
0020	RETURN :	POP BP
0022		RET 2
	END	MAIN

00EE	
00FO	
00F2	1
00F4	00FA
00F6	0001:001E
00F8	2
00FA	0100
00FC	0001:0006
00FE	3
0100	

SP → 00F2

BP ←

MAIN PROC

0000	MOV	AX,3
0002	PUSH	AX
0004	CALL	FACTORIAL
0006	MOV	AH,4CH
0008	INT	21H

AX	3
CX	1
SP	00F2
BP	00F4

MAIN ENDP

FACTORIAL PROC NEAR

000A	PUSH	BP
000C	MOV	BP,SP
000E	CMP	WORD PTR[BP+4],1
0010	JG	END_IF
0012	MOV	AX,1
0014	JMP	RETURN
0016	END_IF:	MOV CX, [BP+4]
0018	DEC	CX
001A	PUSH	CX
001C	CALL	FACTORIAL
001E	MUL	WORD PTR[BP+4]
0020	RETURN :	POP BP
0022		RET 2
	END	MAIN

00EE	
00FO	
00F2	1
00F4	00FA
00F6	0001:001E
00F8	2
00FA	0100
00FC	0001:0006
00FE	3
0100	

SP → 00F2

BP ← 00FA

MAIN PROC

0000	MOV	AX,3
0002	PUSH	AX
0004	CALL	FACTORIAL
0006	MOV	AH,4CH
0008	INT	21H

AX	3
CX	1
SP	00F0
BP	00F4

MAIN ENDP

FACTORIAL PROC NEAR

000A	PUSH	BP
000C	MOV	BP,SP
000E	CMP	WORD PTR[BP+4],1
0010	JG	END_IF
0012	MOV	AX,1
0014	JMP	RETURN
0016	END_IF:	MOV CX, [BP+4]
0018	DEC	CX
001A	PUSH	CX
001C	CALL	FACTORIAL
001E	MUL	WORD PTR[BP+4]
0020	RETURN :	POP BP
0022		RET 2
	END	MAIN

SP → 00F0 0001:001E

BP ← 00F4 00FA

00EE	
00F0	0001:001E
00F2	1
00F4	00FA
00F6	0001:001E
00F8	2
00FA	0100
00FC	0001:0006
00FE	3
0100	

MAIN PROC

0000	MOV	AX,3
0002	PUSH	AX
0004	CALL	FACTORIAL
0006	MOV	AH,4CH
0008	INT	21H

AX	3
CX	1
SP	00F0
BP	00F4

MAIN ENDP

FACTORIAL PROC NEAR

000A	PUSH	BP
000C	MOV	BP,SP
000E	CMP	WORD PTR[BP+4],1
0010	JG	END_IF
0012	MOV	AX,1
0014	JMP	RETURN
0016	END_IF:	MOV CX, [BP+4]
0018	DEC	CX
001A	PUSH	CX
001C	CALL	FACTORIAL
001E	MUL	WORD PTR[BP+4]
0020	RETURN :	POP BP
0022	END	RET 2
		MAIN

00EE	
00F0	0001:001E
00F2	1
00F4	00FA
00F6	0001:001E
00F8	2
00FA	0100
00FC	0001:0006
00FE	3
0100	

SP → 00F0

← BP

MAIN PROC

0000	MOV	AX,3
0002	PUSH	AX
0004	CALL	FACTORIAL
0006	MOV	AH,4CH
0008	INT	21H

AX	3
CX	1
SP	00EE
BP	00F4

MAIN ENDP

FACTORIAL PROC NEAR

000A	PUSH	BP
000C	MOV	BP,SP
000E	CMP	WORD PTR[BP+4],1
0010	JG	END_IF
0012	MOV	AX,1
0014	JMP	RETURN
0016	END_IF:	MOV CX, [BP+4]
0018	DEC	CX
001A	PUSH	CX
001C	CALL	FACTORIAL
001E	MUL	WORD PTR[BP+4]
0020	RETURN :	POP BP
0022	END	RET 2
		MAIN

SP → 00EE 00F4

00F0	0001:001E
00F2	1
00F4	00FA
00F6	0001:001E
00F8	2
00FA	0100
00FC	0001:0006
00FE	3
0100	

BP ←

MAIN PROC

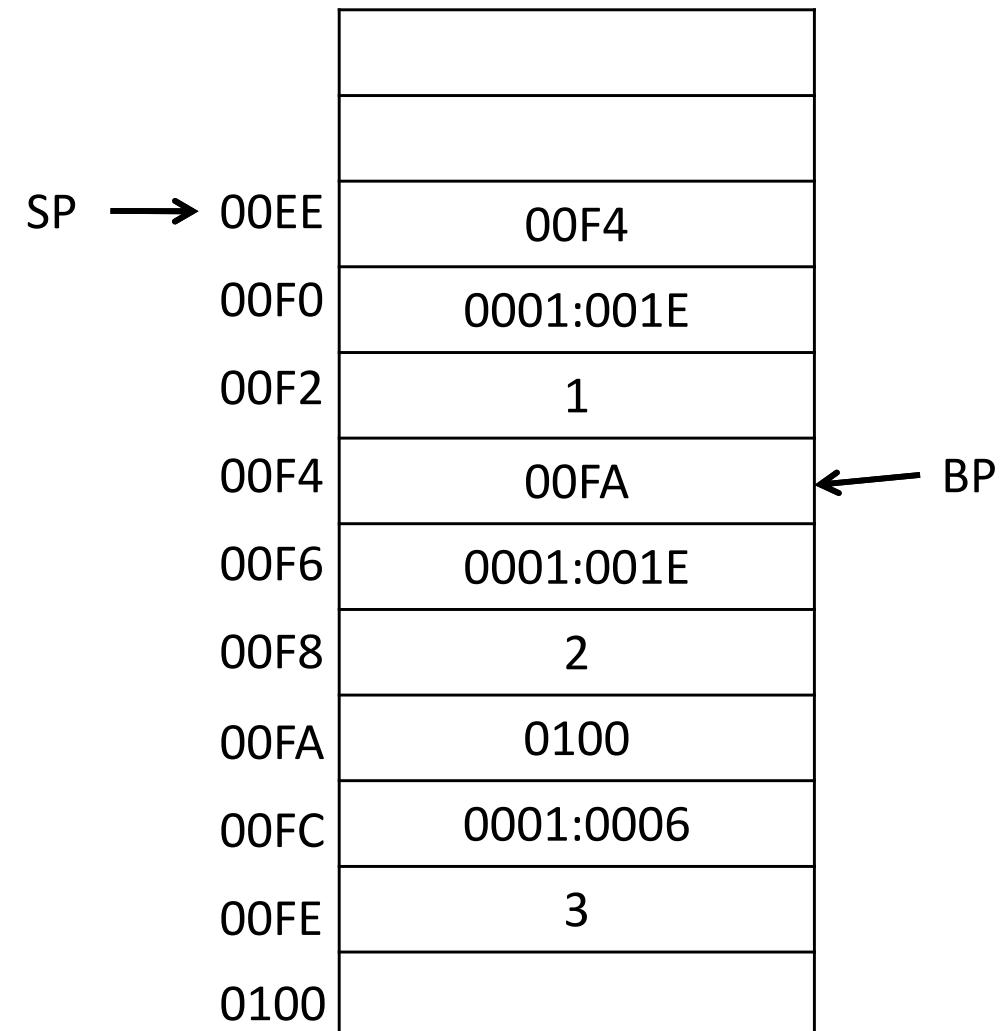
0000	MOV	AX,3
0002	PUSH	AX
0004	CALL	FACTORIAL
0006	MOV	AH,4CH
0008	INT	21H

AX	3
CX	1
SP	00EE
BP	00F4

MAIN ENDP

FACTORIAL PROC NEAR

000A	PUSH	BP
000C	MOV	BP,SP
000E	CMP	WORD PTR[BP+4],1
0010	JG	END_IF
0012	MOV	AX,1
0014	JMP	RETURN
0016	END_IF:	MOV CX, [BP+4]
0018	DEC	CX
001A	PUSH	CX
001C	CALL	FACTORIAL
001E	MUL	WORD PTR[BP+4]
0020	RETURN :	POP BP
0022	RET	2
	END	MAIN



MAIN PROC

0000	MOV	AX,3
0002	PUSH	AX
0004	CALL	FACTORIAL
0006	MOV	AH,4CH
0008	INT	21H

AX	3
CX	1
SP	00EE
BP	00EE

MAIN ENDP

FACTORIAL PROC NEAR

000A	PUSH	BP
000C	MOV	BP,SP
000E	CMP	WORD PTR[BP+4],1
0010	JG	END_IF
0012	MOV	AX,1
0014	JMP	RETURN
0016	END_IF:	MOV CX, [BP+4]
0018	DEC	CX
001A	PUSH	CX
001C	CALL	FACTORIAL
001E	MUL	WORD PTR[BP+4]
0020	RETURN :	POP BP
0022		RET 2
	END	MAIN

SP →

00EE

00F4	00F4
00F0	0001:001E
00F2	1
00F4	00FA
00F6	0001:001E
00F8	2
00FA	0100
00FC	0001:0006
00FE	3
0100	

BP

MAIN PROC

0000	MOV	AX,3
0002	PUSH	AX
0004	CALL	FACTORIAL
0006	MOV	AH,4CH
0008	INT	21H

AX	3
CX	1
SP	00EE
BP	00EE

MAIN ENDP

FACTORIAL PROC NEAR

000A	PUSH	BP
000C	MOV	BP,SP
000E	CMP	WORD PTR[BP+4],1
0010	JG	END_IF
0012	MOV	AX,1
0014	JMP	RETURN
0016	END_IF:	MOV CX, [BP+4]
0018	DEC	CX
001A	PUSH	CX
001C	CALL	FACTORIAL
001E	MUL	WORD PTR[BP+4]
0020	RETURN :	POP BP
0022		RET 2
	END	MAIN

SP → 00EE ← BP

00F4	00F4
00F0	0001:001E
00F2	1
00F4	00FA
00F6	0001:001E
00F8	2
00FA	0100
00FC	0001:0006
00FE	3
0100	

MAIN PROC

0000	MOV	AX,3
0002	PUSH	AX
0004	CALL	FACTORIAL
0006	MOV	AH,4CH
0008	INT	21H

AX	3
CX	1
SP	00EE
BP	00EE

MAIN ENDP

FACTORIAL PROC NEAR

000A	PUSH	BP
000C	MOV	BP,SP
000E	CMP	WORD PTR[BP+4],1
0010	JG	END_IF
0012	MOV	AX,1
0014	JMP	RETURN
0016	END_IF:	MOV CX, [BP+4]
0018	DEC	CX
001A	PUSH	CX
001C	CALL	FACTORIAL
001E	MUL	WORD PTR[BP+4]
0020	RETURN :	POP BP
0022		RET 2
	END	MAIN

SP →

00EE 00F4 ← BP

00F0	0001:001E
00F2	1
00F4	00FA
00F6	0001:001E
00F8	2
00FA	0100
00FC	0001:0006
00FE	3
0100	

MAIN PROC

0000	MOV	AX,3
0002	PUSH	AX
0004	CALL	FACTORIAL
0006	MOV	AH,4CH
0008	INT	21H

AX	3
CX	1
SP	00EE
BP	00EE

MAIN ENDP

FACTORIAL PROC NEAR

000A	PUSH	BP
000C	MOV	BP,SP
000E	CMP	WORD PTR[BP+4],1
0010	JG	END_IF
0012	MOV	AX,1
0014	JMP	RETURN
0016	END_IF:	MOV CX, [BP+4]
0018	DEC	CX
001A	PUSH	CX
001C	CALL	FACTORIAL
001E	MUL	WORD PTR[BP+4]
0020	RETURN :	POP BP
0022		RET 2
	END	MAIN

SP → 00EE

← BP

00F4	00F4
00F0	0001:001E
00F2	1
00F4	00FA
00F6	0001:001E
00F8	2
00FA	0100
00FC	0001:0006
00FE	3
0100	

MAIN PROC

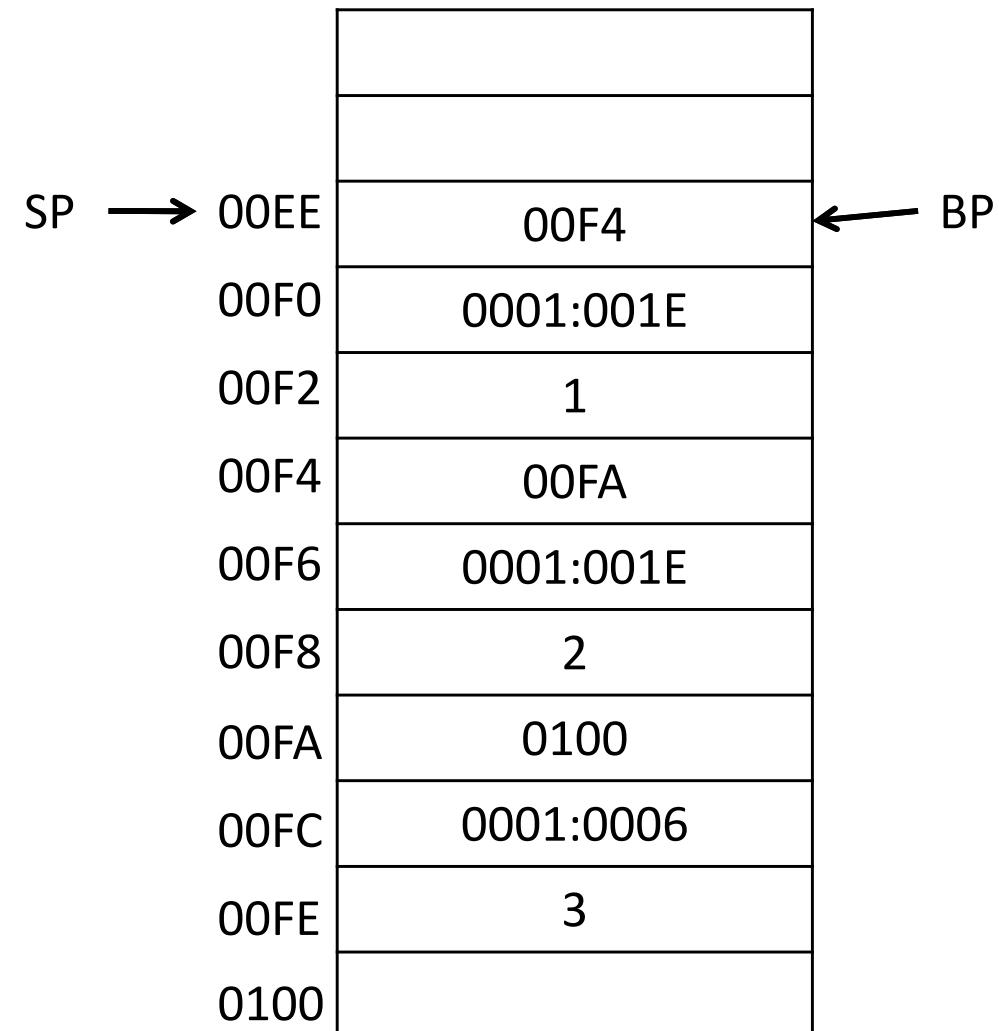
0000	MOV	AX,3
0002	PUSH	AX
0004	CALL	FACTORIAL
0006	MOV	AH,4CH
0008	INT	21H

AX	1
CX	1
SP	00EE
BP	00EE

MAIN ENDP

FACTORIAL PROC NEAR

000A	PUSH	BP
000C	MOV	BP,SP
000E	CMP	WORD PTR[BP+4],1
0010	JG	END_IF
0012	MOV	AX,1
0014	JMP	RETURN
0016	END_IF:	MOV CX, [BP+4]
0018	DEC	CX
001A	PUSH	CX
001C	CALL	FACTORIAL
001E	MUL	WORD PTR[BP+4]
0020	RETURN :	POP BP
0022	RET	2
	END	MAIN



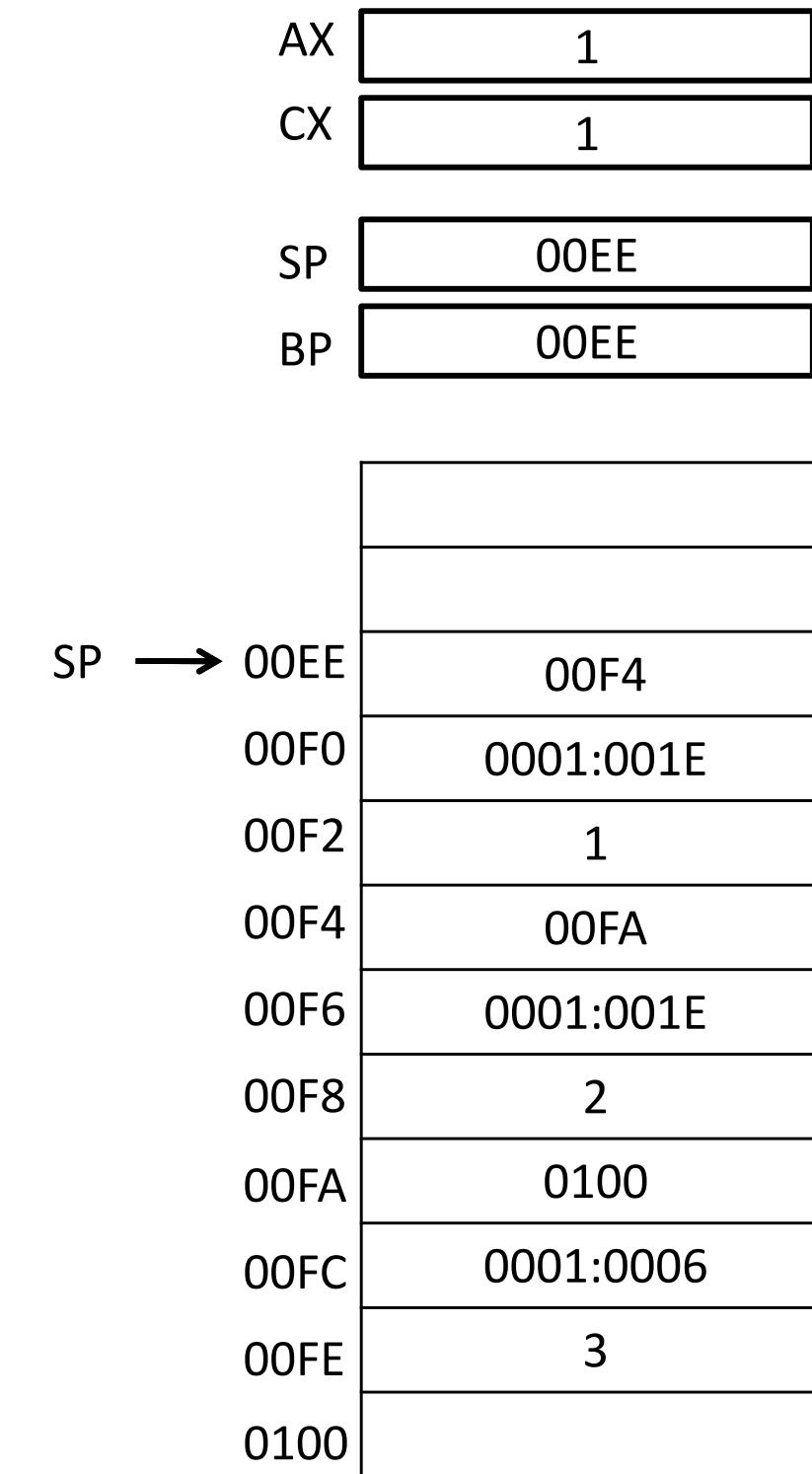
MAIN PROC

0000	MOV	AX,3	AX	1
0002	PUSH	AX	CX	1
0004	CALL	FACTORIAL	SP	00EE
0006	MOV	AH,4CH	BP	00EE
0008	INT	21H		

MAIN ENDP

FACTORIAL PROC NEAR

000A	PUSH	BP		
000C	MOV	BP,SP		
000E	CMP	WORD PTR[BP+4],1	SP → 00EE	00F4
0010	JG	END_IF		00F0 0001:001E
0012	MOV	AX,1		00F2 1
0014	JMP	RETURN		00F4 00FA
0016	END_IF:	MOV CX, [BP+4]		00F6 0001:001E
0018	DEC	CX		00F8 2
001A	PUSH	CX		00FA 0100
001C	CALL	FACTORIAL		00FC 0001:0006
001E	MUL	WORD PTR[BP+4]		00FE 3
0020	RETURN :	POP BP		0100
0022		RET 2		
	END	MAIN		



BP

MAIN PROC

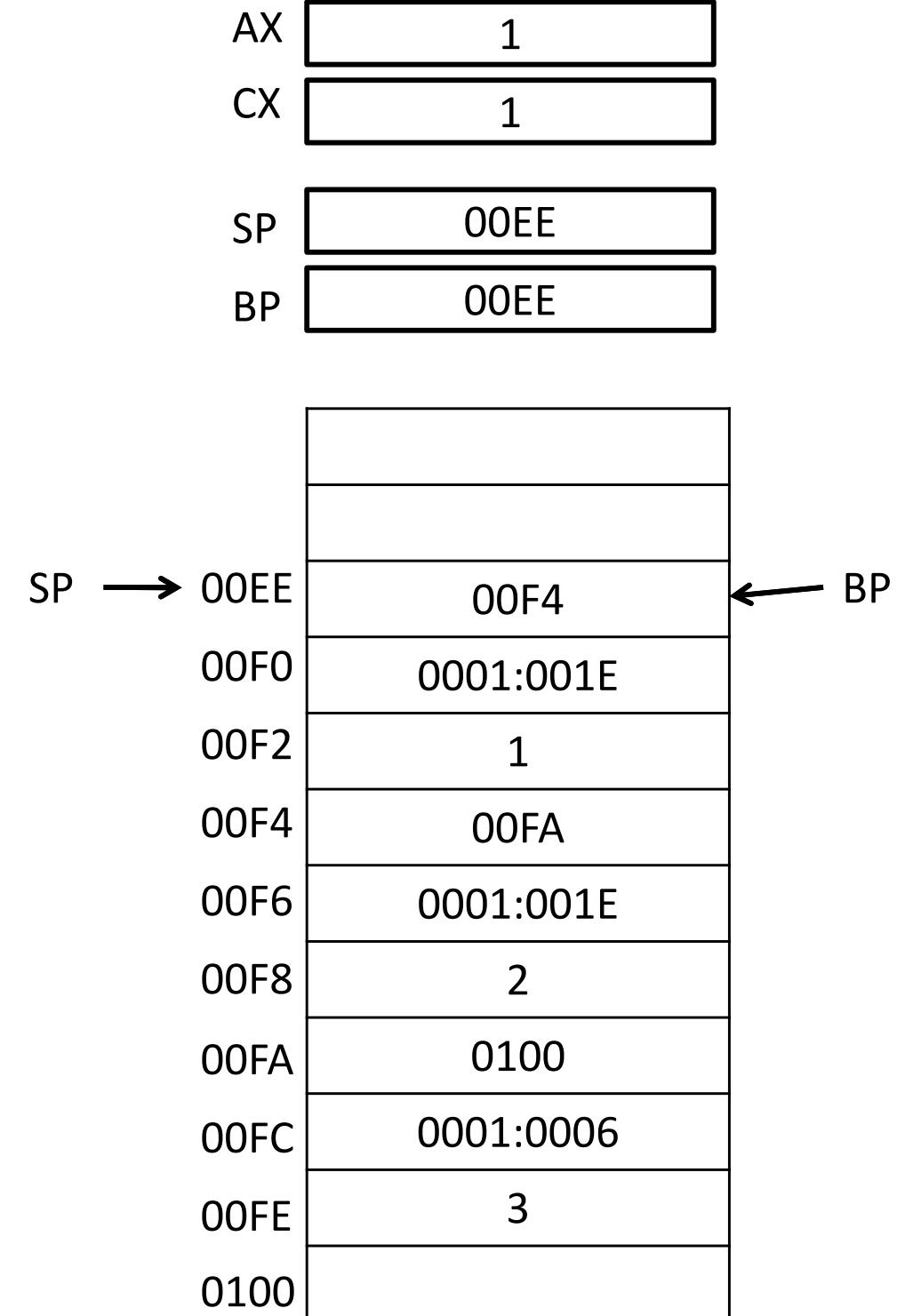
0000	MOV	AX,3
0002	PUSH	AX
0004	CALL	FACTORIAL
0006	MOV	AH,4CH
0008	INT	21H

AX	1
CX	1
SP	00EE
BP	00EE

MAIN ENDP

FACTORIAL PROC NEAR

000A	PUSH	BP
000C	MOV	BP,SP
000E	CMP	WORD PTR[BP+4],1
0010	JG	END_IF
0012	MOV	AX,1
0014	JMP	RETURN
0016	END_IF:	MOV CX, [BP+4]
0018	DEC	CX
001A	PUSH	CX
001C	CALL	FACTORIAL
001E	MUL	WORD PTR[BP+4]
0020	RETURN :	POP BP
0022	END	RET 2
		MAIN



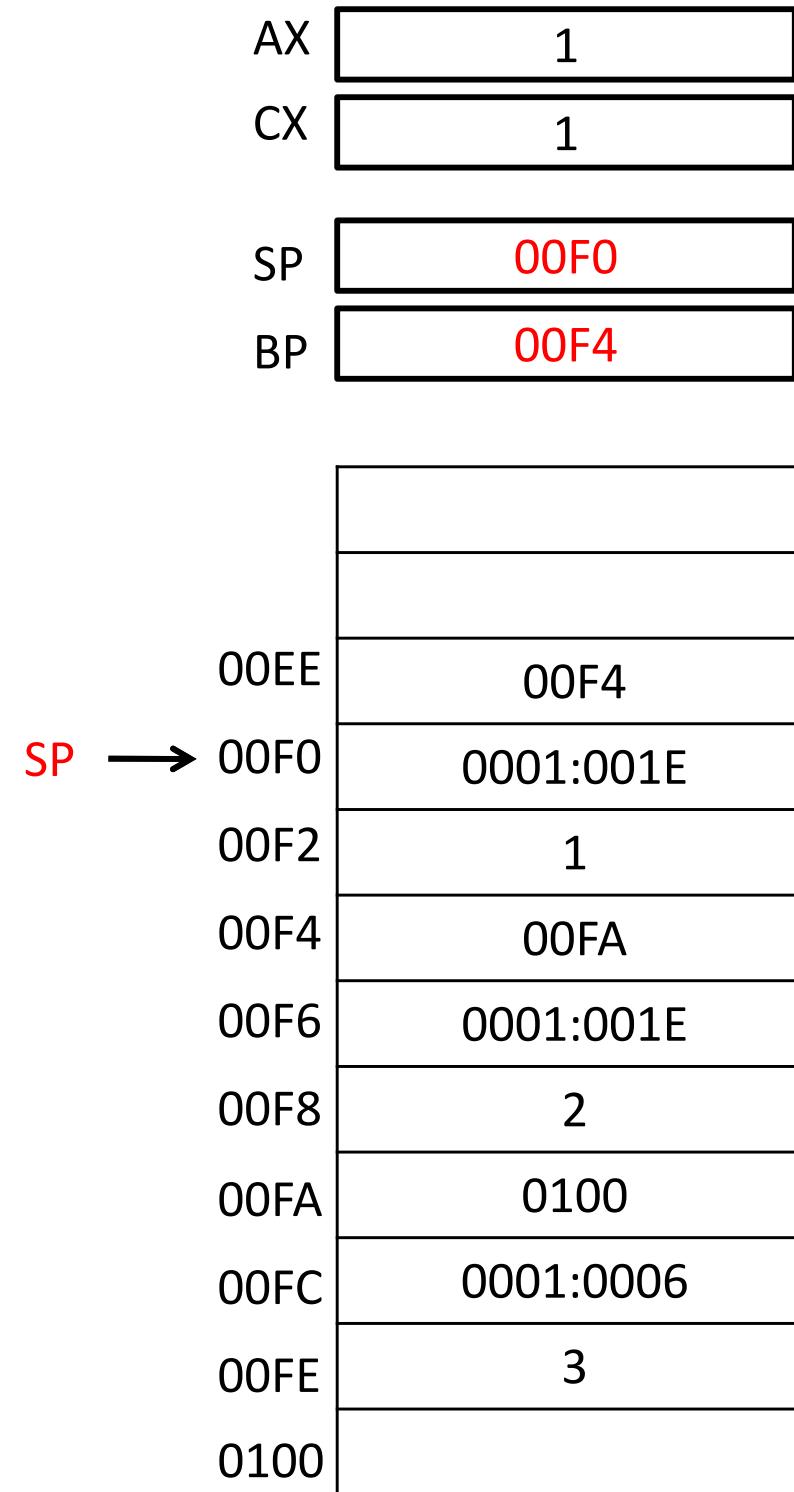
MAIN PROC

0000	MOV	AX,3	AX	1
0002	PUSH	AX	CX	1
0004	CALL	FACTORIAL	SP	00F0
0006	MOV	AH,4CH	BP	00F4
0008	INT	21H		

MAIN ENDP

FACTORIAL PROC NEAR

000A	PUSH	BP		
000C	MOV	BP,SP		
000E	CMP	WORD PTR[BP+4],1	00EE	
0010	JG	END_IF	00F0	0001:001E
0012	MOV	AX,1	00F2	1
0014	JMP	RETURN	00F4	00FA
0016	END_IF:	MOV CX, [BP+4]	00F6	0001:001E
0018	DEC	CX	00F8	2
001A	PUSH	CX	00FA	0100
001C	CALL	FACTORIAL	00FC	0001:0006
001E	MUL	WORD PTR[BP+4]	00FE	3
0020	RETURN :	POP BP	0100	
0022	END	RET 2		



MAIN PROC

0000	MOV	AX,3
0002	PUSH	AX
0004	CALL	FACTORIAL
0006	MOV	AH,4CH
0008	INT	21H

AX	1
CX	1
SP	00F0
BP	00F4

MAIN ENDP

FACTORIAL PROC NEAR

000A	PUSH	BP
000C	MOV	BP,SP
000E	CMP	WORD PTR[BP+4],1
0010	JG	END_IF
0012	MOV	AX,1
0014	JMP	RETURN
0016	END_IF:	MOV CX, [BP+4]
0018	DEC	CX
001A	PUSH	CX
001C	CALL	FACTORIAL
001E	MUL	WORD PTR[BP+4]
0020	RETURN :	POP BP
0022	RET	2
	END	MAIN

SP → 00F0 0001:001E

00EE	00F4
00F0	0001:001E
00F2	1
00F4	00FA
00F6	0001:001E
00F8	2
00FA	0100
00FC	0001:0006
00FE	3
0100	

MAIN PROC

0000	MOV	AX,3
0002	PUSH	AX
0004	CALL	FACTORIAL
0006	MOV	AH,4CH
0008	INT	21H

AX	1
CX	1
SP	00F4
BP	00F4

MAIN ENDP

FACTORIAL PROC NEAR

000A	PUSH	BP
000C	MOV	BP,SP
000E	CMP	WORD PTR[BP+4],1
0010	JG	END_IF
0012	MOV	AX,1
0014	JMP	RETURN
0016	END_IF:	MOV CX, [BP+4]
0018	DEC	CX
001A	PUSH	CX
001C	CALL	FACTORIAL
001E	MUL	WORD PTR[BP+4]
0020	RETURN :	POP BP
0022	END	RET 2
		MAIN

SP → 00F4 ← BP

00EE	00F4
00F0	0001:001E
00F2	1
00F4	00FA
00F6	0001:001E
00F8	2
00FA	0100
00FC	0001:0006
00FE	3
0100	

MAIN PROC

0000	MOV	AX,3	
0002	PUSH	AX	
0004	CALL	FACTORIAL	
0006	MOV	AH,4CH	
0008	INT	21H	

AX	1
CX	1
SP	00F4
BP	00F4

MAIN ENDP

FACTORIAL PROC NEAR

000A	PUSH	BP	
000C	MOV	BP,SP	
000E	CMP	WORD PTR[BP+4],1	
0010	JG	END_IF	
0012	MOV	AX,1	
0014	JMP	RETURN	
0016	END_IF:	MOV CX, [BP+4]	
0018	DEC	CX	
001A	PUSH	CX	
001C	CALL	FACTORIAL	
001E	MUL	WORD PTR[BP+4]	
0020	RETURN :	POP BP	
0022		RET 2	
	END	MAIN	

00EE	00F4	
00F0	0001:001E	
00F2	1	
00F4	00FA	BP
00F6	0001:001E	
00F8	2	
00FA	0100	
00FC	0001:0006	
00FE	3	
0100		

SP → 00F4

MAIN PROC

0000	MOV	AX,3
0002	PUSH	AX
0004	CALL	FACTORIAL
0006	MOV	AH,4CH
0008	INT	21H

AX	2
CX	1
SP	00F4
BP	00F4

MAIN ENDP

FACTORIAL PROC NEAR

000A	PUSH	BP
000C	MOV	BP,SP
000E	CMP	WORD PTR[BP+4],1
0010	JG	END_IF
0012	MOV	AX,1
0014	JMP	RETURN
0016	END_IF:	MOV CX, [BP+4]
0018	DEC	CX
001A	PUSH	CX
001C	CALL	FACTORIAL
001E	MUL	WORD PTR[BP+4]
0020	RETURN :	POP BP
0022		RET 2
	END	MAIN

00EE	00F4
00F0	0001:001E
00F2	1
00F4	00FA
00F6	0001:001E
00F8	2
00FA	0100
00FC	0001:0006
00FE	3
0100	

SP → 00F4

BP ←

MAIN PROC

0000	MOV	AX,3
0002	PUSH	AX
0004	CALL	FACTORIAL
0006	MOV	AH,4CH
0008	INT	21H

AX	2
CX	1
SP	00F4
BP	00F4

MAIN ENDP

FACTORIAL PROC NEAR

000A	PUSH	BP
000C	MOV	BP,SP
000E	CMP	WORD PTR[BP+4],1
0010	JG	END_IF
0012	MOV	AX,1
0014	JMP	RETURN
0016	END_IF:	MOV CX, [BP+4]
0018	DEC	CX
001A	PUSH	CX
001C	CALL	FACTORIAL
001E	MUL	WORD PTR[BP+4]
0020	RETURN :	POP BP
0022	END	RET 2
		MAIN

SP → 00F4

00EE	00F4
00F0	0001:001E
00F2	1
00F4	00FA
00F6	0001:001E
00F8	2
00FA	0100
00FC	0001:0006
00FE	3
0100	

MAIN PROC

0000	MOV	AX,3
0002	PUSH	AX
0004	CALL	FACTORIAL
0006	MOV	AH,4CH
0008	INT	21H

AX	2
CX	1
SP	00F6
BP	00FA

MAIN ENDP

FACTORIAL PROC NEAR

000A	PUSH	BP
000C	MOV	BP,SP
000E	CMP	WORD PTR[BP+4],1
0010	JG	END_IF
0012	MOV	AX,1
0014	JMP	RETURN
0016	END_IF:	MOV CX, [BP+4]
0018	DEC	CX
001A	PUSH	CX
001C	CALL	FACTORIAL
001E	MUL	WORD PTR[BP+4]
0020	RETURN :	POP BP
0022	END	RET 2
		MAIN

00EE	00F4
00F0	0001:001E
00F2	1
00F4	00FA
00F6	0001:001E
00F8	2
00FA	0100
00FC	0001:0006
00FE	3
0100	

SP →

BP ←

MAIN PROC

0000	MOV	AX,3
0002	PUSH	AX
0004	CALL	FACTORIAL
0006	MOV	AH,4CH
0008	INT	21H

AX	2
CX	1
SP	00F6
BP	00FA

MAIN ENDP

FACTORIAL PROC NEAR

000A	PUSH	BP
000C	MOV	BP,SP
000E	CMP	WORD PTR[BP+4],1
0010	JG	END_IF
0012	MOV	AX,1
0014	JMP	RETURN
0016	END_IF:	MOV CX, [BP+4]
0018	DEC	CX
001A	PUSH	CX
001C	CALL	FACTORIAL
001E	MUL	WORD PTR[BP+4]
0020	RETURN :	POP BP
0022	RET	2
	END	MAIN

00EE	00F4
00F0	0001:001E
00F2	1
00F4	00FA
00F6	0001:001E
00F8	2
00FA	0100
00FC	0001:0006
00FE	3
0100	

SP → 00F6

BP ←

MAIN PROC

0000	MOV	AX,3
0002	PUSH	AX
0004	CALL	FACTORIAL
0006	MOV	AH,4CH
0008	INT	21H

AX	2
CX	1
SP	00FA
BP	00FA

MAIN ENDP

FACTORIAL PROC NEAR

000A	PUSH	BP
000C	MOV	BP,SP
000E	CMP	WORD PTR[BP+4],1
0010	JG	END_IF
0012	MOV	AX,1
0014	JMP	RETURN
0016	END_IF:	MOV CX, [BP+4]
0018	DEC	CX
001A	PUSH	CX
001C	CALL	FACTORIAL
001E	MUL	WORD PTR[BP+4]
0020	RETURN :	POP BP
0022	END	RET 2
		MAIN

00EE	00F4
00F0	0001:001E
00F2	1
00F4	00FA
00F6	0001:001E
00F8	2
00FA	0100
00FC	0001:0006
00FE	3
0100	

SP →

BP ←

MAIN PROC

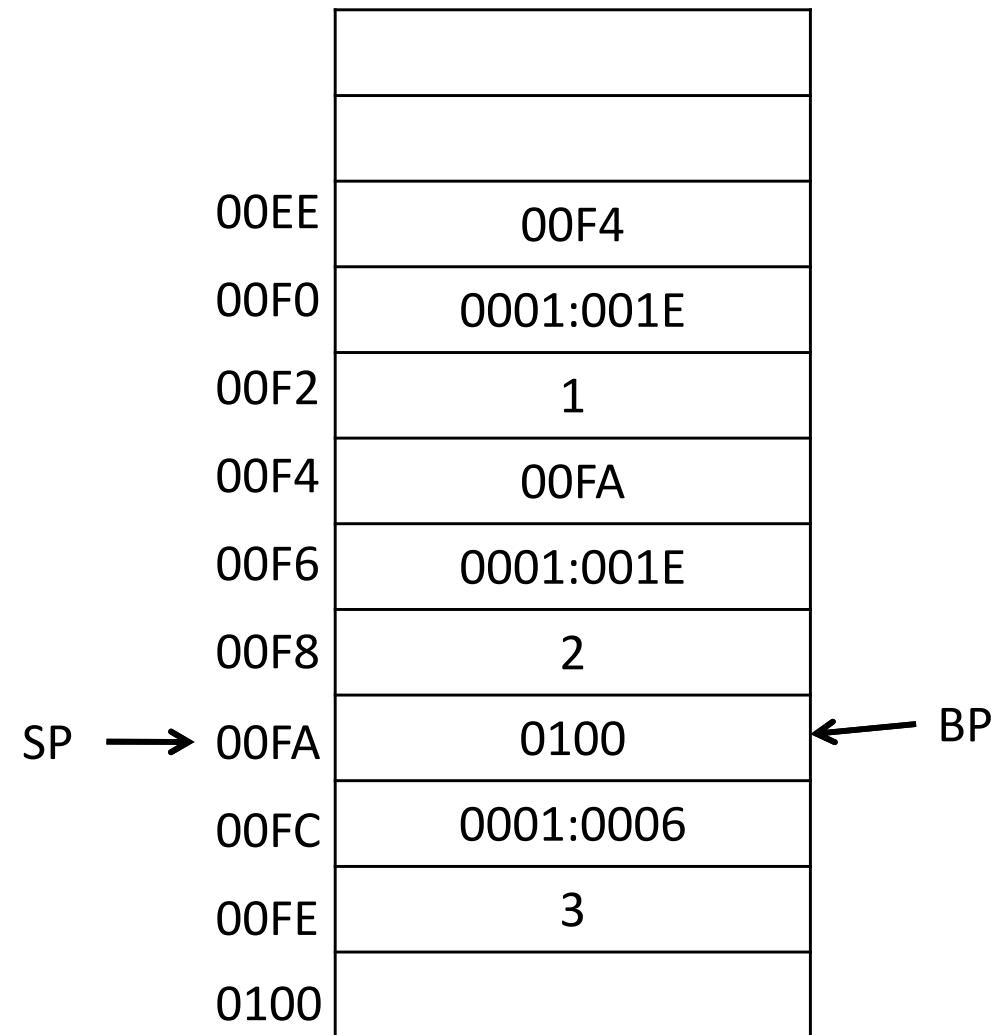
0000	MOV	AX,3
0002	PUSH	AX
0004	CALL	FACTORIAL
0006	MOV	AH,4CH
0008	INT	21H

AX	2
CX	1
SP	00FA
BP	00FA

MAIN ENDP

FACTORIAL PROC NEAR

000A	PUSH	BP
000C	MOV	BP,SP
000E	CMP	WORD PTR[BP+4],1
0010	JG	END_IF
0012	MOV	AX,1
0014	JMP	RETURN
0016	END_IF:	MOV CX, [BP+4]
0018	DEC	CX
001A	PUSH	CX
001C	CALL	FACTORIAL
001E	MUL	WORD PTR[BP+4]
0020	RETURN :	POP BP
0022	END	RET 2
		MAIN



MAIN PROC

0000	MOV	AX,3
0002	PUSH	AX
0004	CALL	FACTORIAL
0006	MOV	AH,4CH
0008	INT	21H

AX	6
CX	1
SP	00FA
BP	00FA

MAIN ENDP

FACTORIAL PROC NEAR

000A	PUSH	BP
000C	MOV	BP,SP
000E	CMP	WORD PTR[BP+4],1
0010	JG	END_IF
0012	MOV	AX,1
0014	JMP	RETURN
0016	END_IF:	MOV CX, [BP+4]
0018	DEC	CX
001A	PUSH	CX
001C	CALL	FACTORIAL
001E	MUL	WORD PTR[BP+4]
0020	RETURN :	POP BP
0022	END	RET 2
		MAIN

00EE	00F4
00F0	0001:001E
00F2	1
00F4	00FA
00F6	0001:001E
00F8	2
00FA	0100
00FC	0001:0006
00FE	3
0100	

SP →

BP ←

MAIN PROC

0000	MOV	AX,3
0002	PUSH	AX
0004	CALL	FACTORIAL
0006	MOV	AH,4CH
0008	INT	21H

AX	6
CX	1
SP	00FA
BP	00FA

MAIN ENDP

FACTORIAL PROC NEAR

000A	PUSH	BP
000C	MOV	BP,SP
000E	CMP	WORD PTR[BP+4],1
0010	JG	END_IF
0012	MOV	AX,1
0014	JMP	RETURN
0016	END_IF:	MOV CX, [BP+4]
0018	DEC	CX
001A	PUSH	CX
001C	CALL	FACTORIAL
001E	MUL	WORD PTR[BP+4]
0020	RETURN :	POP BP
0022	END	RET 2
		MAIN

00EE	00F4
00F0	0001:001E
00F2	1
00F4	00FA
00F6	0001:001E
00F8	2
00FA	0100
00FC	0001:0006
00FE	3
0100	

SP → 00FA

BP ←

MAIN PROC

0000	MOV	AX,3
0002	PUSH	AX
0004	CALL	FACTORIAL
0006	MOV	AH,4CH
0008	INT	21H

AX	6
CX	1
SP	00FC
BP	0100

MAIN ENDP

FACTORIAL PROC NEAR

000A	PUSH	BP
000C	MOV	BP,SP
000E	CMP	WORD PTR[BP+4],1
0010	JG	END_IF
0012	MOV	AX,1
0014	JMP	RETURN
0016	END_IF:	MOV CX, [BP+4]
0018	DEC	CX
001A	PUSH	CX
001C	CALL	FACTORIAL
001E	MUL	WORD PTR[BP+4]
0020	RETURN :	POP BP
0022		RET 2
	END	MAIN

00EE	00F4
00F0	0001:001E
00F2	1
00F4	00FA
00F6	0001:001E
00F8	2
00FA	0100
00FC	0001:0006
00FE	3
0100	

SP →

← BP

MAIN PROC

0000	MOV	AX,3
0002	PUSH	AX
0004	CALL	FACTORIAL
0006	MOV	AH,4CH
0008	INT	21H

AX	6
CX	1
SP	00FC
BP	0100

MAIN ENDP

FACTORIAL PROC NEAR

000A	PUSH	BP
000C	MOV	BP,SP
000E	CMP	WORD PTR[BP+4],1
0010	JG	END_IF
0012	MOV	AX,1
0014	JMP	RETURN
0016	END_IF:	MOV CX, [BP+4]
0018	DEC	CX
001A	PUSH	CX
001C	CALL	FACTORIAL
001E	MUL	WORD PTR[BP+4]
0020	RETURN :	POP BP
0022	RET	2
END	MAIN	

00EE	00F4
00F0	0001:001E
00F2	1
00F4	00FA
00F6	0001:001E
00F8	2
00FA	0100
00FC	0001:0006
00FE	3
0100	

SP → 00FC ← BP

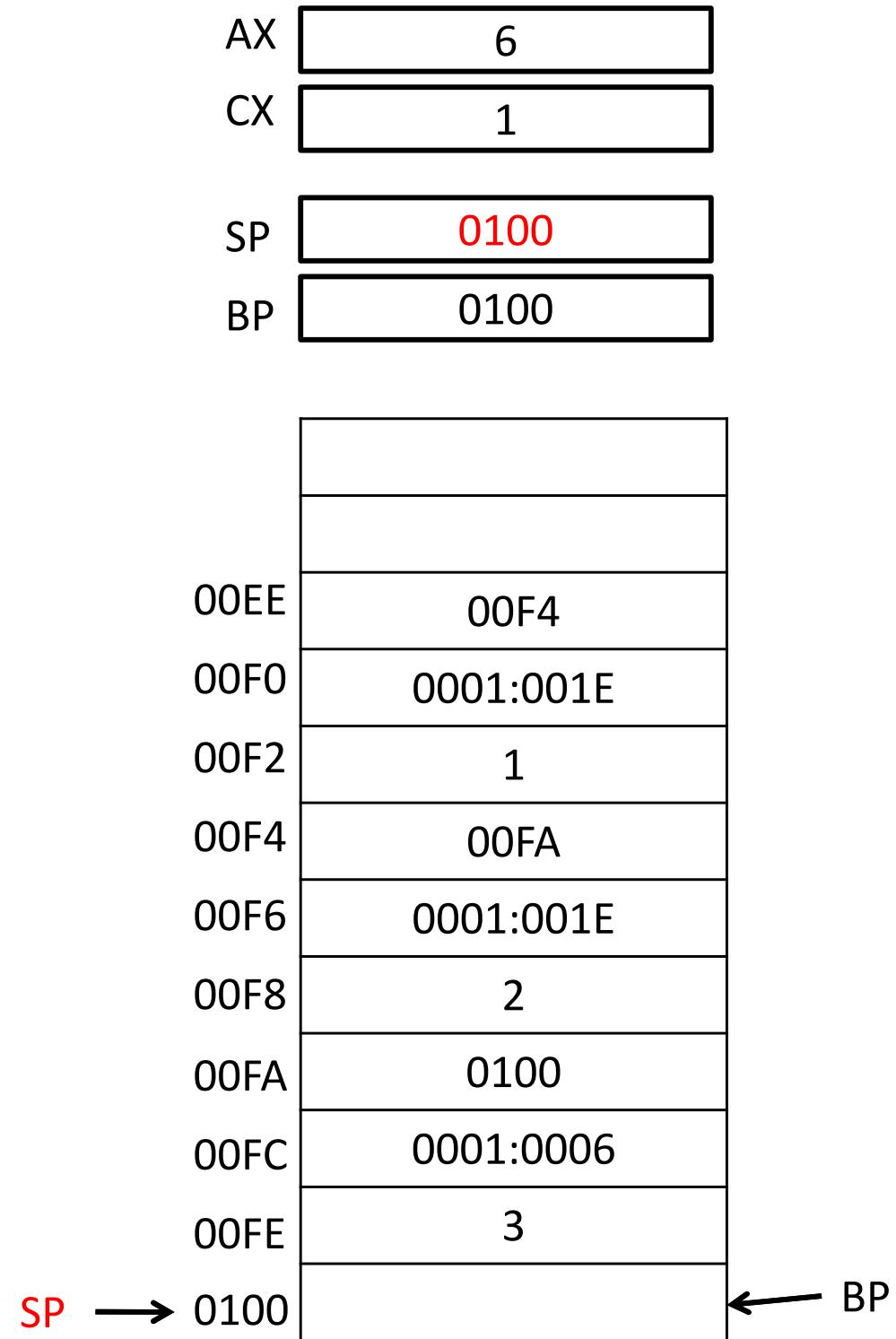
MAIN PROC

0000	MOV	AX,3	AX	6
0002	PUSH	AX	CX	1
0004	CALL	FACTORIAL	SP	0100
0006	MOV	AH,4CH	BP	0100
0008	INT	21H		

MAIN ENDP

FACTORIAL PROC NEAR

000A	PUSH	BP		
000C	MOV	BP,SP		
000E	CMP	WORD PTR[BP+4],1	00EE	00F4
0010	JG	END_IF	00F0	0001:001E
0012	MOV	AX,1	00F2	1
0014	JMP	RETURN	00F4	00FA
0016	END_IF:	MOV CX, [BP+4]	00F6	0001:001E
0018	DEC	CX	00F8	2
001A	PUSH	CX	00FA	0100
001C	CALL	FACTORIAL	00FC	0001:0006
001E	MUL	WORD PTR[BP+4]	00FE	3
0020	RETURN :	POP BP		
0022	RET	2		
	END	MAIN		



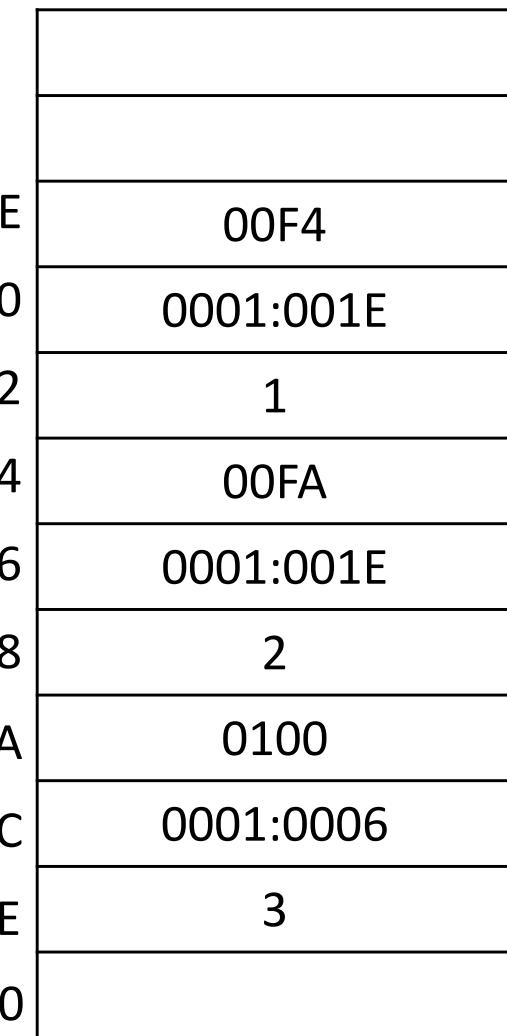
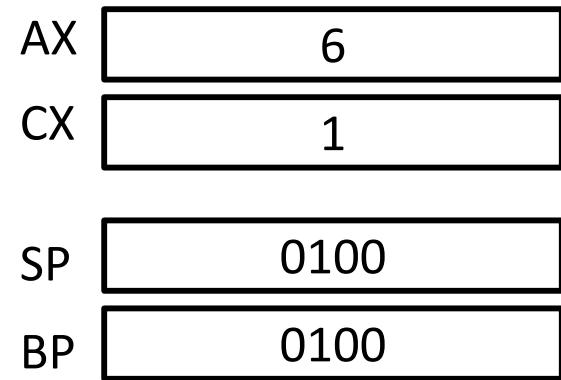
MAIN PROC

0000	MOV	AX,3	AX	6
0002	PUSH	AX	CX	1
0004	CALL	FACTORIAL	SP	0100
0006	MOV	AH,4CH	BP	0100
0008	INT	21H		

MAIN ENDP

FACTORIAL PROC NEAR

000A	PUSH	BP		
000C	MOV	BP,SP		
000E	CMP	WORD PTR[BP+4],1	00EE	00F4
0010	JG	END_IF	00F0	0001:001E
0012	MOV	AX,1	00F2	1
0014	JMP	RETURN	00F4	00FA
0016	END_IF:	MOV CX, [BP+4]	00F6	0001:001E
0018	DEC	CX	00F8	2
001A	PUSH	CX	00FA	0100
001C	CALL	FACTORIAL	00FC	0001:0006
001E	MUL	WORD PTR[BP+4]	00FE	3
0020	RETURN :	POP BP	SP → 0100	
0022		RET 2		BP ←
	END	MAIN		



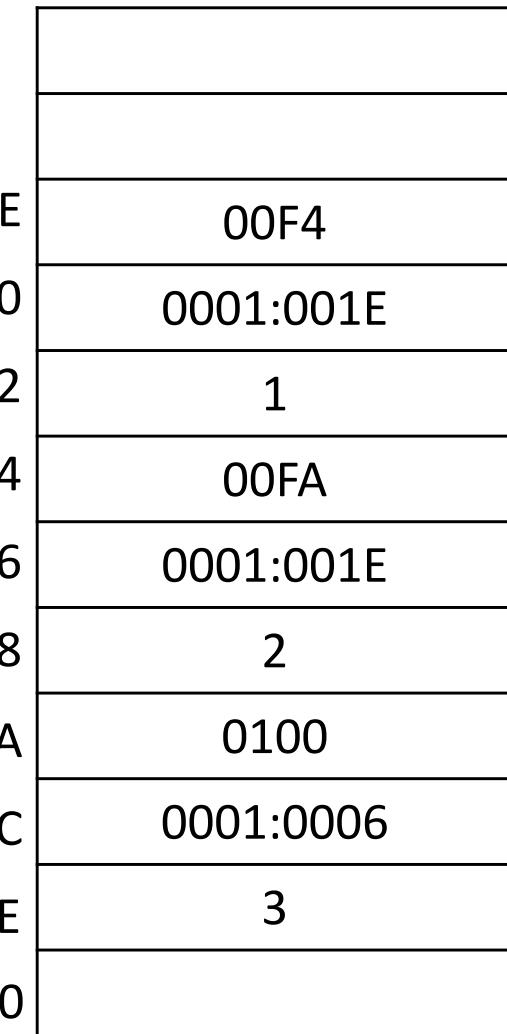
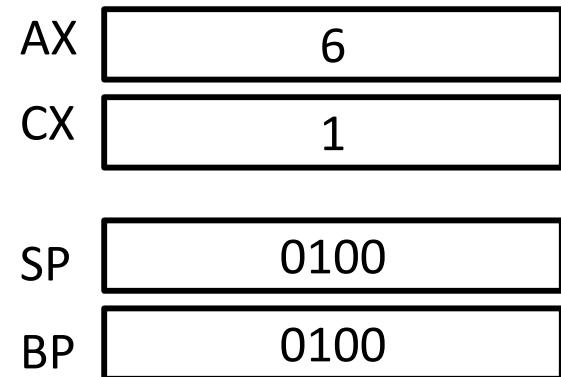
MAIN PROC

0000	MOV	AX,3	AX	6
0002	PUSH	AX	CX	1
0004	CALL	FACTORIAL	SP	0100
0006	MOV	AH,4CH	BP	0100
0008	INT	21H		

MAIN ENDP

FACTORIAL PROC NEAR

000A	PUSH	BP		
000C	MOV	BP,SP		
000E	CMP	WORD PTR[BP+4],1	00EE	00F4
0010	JG	END_IF	00F0	0001:001E
0012	MOV	AX,1	00F2	1
0014	JMP	RETURN	00F4	00FA
0016	END_IF:	MOV CX, [BP+4]	00F6	0001:001E
0018	DEC	CX	00F8	2
001A	PUSH	CX	00FA	0100
001C	CALL	FACTORIAL	00FC	0001:0006
001E	MUL	WORD PTR[BP+4]	00FE	3
0020	RETURN :	POP BP	SP → 0100	
0022		RET 2		
	END	MAIN		BP ←



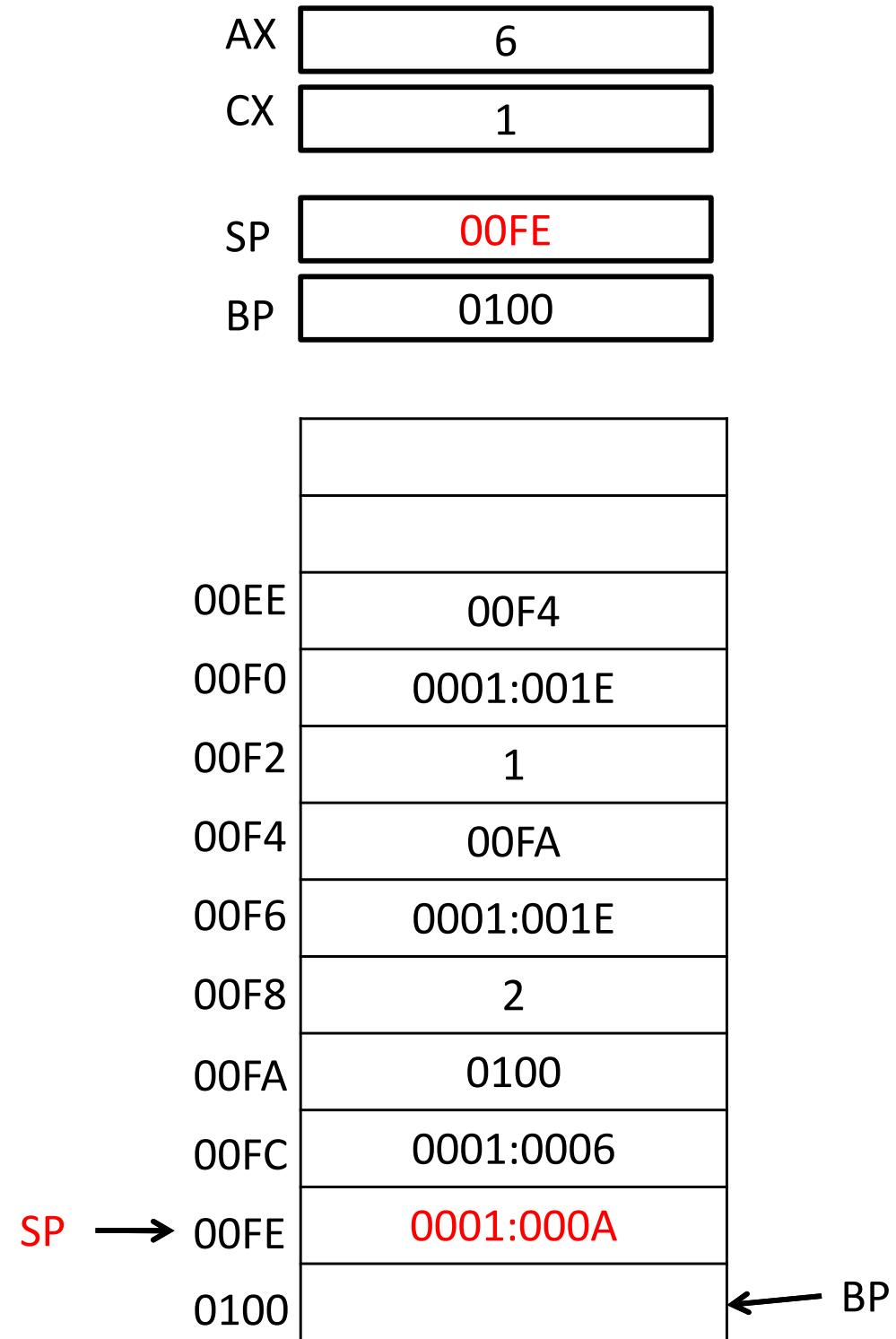
MAIN PROC

0000	MOV	AX,3	AX	6
0002	PUSH	AX	CX	1
0004	CALL	FACTORIAL	SP	00FE
0006	MOV	AH,4CH	BP	0100
0008	INT	21H		

MAIN ENDP

FACTORIAL PROC NEAR

000A	PUSH	BP		
000C	MOV	BP,SP		
000E	CMP	WORD PTR[BP+4],1	00EE	00F4
0010	JG	END_IF	00F0	0001:001E
0012	MOV	AX,1	00F2	1
0014	JMP	RETURN	00F4	00FA
0016	END_IF:	MOV CX, [BP+4]	00F6	0001:001E
0018	DEC	CX	00F8	2
001A	PUSH	CX	00FA	0100
001C	CALL	FACTORIAL	00FC	0001:0006
001E	MUL	WORD PTR[BP+4]	00FE	0001:000A
0020	RETURN :	POP BP		
0022		RET 2		
	END	MAIN		



References

- Ch 8, 17 Assembly Language Programming –
by Yu and Marut

Courtesy

- All the slides on 8086 assembly language have been prepared by:

Dr. Atif Hasan Rahman

Associate Professor,

Department of CSE, BUET

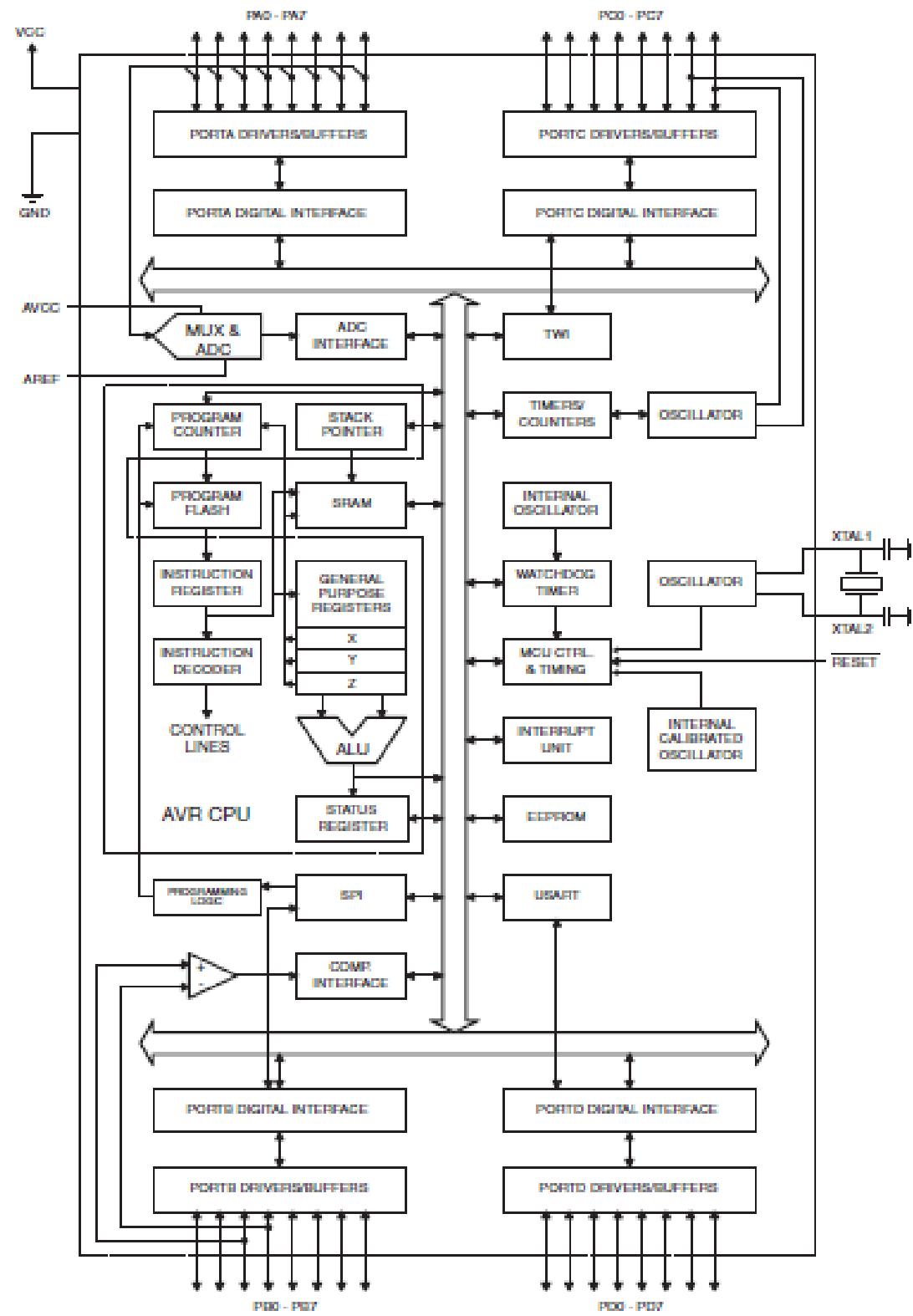
CSE 315

Microprocessors, Microcontrollers, and
Embedded Systems

Microcontrollers:
BASIC I/O in ATmega32/16

ATmega32

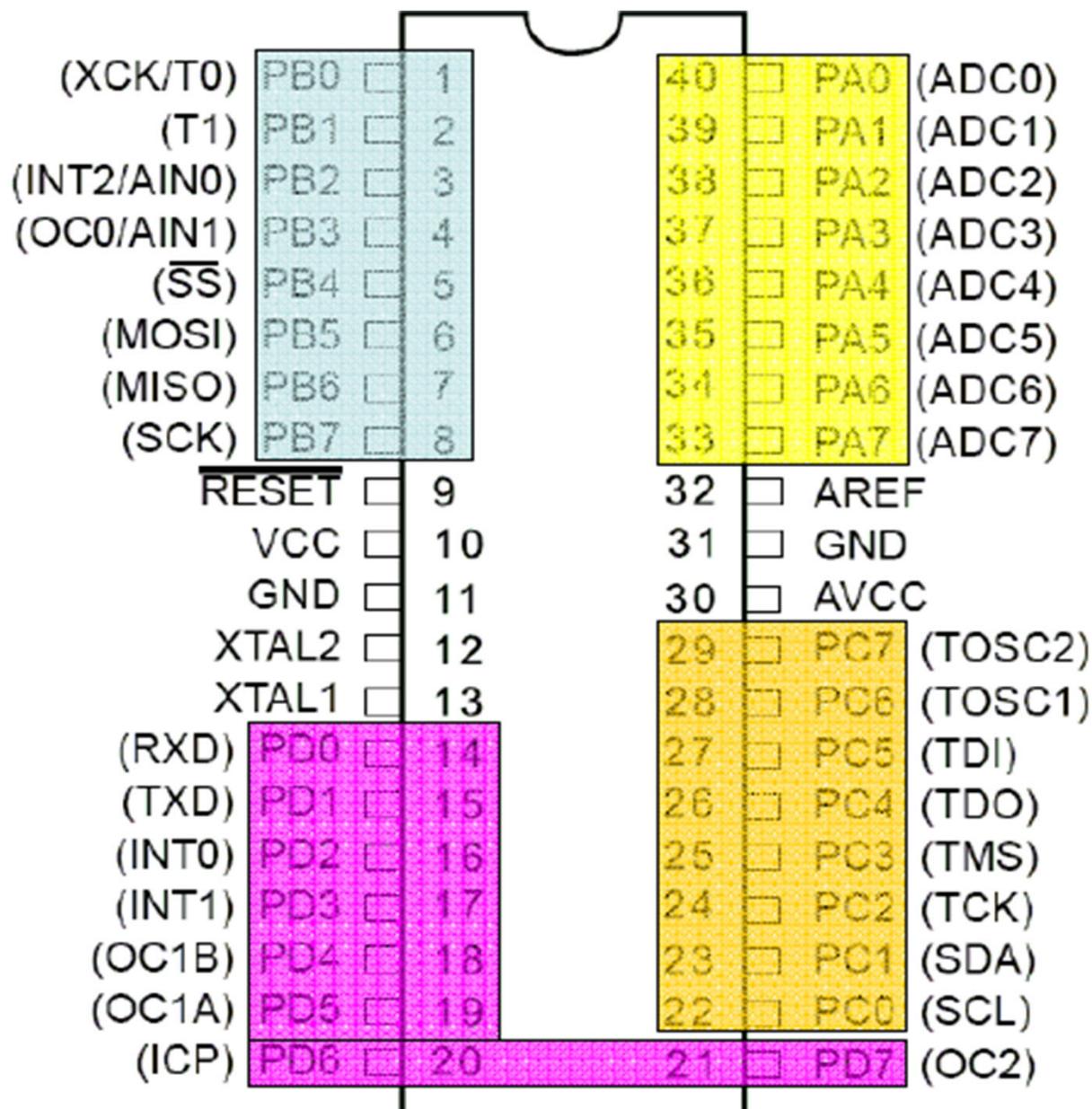
- RISC
- 131 instructions
- 8 bit processor
- 16MHz
- Flash program memory
- SRAM for data
- Timer, counter, ADC, USART



Ports for I/O

- 4 different ports for I/O
 - A, B, C, D
 - 8 bit \leftrightarrow 8 data pins
 - Every pin is bidirectional, can be used as input or output

I/O Ports of ATmega32



Port Operation Registers

- DDRx – Data Direction Register
- PORTx – Pin Output Register
- PINx – Pin Input Register
- x = A, B, C or D i.e. DDRA, PORTA, PINA

Configuration

- For configuration we have to use the Data Direction Registers
 - One register for each port
 - DDRx (DDRA, DDRB, DDRC, DDRD)
 - Configures each pin as input or output
- Pin configuration
 - Input – 0
 - Output – 1

C Code Example

- DDRA = 0b11111111;
 - Sets each bit of port A as output
- DDRB = 0b00000000;
 - Sets each bit of port B as input
- DDRC = 0b01010101;
 - ???

Input

- You have to read the PINx register

```
unsigned char ch;  
ch = PINA;
```

- What to do if only some of the pins are configured as input ?

Output

- You have to use the PORTx register

```
PORTB = 0b11111111;
```

- What to do if only some of the pins are configured as output ?

Write a simple program to set
the port D to 0xFF

C Code

```
#include <avr/io.h>

int main(void)
{
    DDRD= 0b11111111; //initializing portD in
                      //output mode
    PORTD=0b11111111;//writing value to portD

    while(1)
    {
        //TODO:: Nothing
    }
}
```

Write a simple program to blink a
LED on port B pin 0

C Code

```
#include <avr/io.h>

int main(void)
{
unsigned char c = 1;
DDRB= 0b00000001;

while(1)
{
    PORTB = c;
    if(c)c=0;
    else c=1;
    delay();
}
}

void delay()
{
unsigned char i,j,k;
for(i=0;i<255;i++)
for(j=0;j<255;j++)
for(k=0;k<100;k++);
}
```

Write a simple program to animate 8 LEDs connected to PORT B

- One LED at a time is ON
- The rest are off
- At first LED 0 is on, then LED 1, and so on.

C Code

```
int main(void)
{
    unsigned char c = 0x01;
    DDRB= 0xFF;

    while(1)
    {
        PORTB = c ;
        if(c==1<<7)c = 1;
        else c = c << 1;
        delay();
    }
}
```

Accurately Generating Delay

```
#include <avr/io.h>
#define F_CPU 1000000 // Clock Frequency
#include <util/delay.h>

int main(void)
{
    while(1)
    {
        //1000 ms delay
        _delay_ms(1000);
    }
}
```

Simple Input

- A 8 bit input is connected to PORT A
- Show the input state on PORT B

C Code

```
int main(void)
{
    unsigned char c;
    DDRA = 0x00;
    DDRB = 0xFF;

    while(1)
    {
        c = PINA;
        PORTB = c ;
    }
}
```

Simple Counter

- PA0 is connected with push button
 - 1 when pressed
- PORT B connected to 8 LEDs
- Increment count when pressed the push button

C Code

```
int main(void)
{
unsigned char c=0,in=0;
DDRA = 0xFE;
DDRB = 0xFF;

while(1)
{
    PORTB = c;

    in = PINA;
    if(in)
    {
        c++;
        _delay_ms(1000);
    }
}

}
```

C Code

```
int main(void)
{
unsigned char c=0,in=0;
DDRA = 0xFE;
DDRB = 0xFF;

while(1)
{
    PORTB = c;

    in = PINA;
    if(in)
    {
        c++;
        _delay_ms(1000);
    }
}
}
```

There is still a problem in the design.

C Code

```
int main(void)
{
    unsigned char c=0,in=0;
    DDRA = 0xFE;
    DDRB = 0xFF;

    PORTB = c;
    while(1)
    {
        in = PINA;
        if(in & 0x01)
        {
            c++;
            PORTB = c;
            _delay_ms(1000);
        }
    }
}
```

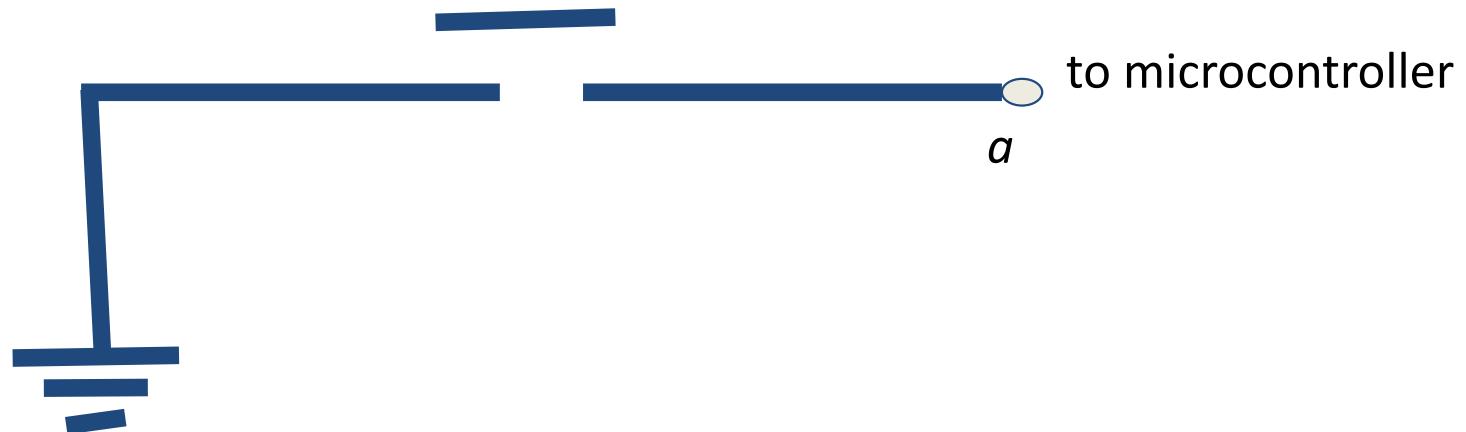
Write a program to read a byte from
PORT A and write it to PORT B

If input (taken from *PORTC*) is less than 100, send it to *PORTB*, otherwise, send it to *PORTD*

Write a program to read a byte from PORT A and write its upper nibble to PORT B (lower nibble) and lower nibble to PORT C (upper nibble)

Discussions on push buttons

- This might seem like a reasonable connection
 - what is wrong with it ?

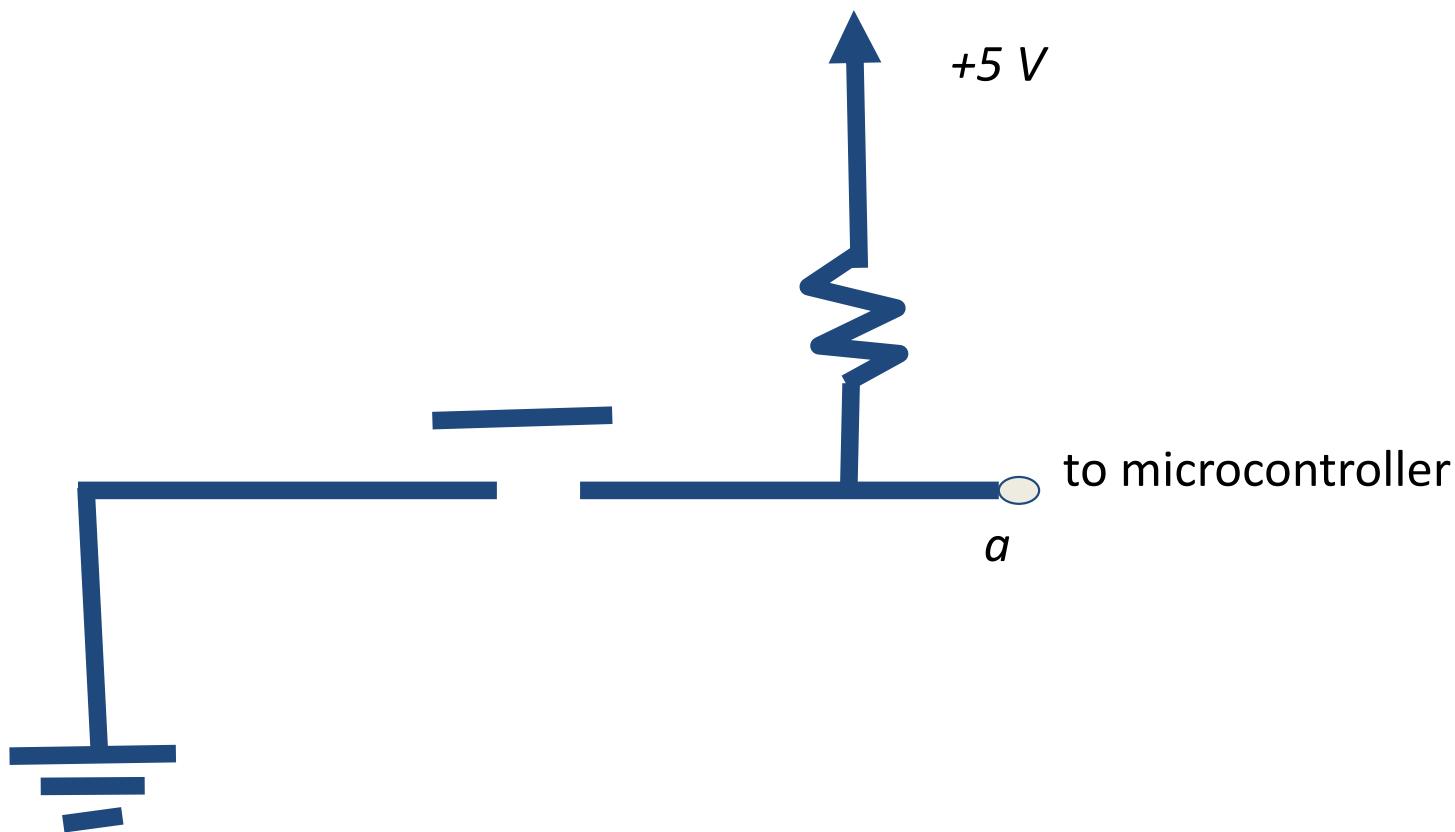


Discussions on push buttons

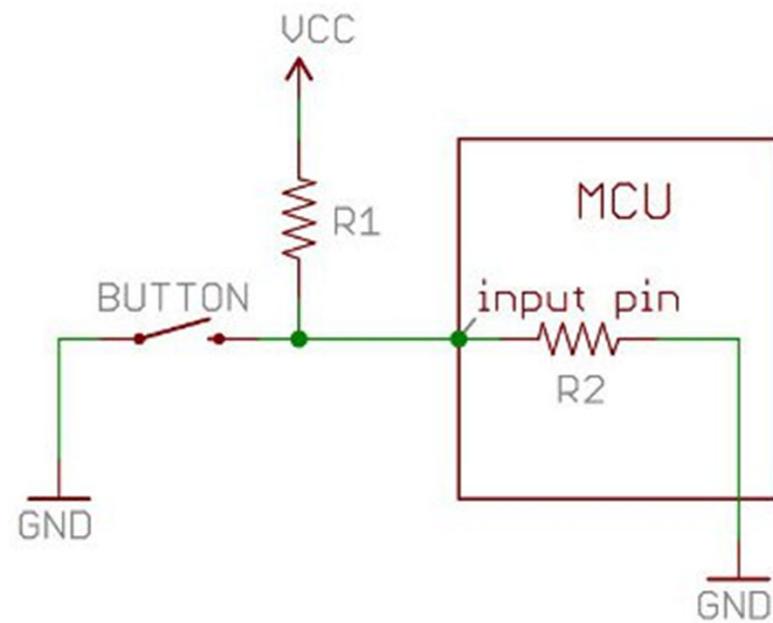
- This might seem like a reasonable connection
 - what is wrong with it ?
 - What is the voltage at terminal *a* when the button is not pressed?



Pull Up Resistors

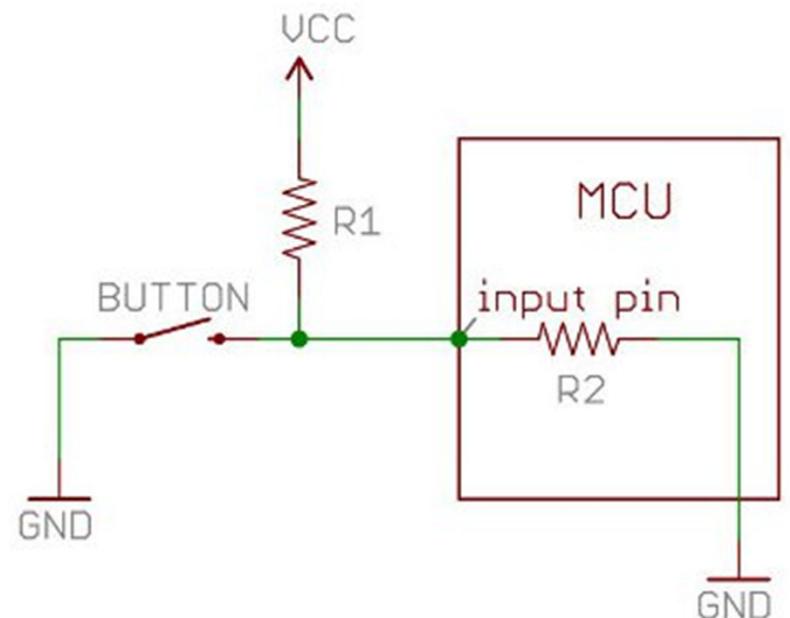


A deeper look



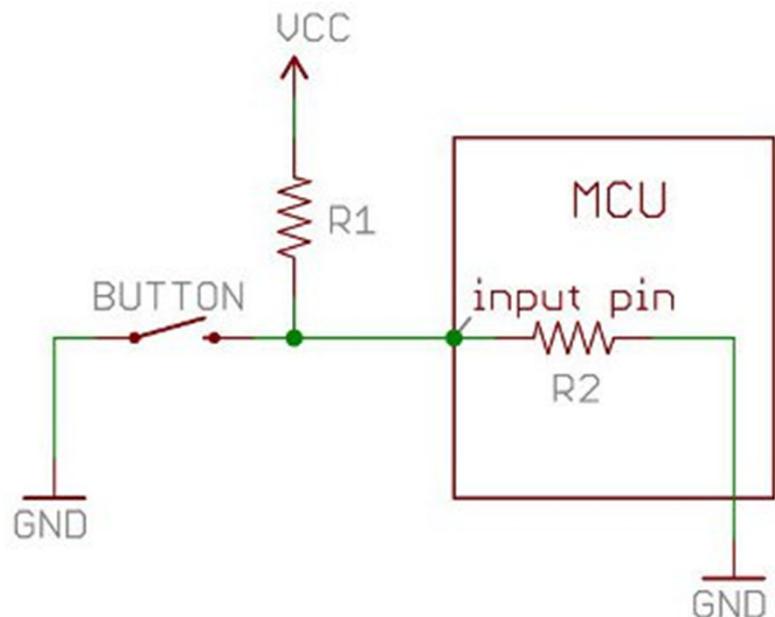
A deeper look

- When the button is pressed, the input pin is pulled low. The value of resistor R1 controls how much current we want to flow from VCC, through the button, and then to ground.



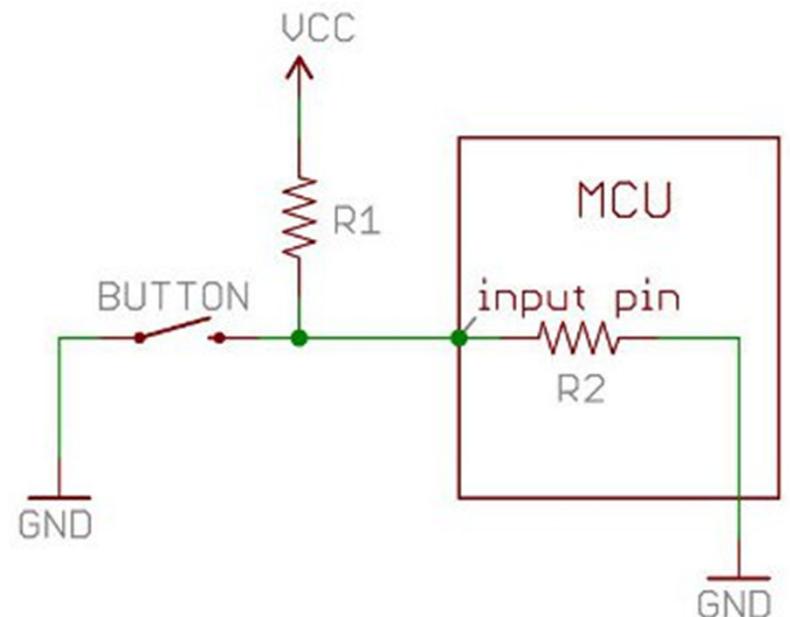
A deeper look

- When the button is not pressed, the input pin is pulled high. The value of the pull-up resistor controls the voltage on the input pin.

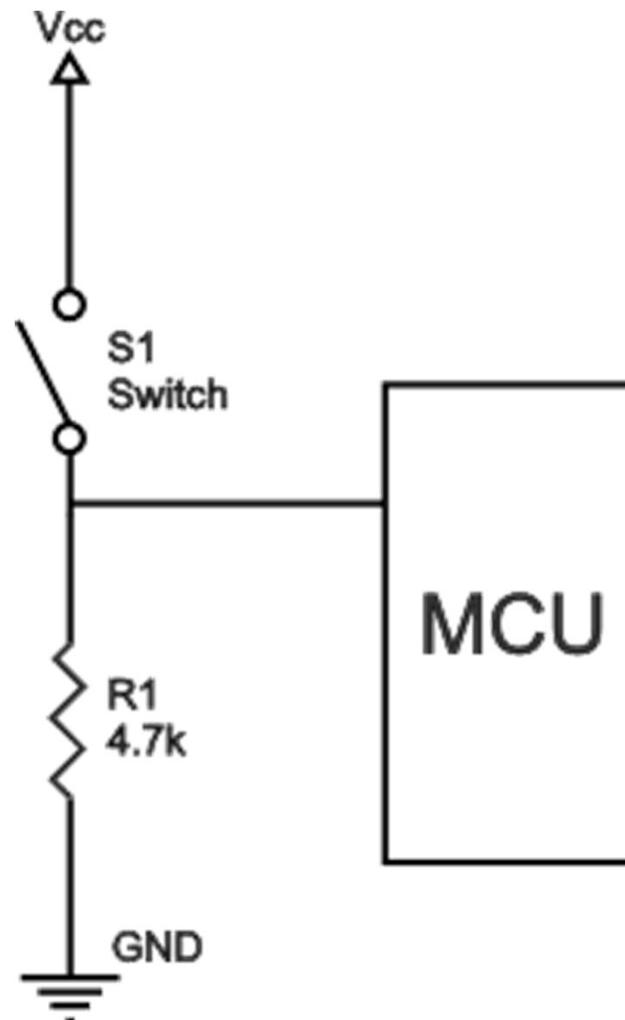


A deeper look

- Because of the two opposing factors the resistor value cannot be too high not too low
- Depending on the context you choose an appropriate value



Pull Down Resistor



ATmega 32 has internal pull up resistors!

If PORT xn is written logic one when the pin is configured as an input pin, the pull-up resistor is activated.

Here, $x = A, B, C$, or D

n is bit number, e.g., 0,1,2,..., or 7

General Discussion

- If some pins are unused, it is recommended to ensure that these pins have a defined level.
- The simplest method to ensure a defined level of an unused pin, is to enable the internal pullup.

Resources

- DataSheet
- Pull Up and Pull Down Resistors
 - http://www.resistorguide.com/pull-up-resistor_pull-down-resistor/
- More on Pull Up and Pull Down resistors
 - <https://learn.sparkfun.com/tutorials/pull-up-resistors>

CSE 315

Microprocessors, Microcontrollers, and
Embedded Systems

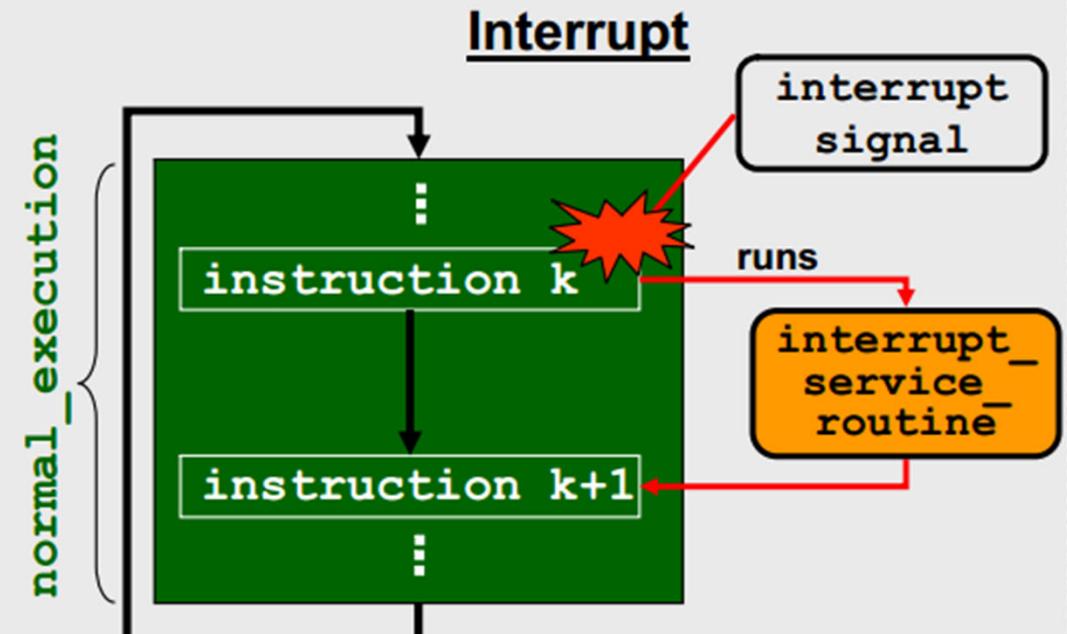
**Microcontrollers:
Interrupts in ATmega32/16**

Interrupts vs Polling

Polling

```
while (1){  
    get_device_status;  
    if (service_required) {  
        service_routine;  
    }  
    normal_execution;  
}
```

Interrupt



Interrupts vs Polling

- Using polling
 - the CPU must continually check the device's status.
- Using interrupt
 - A device will send an interrupt signal when needed.
 - In response, the CPU will perform an interrupt service routine, and then resume its normal execution.

Interrupts vs Polling

- Efficiency
- Monitoring several devices
- Priority

Interrupt execution sequence

A device issues an interrupt

CPU finishes the current instruction

CPU acknowledges the interrupt

CPU saves its states and PC onto stack

CPU loads the address of ISR onto PC

CPU executes the ISR

CPU retrieves its states and PC from stack

Normal execution resumes

ATmega32 interrupt subsystem

- The ATmega32 has total 21 interrupts
- 16 are of interest
 - 3 external interrupts
 - 8 timer/counter interrupts
 - 3 serial port interrupts
 - 1 ADC interrupt
 - 1 SPI interrupt

ATmega32 interrupt subsystem

- The ATmega32 has total 21 interrupts
- There are 5 others
 - 1 reset interrupt
 - 1 analogue comparator interrupt
 - 1 TWI interrupt
 - 2 memory interrupts

Complete List

Vector No.	Program Address	Interrupt vector name	Description
1	\$000	RESET_vect	Reset
2	\$002	INT0_vect	External Interrupt Request 0
3	\$004	INT1_vect	External Interrupt Request 1
4	\$006	TIMER2_COMP_vect	Timer/Counter2 Compare Match
5	\$008	TIMER2_OVF_vect	Timer/Counter2 Overflow
6	\$00A	TIMER1_CAPT_vect	Timer/Counter1 Capture Event
7	\$00C	TIMER1_COMPA_vect	Timer/Counter1 Compare Match A
8	\$00E	TIMER1_COMPB_vect	Timer/Counter1 Compare Match B
9	\$010	TIMER1_OVF_vect	Timer/Counter1 Overflow
10	\$012	TIMER0_OVF_vect	Timer/Counter0 Overflow
11	\$014	SPI_STC_vect	Serial Transfer Complete
12	\$016	USART_RXC_vect	USART, Rx Complete
13	\$018	USART_UDRE_vect	USART Data Register Empty
14	\$01A	USART_TXC_vect	USART, Tx Complete
15	\$01C	ADC_vect	ADC Conversion Complete
16	\$01E	EE_RDY_vect	EEPROM Ready
17	\$020	ANA_COMP_vect	Analog Comparator
18	\$022	TWI_vect	2-wire Serial Interface
19	\$024	INT2_vect	External Interrupt Request 2
20	\$026	TIMER0_COMP_vect	Timer/Counter0 Compare Match
21	\$028	SPM_RDY_vect	Store Program Memory Ready

Complete List

Vector No.	Program Address	Interrupt vector name	Description
1	\$000	RESET_vect	Reset
2	\$002	INT0_vect	External Interrupt Request 0
3	\$004	INT1_vect	External Interrupt Request 1
4	\$006	TIMER2_COMP_vect	Timer/Counter2 Compare Match
5	\$008		
6	\$00A		
7	\$00C		
8	\$00E		
9	\$010		
10	\$012		
11	\$014		
12	\$016		
13	\$018		
14	\$01A		
15	\$01C		
16	\$01E		
17	\$020		
18	\$022	TWI_vect	2-wire Serial Interface
19	\$024	INT2_vect	External Interrupt Request 2
20	\$026	TIMER0_COMP_vect	Timer/Counter0 Compare Match
21	\$028	SPM_RDY_vect	Store Program Memory Ready

Vector No.

- An interrupt with a lower ‘Vector No’ will have a higher priority.
- INT0 has a higher priority than INT1 and INT2

Complete List

Vector No.	Program Address	Interrupt vector name	Description
1	\$000	RESET_vect	Reset
2	\$002	INT0_vect	External Interrupt Request 0
3	\$004	INT1_vect	External Interrupt Request 1
4	\$006	TIMER1_COMP_vect	Timer/Counter2 Compare Match
5	\$008	TIMER2_OVF_vect	Timer/Counter2 Overflow
6	\$00A	TIMER1_CAPT_vect	Timer/Counter1 Capture Event
7	\$00C		
8	\$00E		
9	\$010		
10	\$012		
11	\$014		
12	\$016		
13	\$018		
14	\$01A		
15	\$01C		
16	\$01E		
17	\$020		
18	\$022		
19	\$024		
20	\$026		
21	\$028		

Program Address.

- The fixed memory location for a given interrupt handler.
 - E.g., in response to interrupt INT0, CPU runs instruction at \$002.
 - Usually the instruction is JMP address (address of ISR)

Complete List

Vector No.	Program Address	Interrupt vector name	Description
1	\$000	RESET_vect	Reset
2	\$002	INT0_vect	External Interrupt Request 0
3	\$004	INT1_vect	External Interrupt Request 1
4	\$006	TIMER2_COMP_vect	Timer/Counter2 Compare Match
5	\$008	TIMER2_OVF_vect	Timer/Counter2 Overflow
6	\$00A	TIMER1_CAPT_vect	Timer/Counter1 Capture
7	\$00C	TIMER1_COMPA_vect	Timer/Counter1 Compare A
8	\$00E	TIMER1_COMPB_vect	Timer/Counter1 Compare B
9	\$010	TIMER1_OVF_vect	Timer/Counter1 Overflow
10	\$012	TIMER0_OVF_vect	Timer/Counter0 Overflow
11	\$014	SPI_STC_vect	SPI Serial Transfer Complete
12	\$016	USART_RXC_vect	USART Receive Complete
13	\$018	USART_UDRE_vect	USART Underflow
14	\$01A	USART_TXC_vect	USART Transmit Complete
15	\$01C	ADC_vect	ADC Conversion Complete
16	\$01E	EE_RDY_vect	EEPROM Ready
17	\$020	ANA_COMP_vect	Analog Comparator
18	\$022	TWI_vect	2-wire Serial Interface
19	\$024	INT2_vect	External Interrupt Request 2
20	\$026	TIMER0_COMP_vect	Timer/Counter0 Compare Match
21	\$028	SPM_RDY_vect	Store Program Memory Ready

Vector names
to be used
with ISR

Complete List

Vector No.	Program Address	Interrupt vector name	Description
1	\$000	RESET_vect	Reset
2	\$002	INT0_vect	External Interrupt Request 0
3	\$004	INT1_vect	External Interrupt Request 1
4	\$006	TIMER2_COMP_vect	Timer/Counter2 Compare Match
5	\$008	TIMER2_OVF_vect	Timer/Counter2 Overflow
6	\$00A	TIMER1_CAPT_vect	Timer/Counter1 Capture Event
7	\$00C	TIMER1_COMPA_vect	Timer/Counter1 Compare Match A
8	\$00E	TIMER1_COMPB_vect	Timer/Counter1 Compare Match B
9	\$010	TIMER1_OVF_vect	Timer/Counter1 Overflow
10	\$012	TIMER0_OVF_vect	Timer/Counter0 Overflow
11	\$014	SPI_STC_vect	Serial Transfer Complete
12			USART, Rx Complete
13			USART Data Register Empty
14			USART, Tx Complete
15			ADC Conversion Complete
16			EEPROM Ready
17			Analog Comparator
18			2-wire Serial Interface
19			External Interrupt Request 2
20			Timer/Counter0 Compare Match
21	\$028	SPM_RDY_vect	Store Program Memory Ready

Description

Steps to program an interrupt in C

1. Include header file <avr\interrupt.h>.
2. Use C macro ISR() to declare the interrupt handler and update IVT.
3. Configure details about the interrupt by setting relevant registers.
4. Enable the specific interrupt.
5. Enable the interrupt subsystem globally using sei().

ISR

- Basic construct

```
ISR(interrupt vector name)
{
    //to do logic
}
```

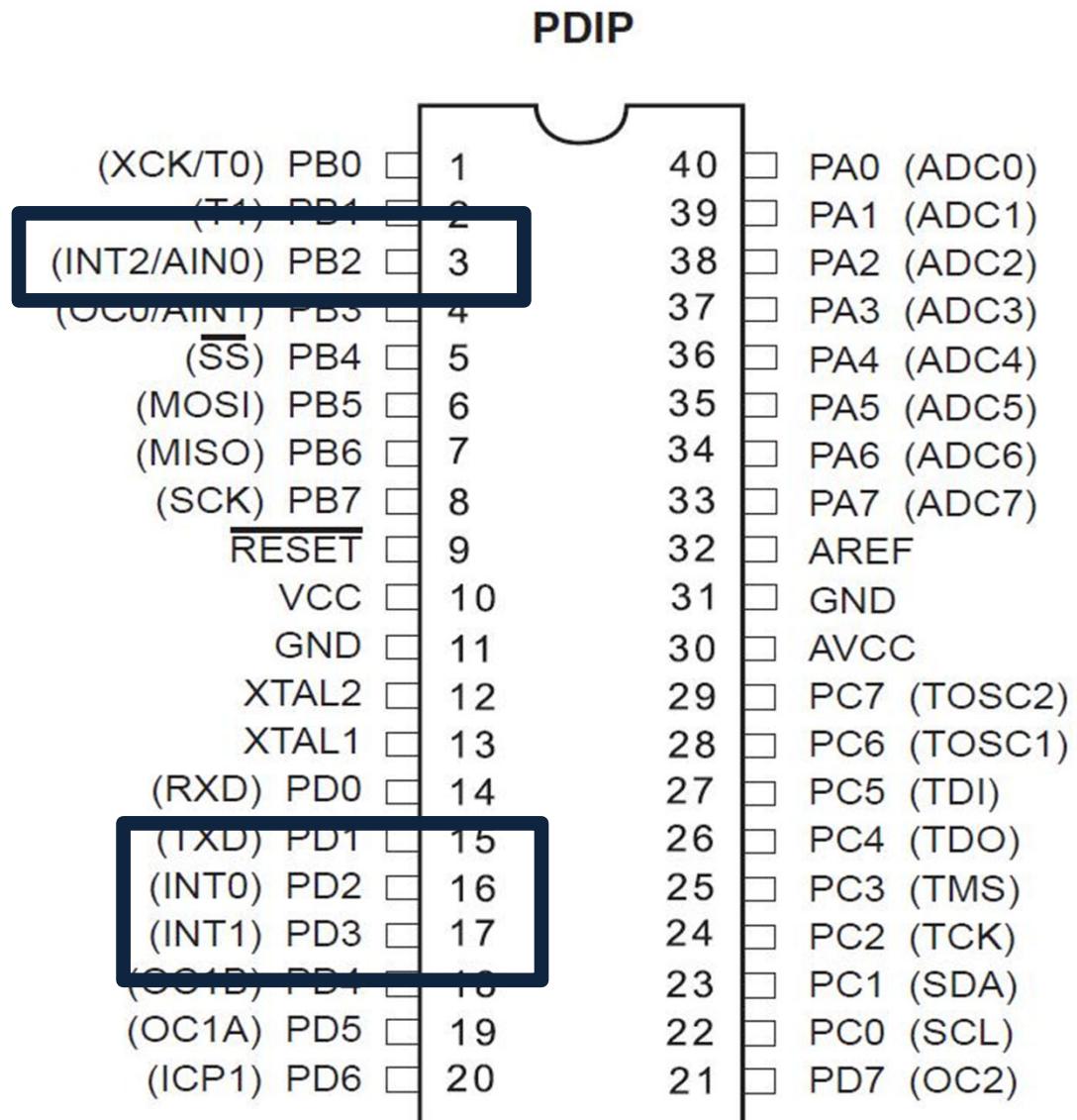
ISR

- To handle external interrupt 1

```
ISR(INT1_vect)
{
    //to do logic
}
```

External Interrupts

- Three external interrupts
 - INT 0
 - INT 1
 - INT 2



External Interrupts

- Key steps in using external interrupts.
 - Specifying what types of event will trigger the interrupt (Step 3)
 - Enabling the interrupt (Step 4)

Specifying Events that Trigger Interrupt (Step 3)

- 2 registers
 - MCU Control Register (For INT0 and INT 1)
 - MCU Control and Status Register (For INT2)

Specifying Events that Trigger Interrupt (Step 3)

Read/Write Initial value	R/W 0	MCUCR						
	SM2	SE	SM1	SM0	ISC11	ISC10	ISC01	ISC00

ISC11	ISC10	Description
0	0	The low level of INT1 generates an interrupt request.
0	1	Any logical change on INT1 generates an interrupt request.
1	0	The falling edge of INT1 generates an interrupt request.
1	1	The rising edge of INT1 generates an interrupt request.

ISC01	ISC00	Description
0	0	The low level of INT0 generates an interrupt request.
0	1	Any logical change on INT0 generates an interrupt request.
1	0	The falling edge of INT0 generates an interrupt request.
1	1	The rising edge of INT0 generates an interrupt request.

Read/Write Initial value	R/W 0	MCUCSR						
	JTD	ISC2	-	JTRF	WDRF	BORF	EXTRF	PORF
0: falling edge generates an interrupt INT2 1: rising edge generates an interrupt INT2								

Specifying Events that Trigger Interrupt (Step 3)

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	MCUCR	
ISC11	ISC10	Description				ISC01	ISC00	Description		
0	0	The low level of INT1 generates an interrupt request.				0	0	The low level of INT0 generates an interrupt request.		
0	1	Any logical change on INT1 generates an interrupt request.				0	1	Any logical change on INT0 generates an interrupt request.		
1	0	The falling edge of INT1 generates an interrupt request.				0	0	The falling edge of INT0 generates an interrupt request.		
1	1	The rising edge of INT1 generates an interrupt request.				1	1	The rising edge of INT0 generates an interrupt request.		

Read/Write	R/W	R/W	R/W
Initial value	0	0	0
	JTD	ISC2	-
	0: falling 1: rising		

To specify that INT1 is triggered on any change in pin D.3

MCUCR = (1<<ISC10);

Enabling the interrupt (Step 4)

- GICR (General Interrupt Control Register)
register is used to enable external interrupts
- To enable Interrupt 1
 - **GICR = (1<<INT1);**
- INT1 is defined in *io.h*

	INT1	INT0	INT2	-	-	-	IVSEL	IVSEL
Read/Write	R/W	R/W	R/W	R	R	R	R/W	R/W
Initial value	0	0	0	0	0	0	0	0

**TOGGLE THE CONTENT OF PORT B,
WHENEVER A CERTAIN SENSOR
CONNECTED TO YOUR SYSTEM GOES
TO LOW STATE**

C Code

```
#include <avr/io.h>
#include <avr/interrupt.h> //STEP1

ISR(INT1_vect)//STEP2
{
    PORTB = ~PORTB;
}
int main(void)
{
    DDRB = 0xFF;
    PORTB = 0b01010101;
    GICR = (1<<INT1); //STEP3
    MCUCR = MCUCR & 0b11110011;//STEP4
    sei();//STEP5
    while(1);
}
```

Disabling global interrupt

- Typically interrupts are turned off while doing a task that should not be interrupted
- One example is reading/writing 16-bit values like TCNT1
 - Details will be discussed when we study Timers
- The *cli()* macro is used to disable all interrupts by clearing the global interrupt mask.

Nested Interrupts

- The global interrupt is disabled by hardware when an interrupt has occurred
 - so by default nested interrupt is disabled in ATmega32
- Global interrupt is set again by the RETI instruction to enable subsequent interrupts
 - This is done automatically by the compiler
- However to enable nested interrupts it can be enabled manually with the sei() in the ISR

ISR Usage

- Understand how often the interrupt occurs
- Understand how much time it takes to service each interrupt
- Make sure there is enough time to service all interrupts while getting work done in the main loop
- Keep ISRs Short and Simple. (short = short time, not short code length)
 - Do only what has to be done. Long ISRs may preclude others from being run

Use of volatile

- As ISRs are not called from main or any other function, it can not take argument or return any values
- We have to use global variables
- We must use volatile to declare such variables
 - why ?

Use of volatile

- Compilers can optimize away some variables
- It does so when it sees that the variable cannot be changed within the scope of the code it is looking at
- Variables changed by the ISR are outside the scope of main()
 - So, they get optimized away

Use of volatile

```
volatile uint8_t tick; //keep tick out of regs!
```

```
ISR(TIMER1_OVF_vect){  
    tick++; //increment my tick count  
}
```

```
main()  
{  
    while(tick == 0x00)  
    {  
        bla, bla, bla...  
    }  
    more bla, bla  
}
```

Without volatile modifier, optimization may remove tick because nothing in while loop can ever change tick

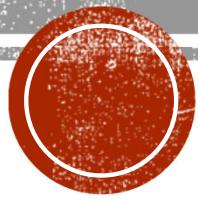
Resource

- ATMega Datasheet
- <http://web.engr.oregonstate.edu/~traylor/ec/e473/lectures/interrupts.pdf>
- <http://www.avrfreaks.net/forum/nested-interrupts-2>

USING ADC IN ATMEGA32

CSE 315

Courtesy: Muhammad Ali Nayeem, Azad Abdus Salam,
Md. Iftekharul Islam Sakib



LEARNING OBJECTIVE

- Why do we need ADC?
- What is ADC?
- ADC in ATmega32
- A simple example



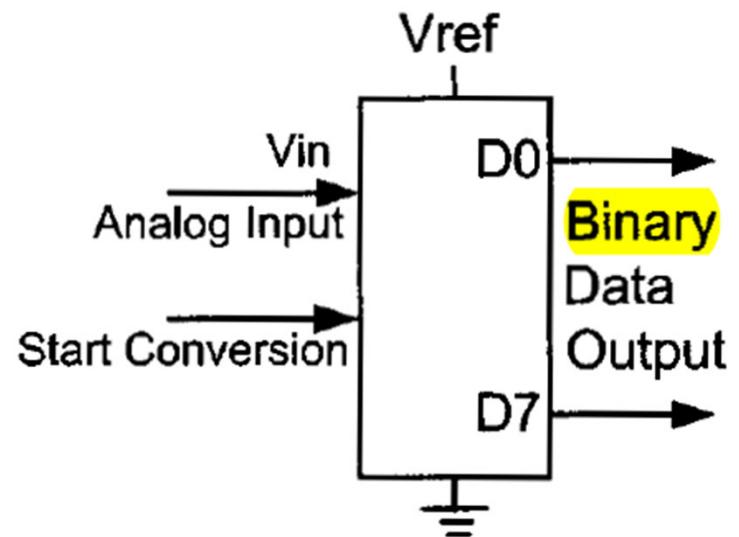
RECAP

- MCU is a **digital** processor
 - Works with binary data
 - Just 2 voltage levels
- MCU needs to collect **information** from the real world
 - Temperature, Pressure, Humidity, Sound ...
 - Continuous property
 - Infinite possible values
- Why do we collect information?
 - To analyze the situation
 - To take appropriate actions

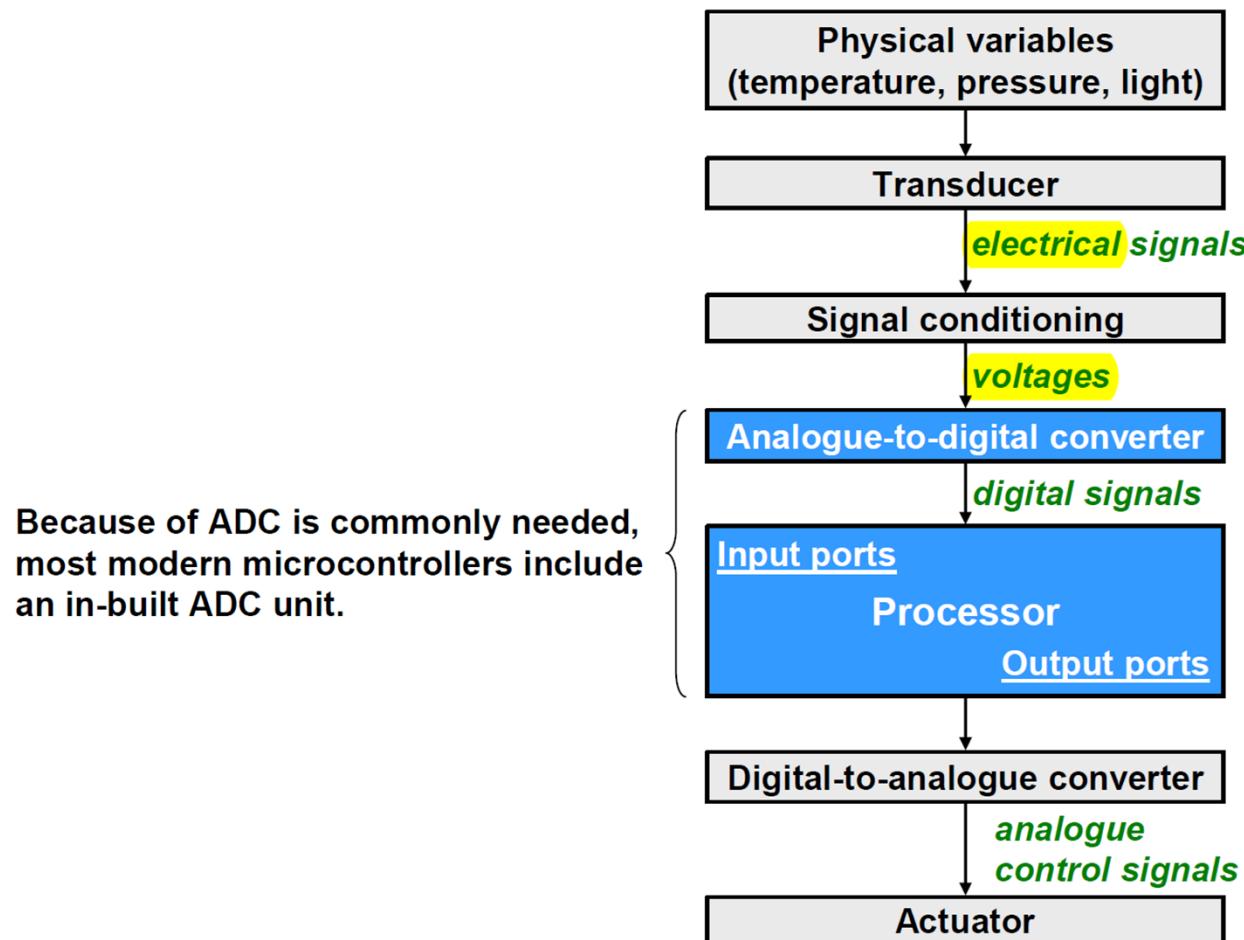


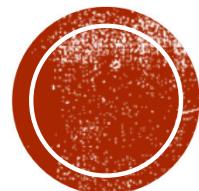
RECAP

- But how to collect information?
 - Sensors: Convert physical property into **analog** signal
 - Voltage, Current, Resistance, ...
- How can MCU recognize these signal?
 - Need to convert into digital data
 - How?
 - Approximation
 - Need something special!



TYPICAL EMBEDDED APPLICATION





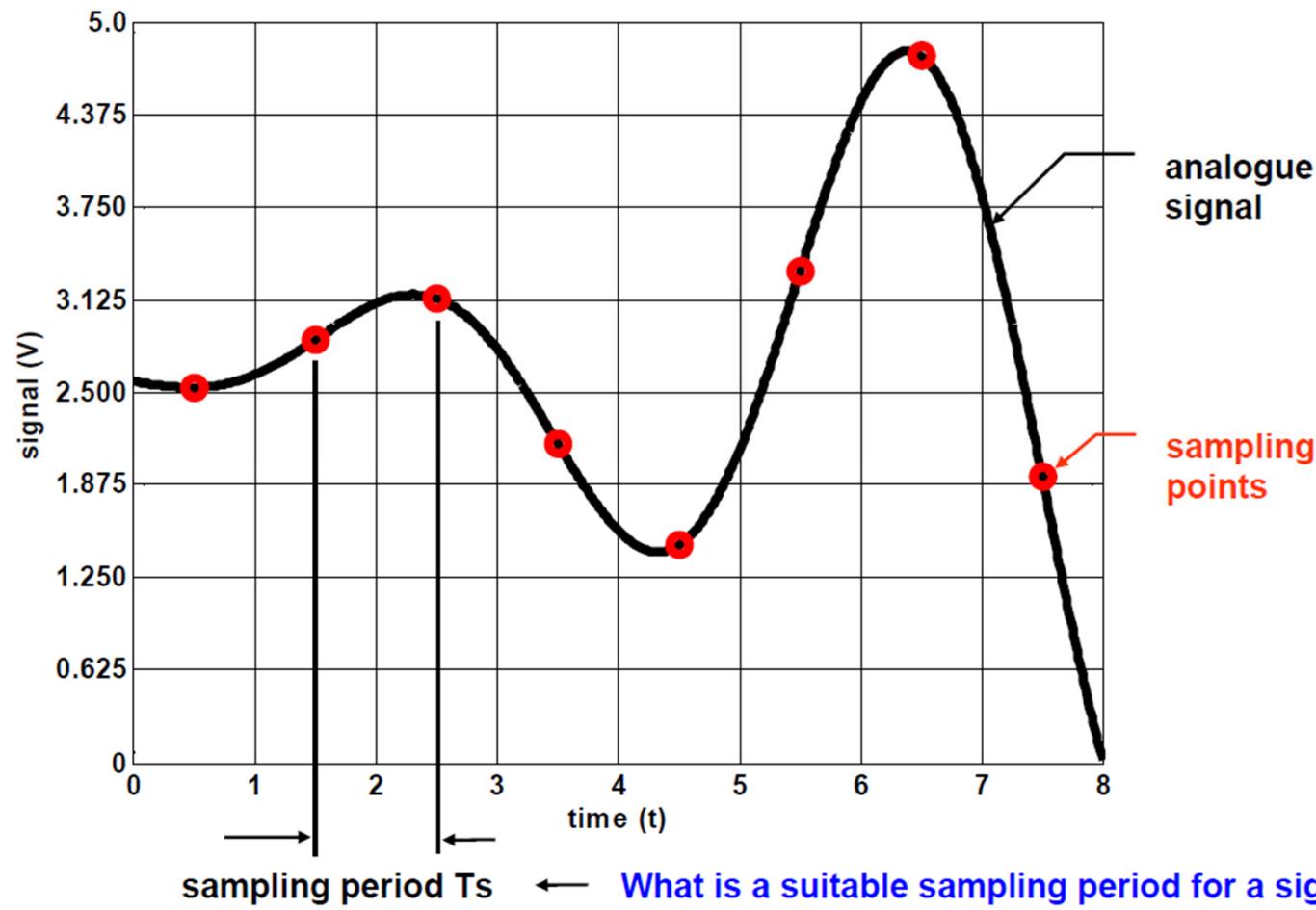
ADC: QUICK REVIEW

A-TO-D CONVERSION

- Two related steps
 - 1. Sampling (Approximating the X-axis)
 - the analogue signal is extracted at **regularly** spaced time instants
 - the samples have **real** values
 - 2. Quantization (Approximating the Y-axis)
 - the samples are quantized to **discrete levels**
 - each sample is represented as a **binary** value

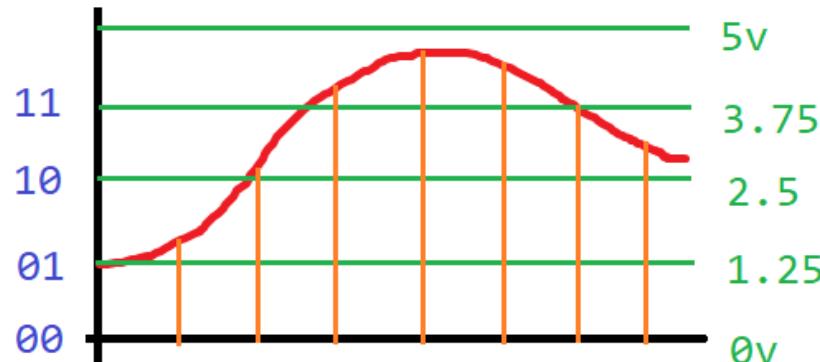


SAMPLING AN ANALOGUE SIGNAL

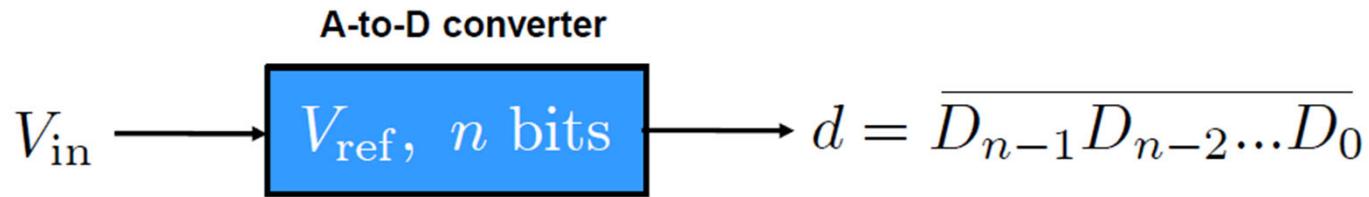


QUANTIZING THE SAMPLED SIGNAL

- How many bit to represent the analogue value?
- Say, 2 bit to represent analogue value from 0V to 5V
- Step size = $5V / 2^2 = 5V / 4 = 1.25V$
- Digital value = $\text{floor}(\text{Analogue value} / 1.25V)$



QUANTIZING THE SAMPLED SIGNAL



- Let's consider an n-bit ADC.
- Let V_{ref} be the **reference voltage**.
- Let V_{in} be the analogue input voltage.
- Let V_{min} be the minimum allowable input voltage, usually $V_{min} = 0$.
- The ADC's digital output, $d = D_{n-1}D_{n-2} \dots D_0$, is given as

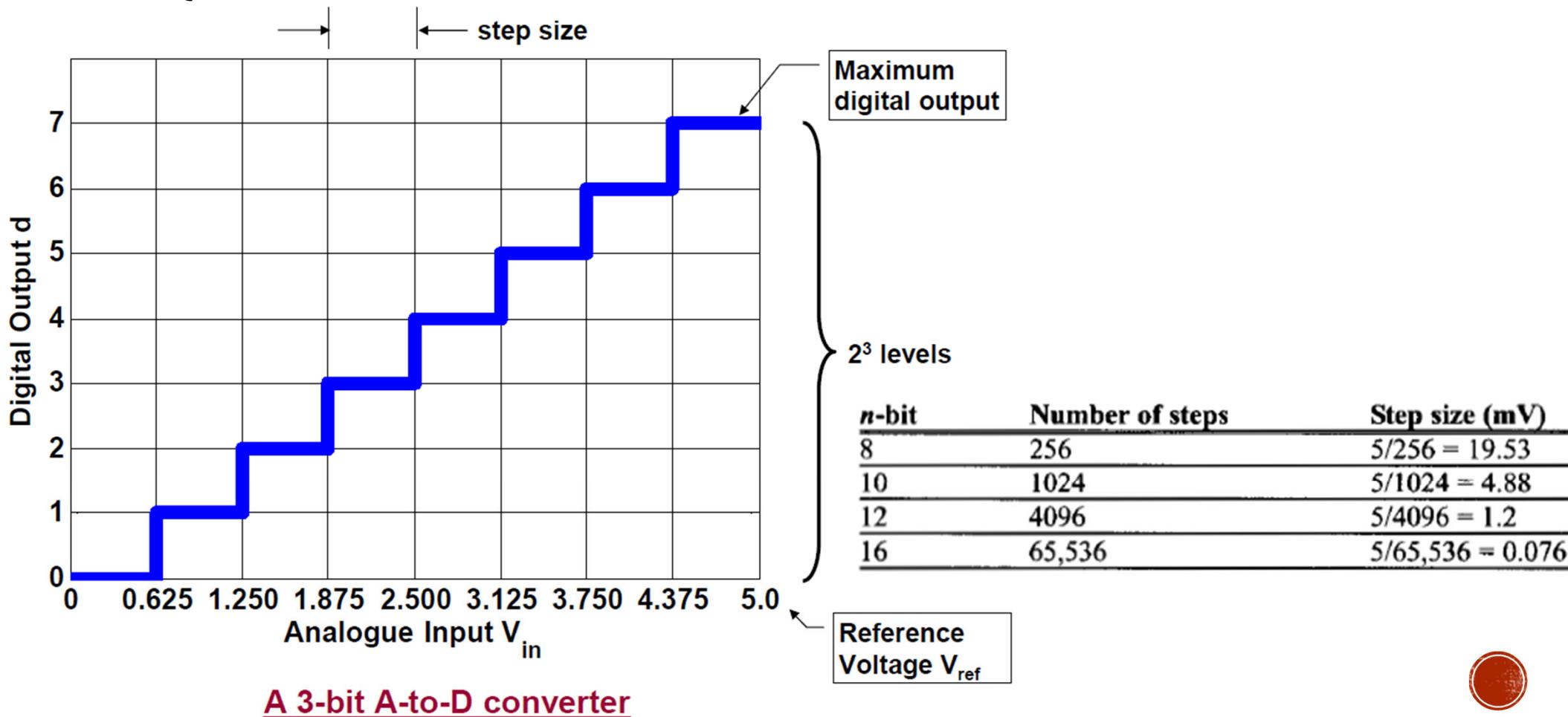
$$d = \text{round down} \left[\frac{V_{in} - V_{min}}{\text{step size}} \right]$$

- The **step size (resolution)** is the smallest change in input that can be discerned by the ADC:

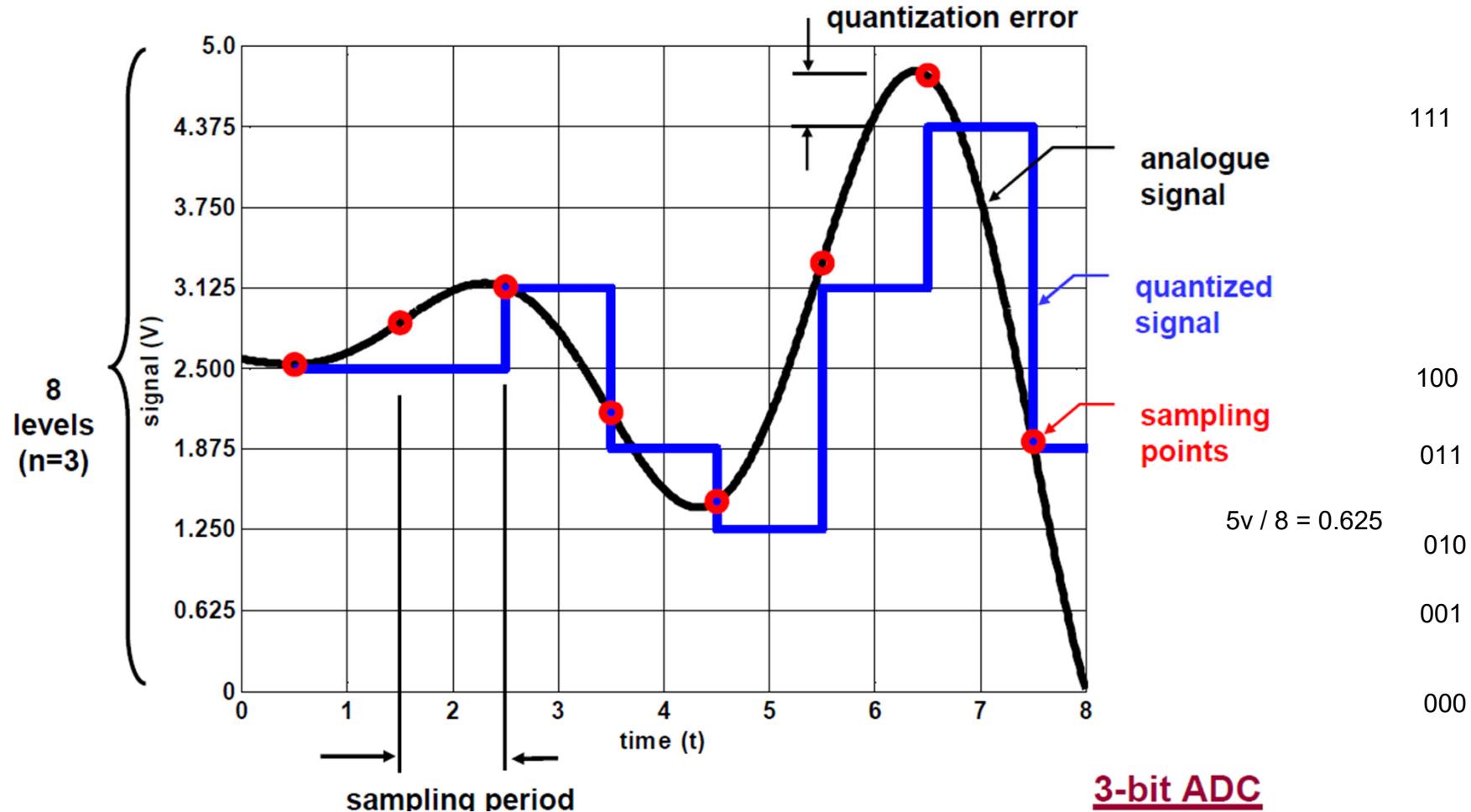
$$\text{step size} = \frac{V_{ref} - V_{min}}{2^n}$$



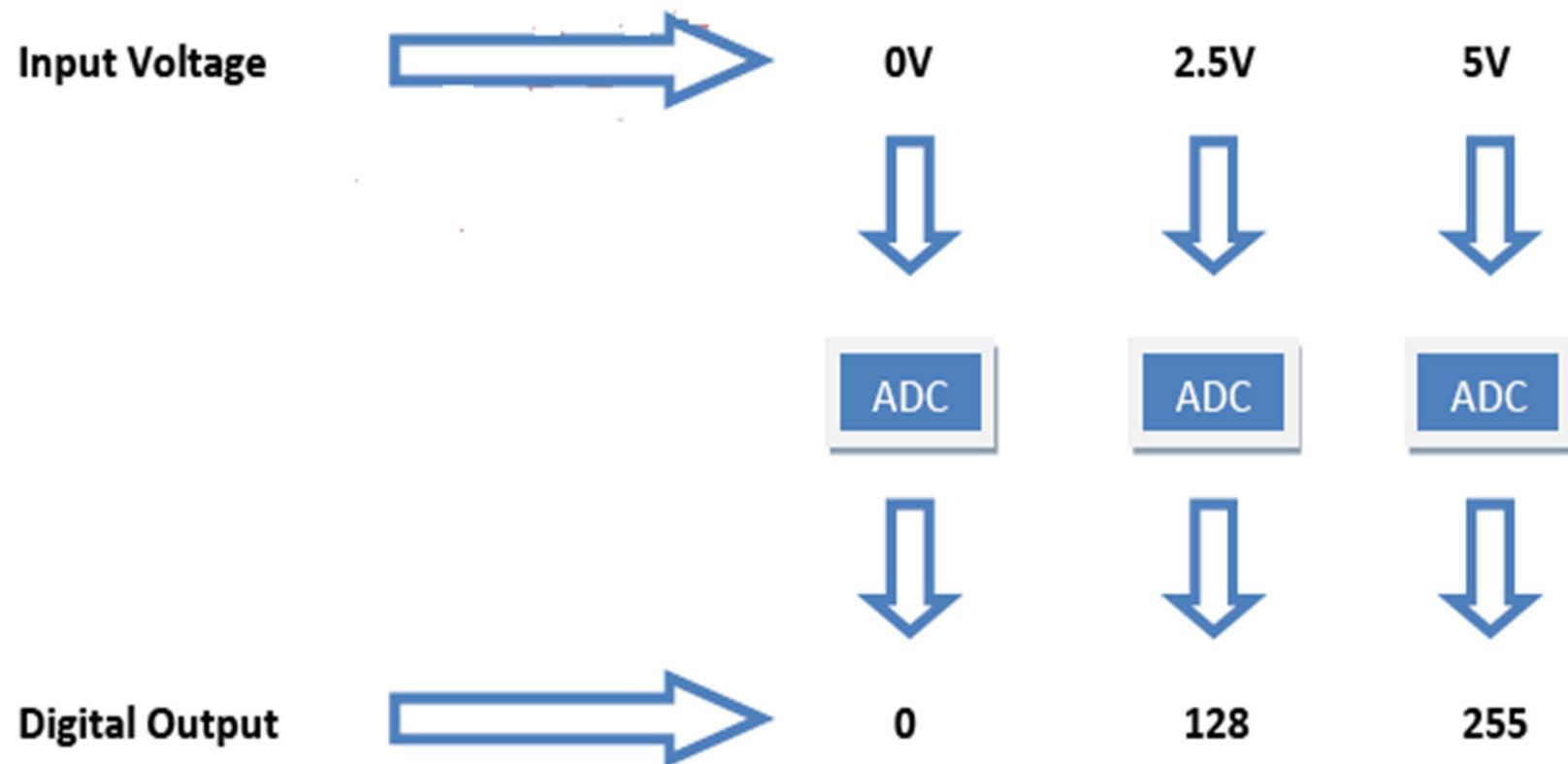
QUANTIZING THE SAMPLED SIGNAL



QUANTIZING THE SAMPLED SIGNAL



8-BIT ADC



EXAMPLE

For an 8-bit ADC, we have $V_{ref} = 2.56$ V. Calculate the D0–D7 output if the analog input is: (a) 1.7 V, and (b) 2.1 V.

Solution:

Because the step size is $2.56/256 = 10$ mV, we have the following:

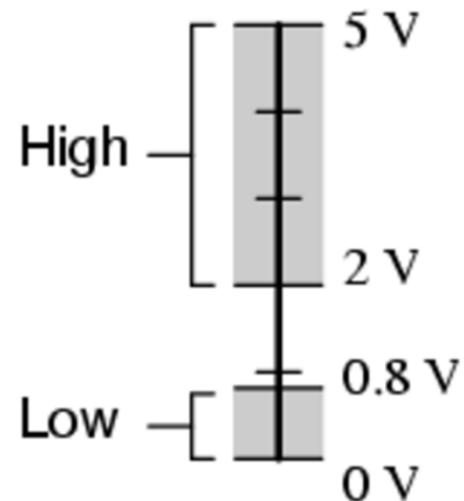
- (a) $D_{out} = 1.7$ V/10 mV = 170 in decimal, which gives us 10101010 in binary for D7–D0.
- (b) $D_{out} = 2.1$ V/10 mV = 210 in decimal, which gives us 11010010 in binary for D7–D0.



A QUESTION

- What is the difference between an ADC and a digital input pin?

*Acceptable TTL gate
input signal levels*



THE ADC IN ATMEGA32

- The ADC has a 10-bit resolution.
 - The binary output has $n = 10$ bits
- The ADC has 8 input channels
 - Analogue input can come from **8 different sources**
 - However, it performs conversion on **only one channel** at a time
- If default reference voltage $V_{ref} = 5V$ is used
 - step size: $5 V / 1024 = 4.88mV$
- The clock rate of the ADC can be different from the CPU clock rate
 - One ADC conversion takes 13 ADC cycles
 - An ADC prescaler will decide the ADC clock rate

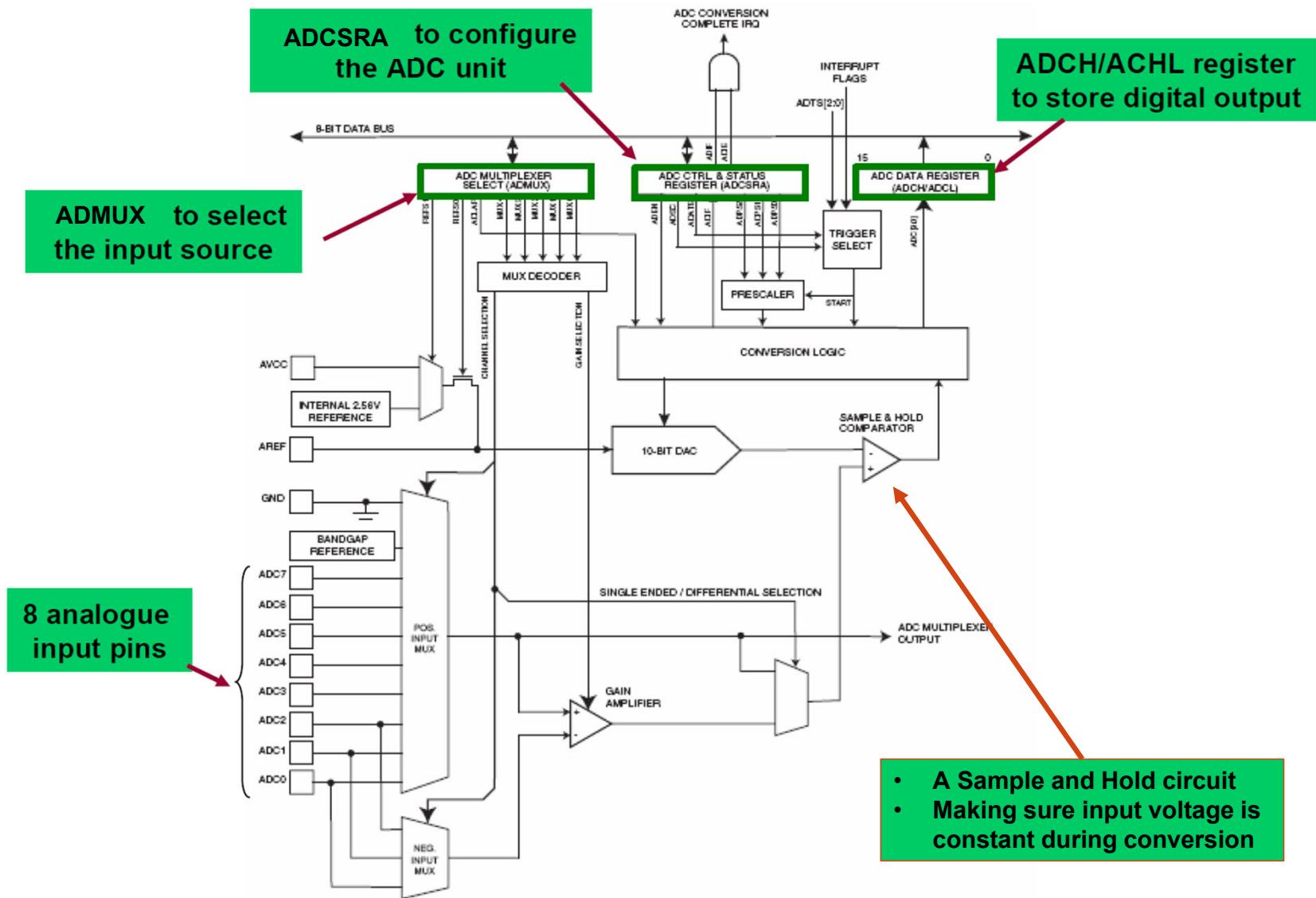


RELEVANT PINS

The diagram shows the pinout of a microcontroller with 40 pins. The pins are numbered 1 through 40. The pins are grouped into three columns: PB (pins 1-8), PA (pins 9-16), and PC/PD (pins 17-40). The PA column is highlighted with a green box and labeled "8 ADC input pins". The AREF pin (pin 9) is highlighted with a grey box and labeled "External V_{ref}". The AVCC pin (pin 11) is highlighted with a grey box and labeled "AVCC".

(XCK/T0)	PB0		1	40	PA0 (ADC0)
(T1)	PB1		2	39	PA1 (ADC1)
(INT2/AIN0)	PB2		3	38	PA2 (ADC2)
(OC0/AIN1)	PB3		4	37	PA3 (ADC3)
(SS)	PB4		5	36	PA4 (ADC4)
(MOSI)	PB5		6	35	PA5 (ADC5)
(MISO)	PB6		7	34	PA6 (ADC6)
(SCK)	PB7		8	33	PA7 (ADC7)
RESET			9	32	AREF
VCC			10	31	GND
GND			11	30	AVCC
XTAL2			12	29	PC7 (TOSC2)
XTAL1			13	28	PC6 (TOSC1)
(RXD)	PD0		14	27	PC5 (TDI)
(TXD)	PD1		15	26	PC4 (TDO)
(INT0)	PD2		16	25	PC3 (TMS)
(INT1)	PD3		17	24	PC2 (TCK)
(OC1B)	PD4		18	23	PC1 (SDA)
(OC1A)	PD5		19	22	PC0 (SCL)
(ICP1)	PD6		20	21	PD7 (OC2)

BLOCK DIAGRAM

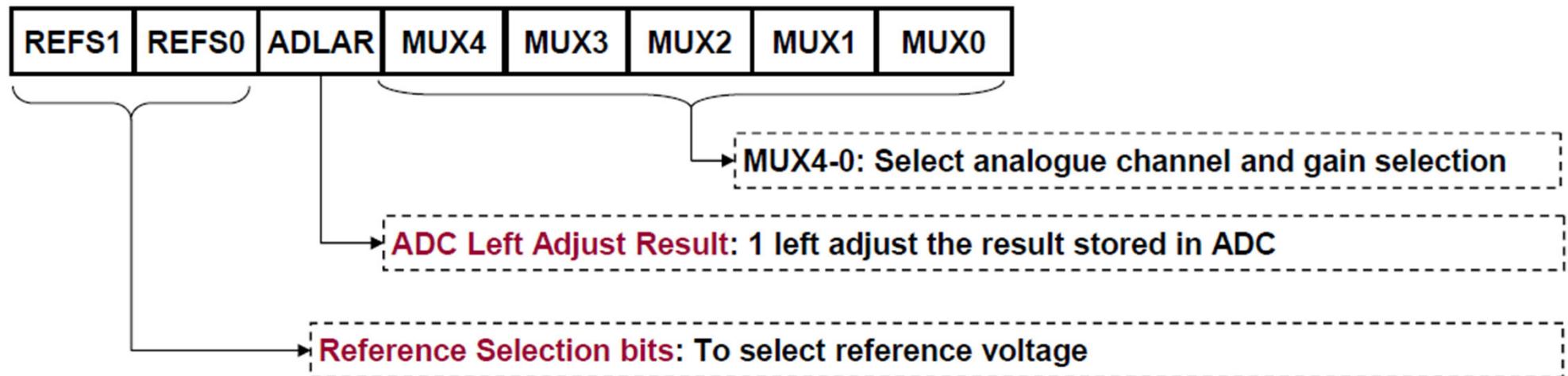


MAIN ASPECTS

- What are the relevant ADC registers?
 - ADMUX
 - ADCH/ADCL
 - ADCSRA
 - **SFIOR**
- What are the steps to use the ADC?
- How to use ADC interrupt?



ADC MULTIPLEXER SELECTION REGISTER (ADMUX)



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

SELECTING REFERENCE VOLTAGE V_{REF}

Table 11.1: ADC reference voltage selection

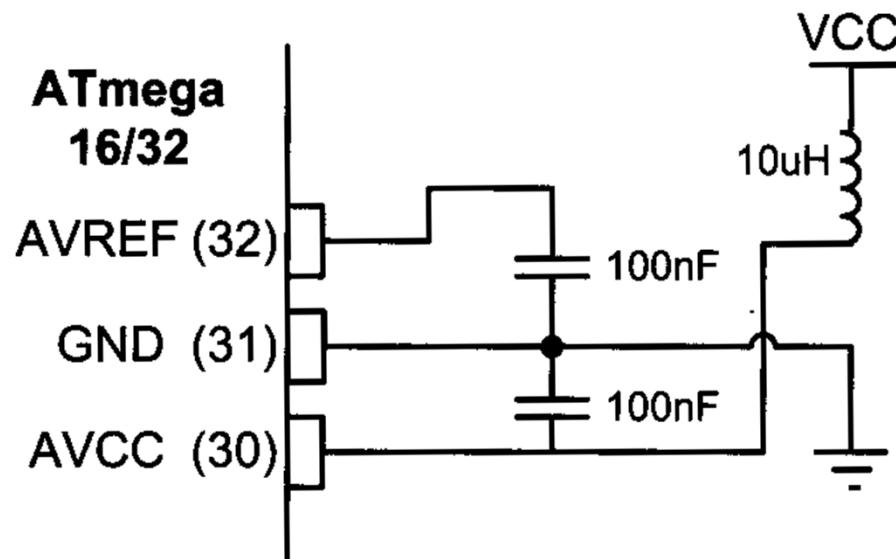
REFS1	REFS0	Voltage Reference Selection
0	0	AREF, Internal Vref turned off
0	1	AVCC with external capacitor at AREF pin
1	0	Reserved
1	1	Internal 2.56V Voltage Reference with external capacitor at AREF pin

- Usually, mode 01 is used: $AVCC = 5V$ as reference voltage.
 AVCC must not differ more than $\pm 0.3V$ from V_{cc} .
- However, if the input voltage has a different dynamic range, we can use mode 00 to select an external reference voltage.



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

ADC RECOMMENDED CONNECTION for Vref = AVCC



<https://electronics.stackexchange.com/questions/86207/atmega8-why-do-vcc-and-avcc-have-to-be-connected>



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

SELECTING INPUT SOURCE

MUX4..0	Single Ended Input	Positive Differential Input	Negative Differential Input	Gain
00000	ADC0			
00001	ADC1			
00010	ADC2			
00011	ADC3			
00100	ADC4			
00101	ADC5			
00110	ADC6			
00111	ADC7			

N/A



MUX4..0	Single Ended Input	Positive Differential Input	Negative Differential Input	Gain
00000	ADC0			
00001	ADC1			
00010	ADC2			
00011	ADC3			
00100	ADC4			
00101	ADC5			
00110	ADC6			
00111	ADC7			
01000		ADC0	ADC0	10x
01001		ADC1	ADC0	10x
01010		ADC0	ADC0	200x
01011		ADC1	ADC0	200x
01100		ADC2	ADC2	10x
01101		ADC3	ADC2	10x
01110		ADC2	ADC2	200x
01111		ADC3	ADC2	200x
10000		ADC0	ADC1	1x
10001		ADC1	ADC1	1x
10010		ADC2	ADC1	1x
10011		ADC3	ADC1	1x
10100		ADC4	ADC1	1x
10101		ADC5	ADC1	1x
10110		ADC6	ADC1	1x
10111		ADC7	ADC1	1x
11000		ADC0	ADC2	1x
11001		ADC1	ADC2	1x
11010		ADC2	ADC2	1x
11011		ADC3	ADC2	1x
11100		ADC4	ADC2	1x

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

SELECTING INPUT SOURCE

Why select same ADC source as both positive input and negative input?
<https://www.avrfreaks.net/forum/pos-differential-input-neg-differential-input>



Bit	7	6	5	4	3	2	1	0	ADMUX
	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	

ADC Output

For single ended conversion, the result is

$$ADC = \frac{V_{IN} \cdot 1024}{V_{REF}}$$

If differential channels are used, the result is

$$ADC = \frac{(V_{POS} - V_{NEG}) \cdot GAIN \cdot 512}{V_{REF}}$$

Differential ADC is presented in two's complement form, from 0x200 (-512d) through 0x1FF (+511d).

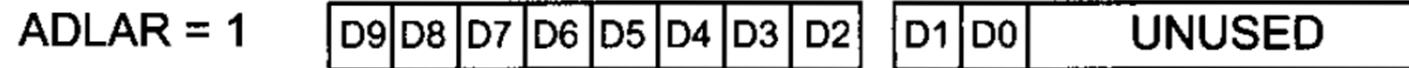


Bit	7	6	5	4	3	2	1	0	ADMUX
Read/Write	R/W								
Initial Value	0	0	0	0	0	0	0	0	

ADCH & ADCL REGISTERS

ADCH

Left-Justified



Right-Justified

- Why two separate registers?
 - Fast read of only one register with a bit of sacrifice on accuracy.
- What is the maximum error while reading only ADCH with ADLAR=1?



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

Read ADCL Before ADCH

- ADCL must be read first, then ADCH, to ensure that the content of the Data Registers belongs to the same conversion.
- Once ADCL is read, ADC access to Data Registers is blocked, i.e. value of ADCL and ADCH will not be changed until ADCH is read.
- This means that if ADCL has been read, and a conversion completes before ADCH is read, neither register is updated and the result from the conversion is lost.
- When ADCH is read, ADC access to the ADCH and ADCL Registers is re-enabled.



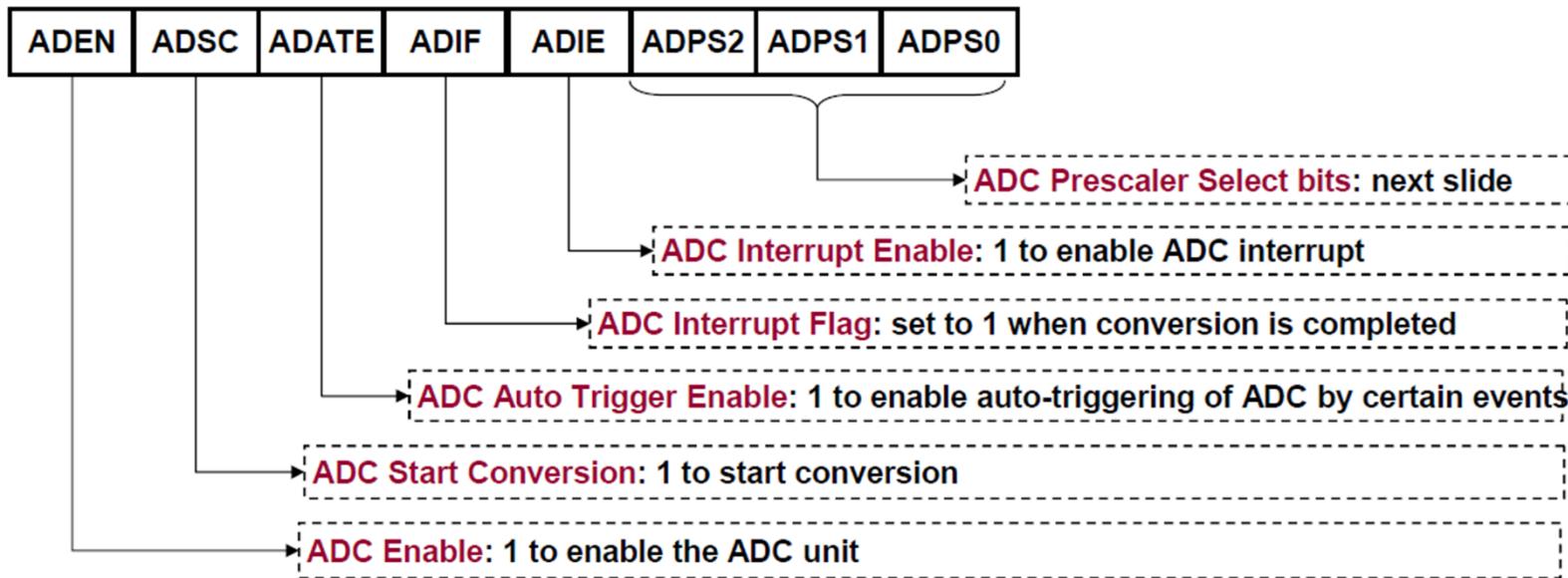
Bit	7	6	5	4	3	2	1	0	ADMUX
Read/Write	R/W								
Initial Value	0	0	0	0	0	0	0	0	

Read ADCL Before ADCH

- Can you read only ADCH?
- Can you read only ADCL?



ADC CONTROL AND STATUS REGISTER (ADCSRA)



- ADC unit can operate in two modes: manual or auto-trigger.
- In manual mode, set bit ADSC will start conversion.
- In auto-trigger mode, an predefined event will start conversion.



ADC CLOCK

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

Table 11.3: ADC Prescaler Selection

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

- The clock of the ADC is obtained by dividing the CPU clock and a division factor.
- There are 8 possible division factors, decided by the three bits {ADPS2, ADPS1, ADPS0}
- Example:** Using internal clock of 1Mz and a ADC prescaler bits of '010', the clock rate of ADC is: $1\text{MHz}/4 = 250\text{KHz}$



STEPS TO USE THE ADC

- **Step 1:** Configure the ADC using registers ADMUX, ADCSRA, and SFIOR.
 - What is the ADC source?
 - What reference voltage to use?
 - Align left or right the result in ADCH, ADCL?
 - Enable or disable ADC auto-trigger?
 - Enable or disable ADC interrupt?
 - What is the prescaler?
- **Step 2:** Start ADC operation
 - Write 1 to flag ADSC of register ADCCSRA.
- **Step 3:** Extract ADC result
 - Wait until flag ADSC becomes 0.
 - Read result from registers ADCL and then ADCH.



SIMPLE EXAMPLE

Write C program that repeatedly performs ADC on a sinusoidal signal and displays the result on LEDs.

■ Step 1: Configure the ADC

- What is the ADC source? **ADC0(pin A.0)**
- What reference voltage to use? **AVCC = 5V**
- Align left or right? **Left, top 8-bit in ADCH**
- Enable or disable ADC auto-trigger? **Disable**
- Enable or disable ADC interrupt? **Disable**
- What is the prescaler? **4 (010)**



SIMPLE EXAMPLE

■ Step 1: Configure the ADC

- What is the ADC source? ADC0(pin A.0)
- What reference voltage to use? AVCC = 5V
- Align left or right? Left, top 8-bit in ADCH
- Enable or disable ADC auto-trigger? Disable
- Enable or disable ADC interrupt? Disable
- What is the prescaler? 4 (010)

0	1	1	0	0	0	0	0	0	ADCMUX
REFS1	REFS0	ADLAR	MUX4	MUX3	MUX2	MUX1	MUX0		

1	0	0	0	0	0	1	0	ADCCSRA
ADEN	ADSC	ADATE	ADIF	ADIE	ADPS2	ADPS1	ADPS0	

■ Steps 2 and 3: Show next in C program.



```

#include<avr/io.h>
int main (void){
    unsigned char result;

    DDRB = 0xFF; // set port B for output

    // Configure the ADC module of the ATmega16
    ADMUX = 0b01100000; // REFS1:0 = 01 -> AVCC as reference,
                        // ADLAR = 1 -> Left adjust
                        // MUX4:0 = 00000 -> ADC0 as input

    ADCSRA = 0b10000001; // ADEN = 1: enable ADC,
                        // ADSC = 0: don't start conversion yet
                        // ADATE = 0: disable auto trigger,
                        // ADIE = 0: disable ADC interrupt
                        // ASPS2:0 = 001: prescaler = 2

    while(1){          // main loop
        // Start conversion by setting flag ADSC
        ADCSRA |= (1 << ADSC);           2

        // Wait until conversion is completed
        while (ADCSRA & (1 << ADSC)){;}  3

        // Read the top 8 bits, output to PORTB
        result = ADCH;
        float voltage = (result << 2) * 5 / 1024;
        PORTB = ~result;
    }
    return 0;
}

```

Bit	7	6	5	4	3	2	1	0	
Read/Write	REFS1	REFS0	ADLAR	MUX4	MUX3	MUX2	MUX1	MUX0	ADMUX
Initial Value	0	0	0	0	0	0	0	0	
Bit	7	6	5	4	3	2	1	0	
Read/Write	ADEN	ADSC	ADATE	ADIF	ADIE	ADPS2	ADPS1	ADPS0	ADCSRA
Initial Value	0	0	0	0	0	0	0	0	



What if you want to do some other work while ADC going on?

SIMPLE EXAMPLE: USING ADC INTERRUPT

Write interrupt-driven program to digitise a sinusoidal signal and display the result on LEDs.

■ Step 1: Configure the ADC

- What is the ADC source? **ADC0**
- What reference voltage to use? **AVCC = 5V**
- Align left or right? **Left, top 8-bit in ADCH**
- Enable or disable ADC auto-trigger? **Disable**
- Enable or disable ADC interrupt? **Enable**
- What is the prescaler? **2 (fastest conversion)**

■ Step 2: Start ADC operation

■ Step 3: In ISR, read and store ADC result.



```

#include<avr/io.h>
#include<avr/interrupt.h>

volatile unsigned char result;

ISR(ADC_vect){
    result = ADCH; // Read the top 8 bits, and store in variable result
}

int main (void){
    DDRB = 0xFF; // set port B for output

    // Configure the ADC module of the ATmega16
    ADMUX = 0b01100000; // REFS1:0 = 01 -> AVCC as reference,
                        // ADLAR   = 1   -> Left adjust
                        // MUX4:0  = 00000 -> ADC0 as input

    ADCSRA = 0b10001111; // ADEN = 1: enable ADC,
                        // ADSC = 0: don't start conversion yet
                        // ADATE = 0: disable auto trigger,
                        // ADIE  = 1: enable ADC interrupt
                        // ASPS2:0 = 002: prescaler = 2

    sei();           // enable interrupt system globally
    while(1){       // main loop
        ADCSRA |= (1 << ADSC); // start a conversion
        PORTB = ~result;      // display on port B
    }
    return 0;
}

```

Bit	7	6	5	4	3	2	1	0	
Read/Write	REFS1	REFS0	ADLAR	MUX4	MUX3	MUX2	MUX1	MUX0	ADMUX
Initial Value	0	0	0	0	0	0	0	0	

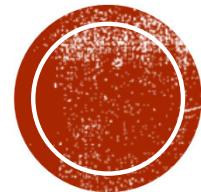
Bit	7	6	5	4	3	2	1	0	
Read/Write	ADEN	ADSC	ADATE	ADIF	ADIE	ADPS2	ADPS1	ADPS0	ADCSRA
Initial Value	0	0	0	0	0	0	0	0	

3

1

2





Measuring Temperature using LM35 Sensor

LM35

- $10\text{mV}/^\circ\text{C}$



TEMPERATURE SENSOR

Part	Temperature Range	Accuracy	Output Scale
LM35A	−55 C to +150 C	+1.0 C	10 mV/C
LM35	−55 C to +150 C	+1.5 C	10 mV/C
LM35CA	−40 C to +110 C	+1.0 C	10 mV/C
LM35C	−40 C to +110 C	+1.5 C	10 mV/C
LM35D	0 C to +100 C	+2.0 C	10 mV/C

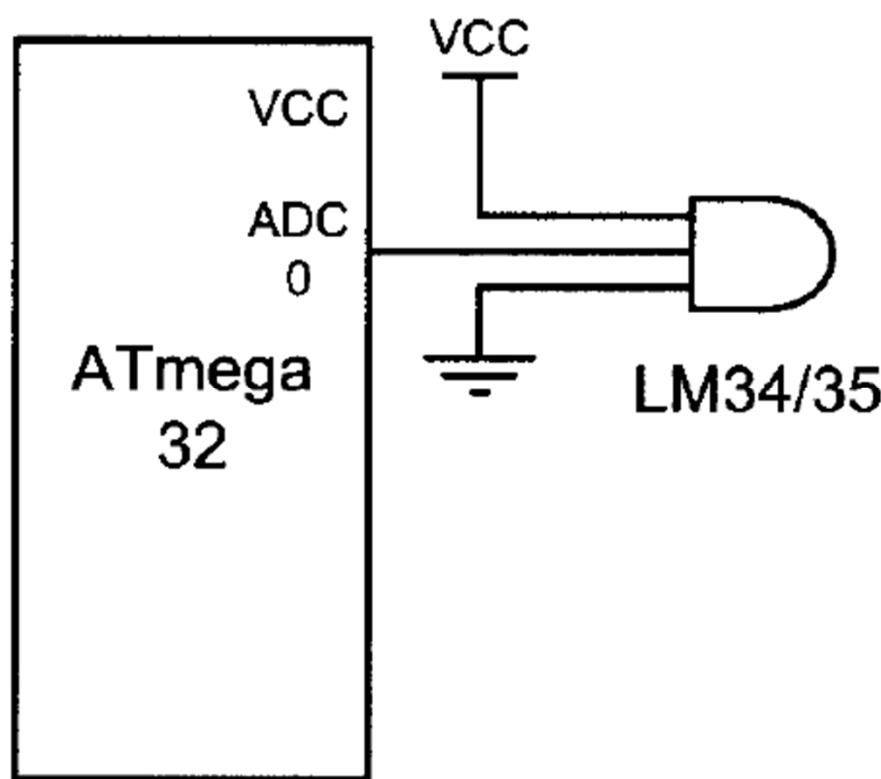


TEMPERATURE SENSOR

Part Scale	Temperature Range	Accuracy	Output
LM34A	-50 F to +300 F	+2.0 F	10 mV/F
LM34	-50 F to +300 F	+3.0 F	10 mV/F
LM34CA	-40 F to +230 F	+2.0 F	10 mV/F
LM34C	-40 F to +230 F	+3.0 F	10 mV/F
LM34D	-32 F to +212 F	+4.0 F	10 mV/F



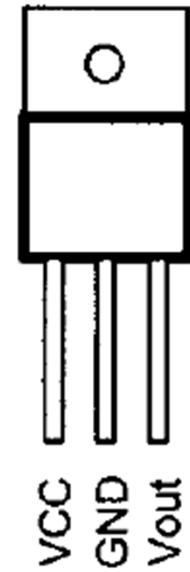
INTERFACING LM34 TO AVR



Bottom View
TO92
Package



Top View
TO220
Package



INTERFACING LM35 TO AVR

- If we use the internal 2.56 V reference voltage
 - step size would be $2.56 \text{ V} / 1024 = 2.5 \text{ mV}$
 - This makes the binary output of the ADC four times the real temperature
 - Divide the binary output by 4 to get the real temperature
 - Step size = $2.56 / 1024$
 - Volt = Step size * ADC
 - Mv = volt * 1000
 - Temp = Mv / 10
- $\text{Temp} = (2.56 / 1024) * \text{ADC} * 1000 / 10 = 0.25 * \text{ADC} = \text{ADC} / 4 \sim \text{ADC} \gg 2$
- $(2.56 * 1000) / (1024 * 10) = 0.25 = 1/4$



INTERFACING LM34 TO AVR

Temp. (F)	V _{in} (mV)	# of steps	Binary V _{out} (b9–b0)	Temp. in Binary
0	0	0	00 00000000	000000000
1	10	4	00 00000100	000000001
2	20	8	00 00001000	000000010
3	30	12	00 00001100	000000011
10	100	20	00 00101000	000010100
20	200	80	00 01010000	000101000
30	300	120	00 01111000	000111100
40	400	160	00 10100000	001010000
50	500	200	00 11001000	001100100
60	600	240	00 11110000	001111000
70	700	300	01 00011000	010001100
80	800	320	01 01000000	010100000
90	900	360	01 01101000	010110100
100	1000	400	01 10010000	011001000



INTERFACING LM34 TO AVR

```
#include <avr/io.h> //standard AVR header
int main (void)
{
    DDRB = 0xFF;                      //make Port B an output
    ADCSRA = 0x87;                    //make ADC enable and select ck/128
    ADMUX = 0xE0;                     //2.56 V Vref and ADC0 single-ended
                                       //data will be left-justified
    while (1 ){
        ADCSRA |= (1<<ADSC);          //start conversion
        while((ADCSRA & (1<<ADIF))==0); //wait for end of conversion
        PORTB = ADCH;                  //give the high byte to PORTB
    }
    return 0;
}
```



SPECIAL FUNCTION IO REGISTER (SFIOR)

ADC Auto Trigger

Source bits

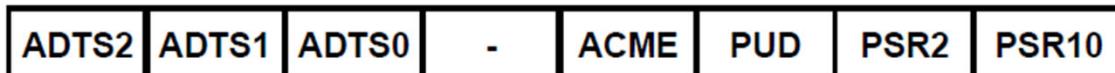


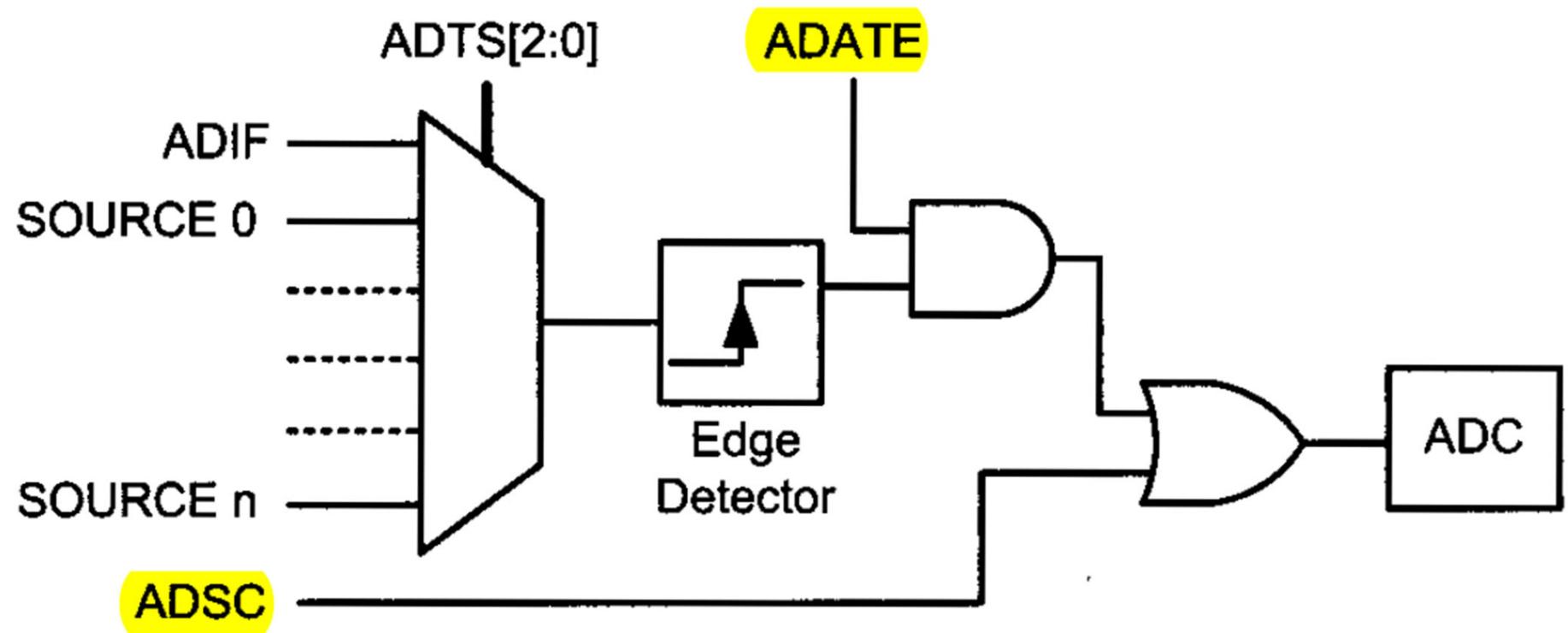
Table 11.4: ADC Auto Trigger Source

ADTS2	ADTS1	ADTS0	Trigger Source
0	0	0	Free Running mode
0	0	1	Analog Comparator
0	1	0	External Interrupt Request 0
0	1	1	Timer/Counter0 Compare Match
1	0	0	Timer/Counter0 Overflow
1	0	1	Timer/Counter1 Compare Match B
1	1	0	Timer/Counter1 Overflow
1	1	1	Timer/Counter1 Capture Event

- Three bits in register SFIOR specify the event that will auto-trigger an A-to-D conversion.



AUTO TRIGGER LOGIC

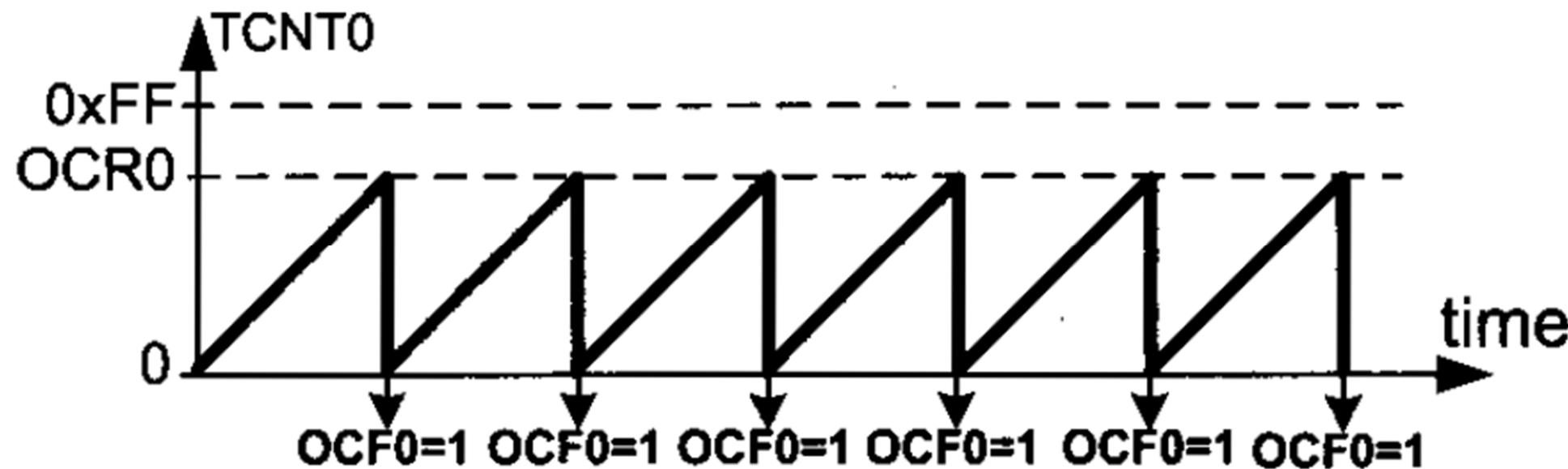


AUTO TRIGGER SOURCE

ADTS2	ADTS1	ADTS0	Trigger Source
0	0	0	Free Running mode
0	0	1	Analog Comparator
0	1	0	External Interrupt Request 0
0	1	1	Timer/Counter0 Compare Match
1	0	0	Timer/Counter0 Overflow
1	0	1	Timer/Counter1 Compare Match B
1	1	0	Timer/Counter1 Overflow
1	1	1	Timer/Counter1 Capture Event

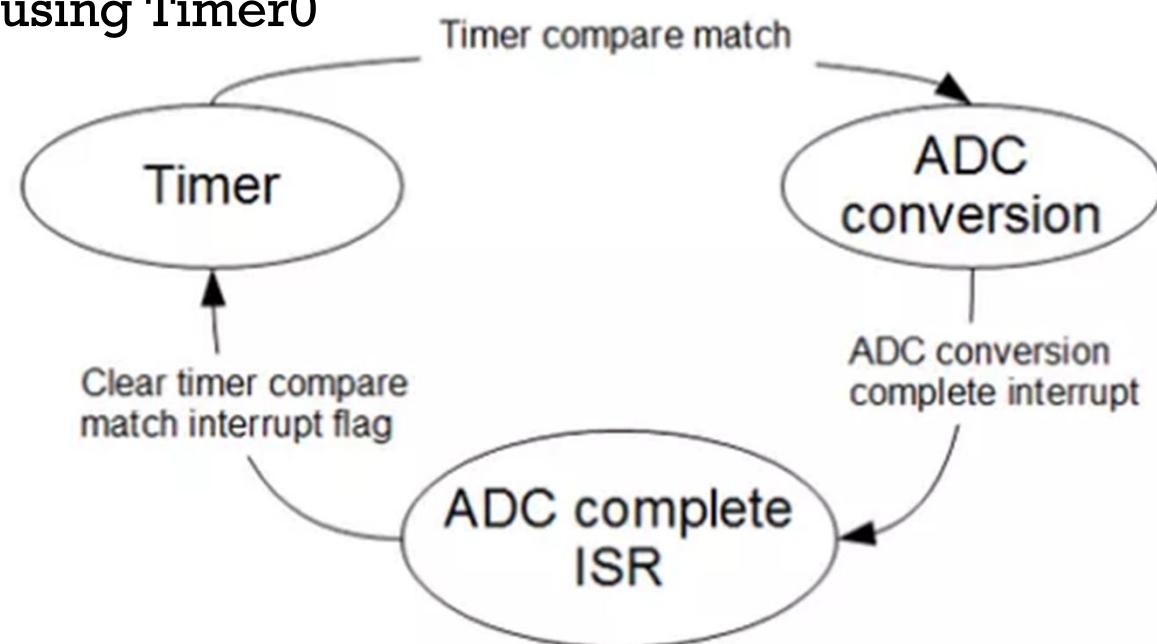


TIMER: NORMAL VS. CTC MODE



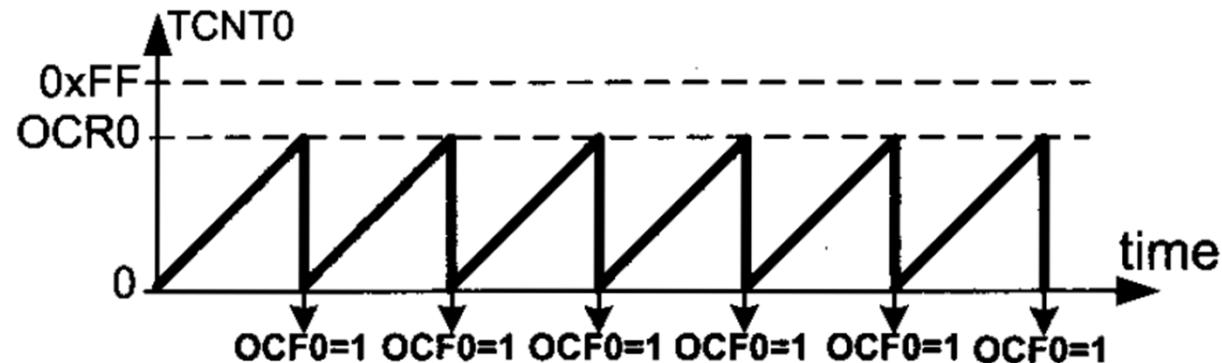
A PRACTICAL PROBLEM

- We need to sample an analog signal @ 20 kHz sampling frequency
- The only way to do this correctly
 - use auto-triggering with exact time intervals
- Let's see how can we do this using Timer0



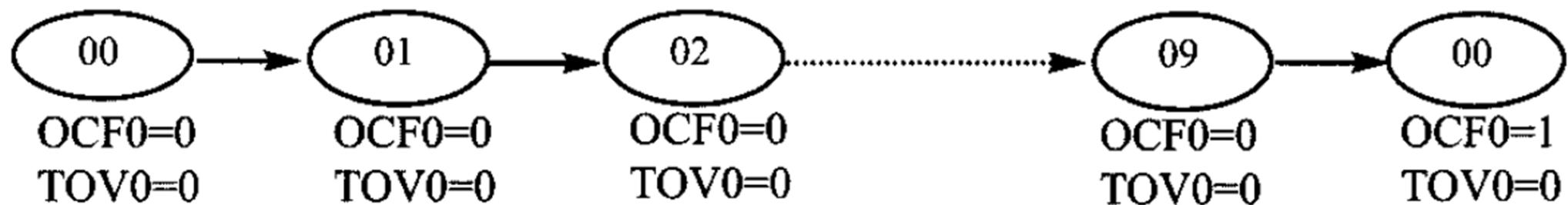
SOLUTION OVERVIEW

- Assume
 - system clock frequency = 16 MHz
 - Timer0 prescaler = 8
- To ensure 20 kHz sampling frequency
 - Time between successive ADC conversions?
 - 50us
 - Use Timer0 to trigger ADC every 50us
 - How to trigger Timer0 compare match every 50us?
 - Need to count?
 - $50\text{us}/(8/16000000) = 100$
 - From 0 to 99
 - Set OCR0 = 99



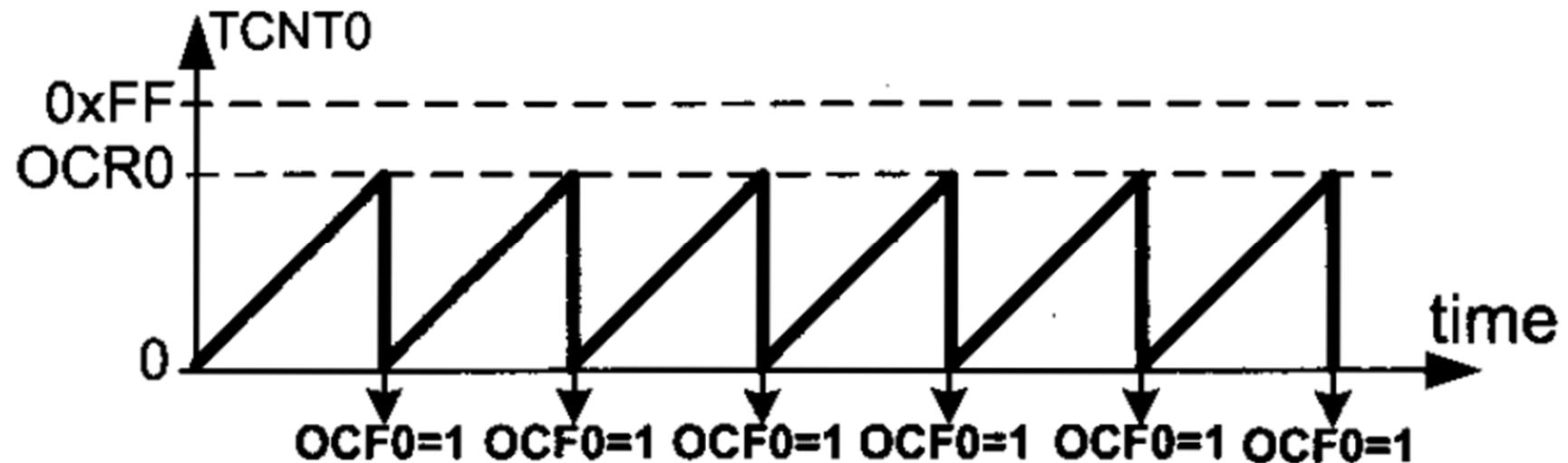
SOLUTION OVERVIEW

- Timer0 compare match trigger every 50us
 - Need to count?
 - $50\text{us}/(8/16000000) = 100$
 - From 0 to 99
 - Set OCR0 = 99
 - Why 99 instead of 100 ?



SOLUTION OVERVIEW

- Now let's select ADC prescaler **properly**
 - ADC conversion must be completed before next Timer0 compare match occur
- One auto triggered ADC conversion time takes 13.5 ADC cycles



SOLUTION OVERVIEW

- Now let's select ADC prescaler **properly**
 - ADC conversion must be completed before next Timer0 compare match occur
- One auto triggered ADC conversion time takes 13.5 ADC cycles

Prescaller	ADC clock, kHz	ADC clock period, us	Total conversion time, us
4	4000	0.25	3.375
8	2000	0.5	6.75
16	1000	1	13.5
32	500	2	27
64	250	4	54
128	125	8	108



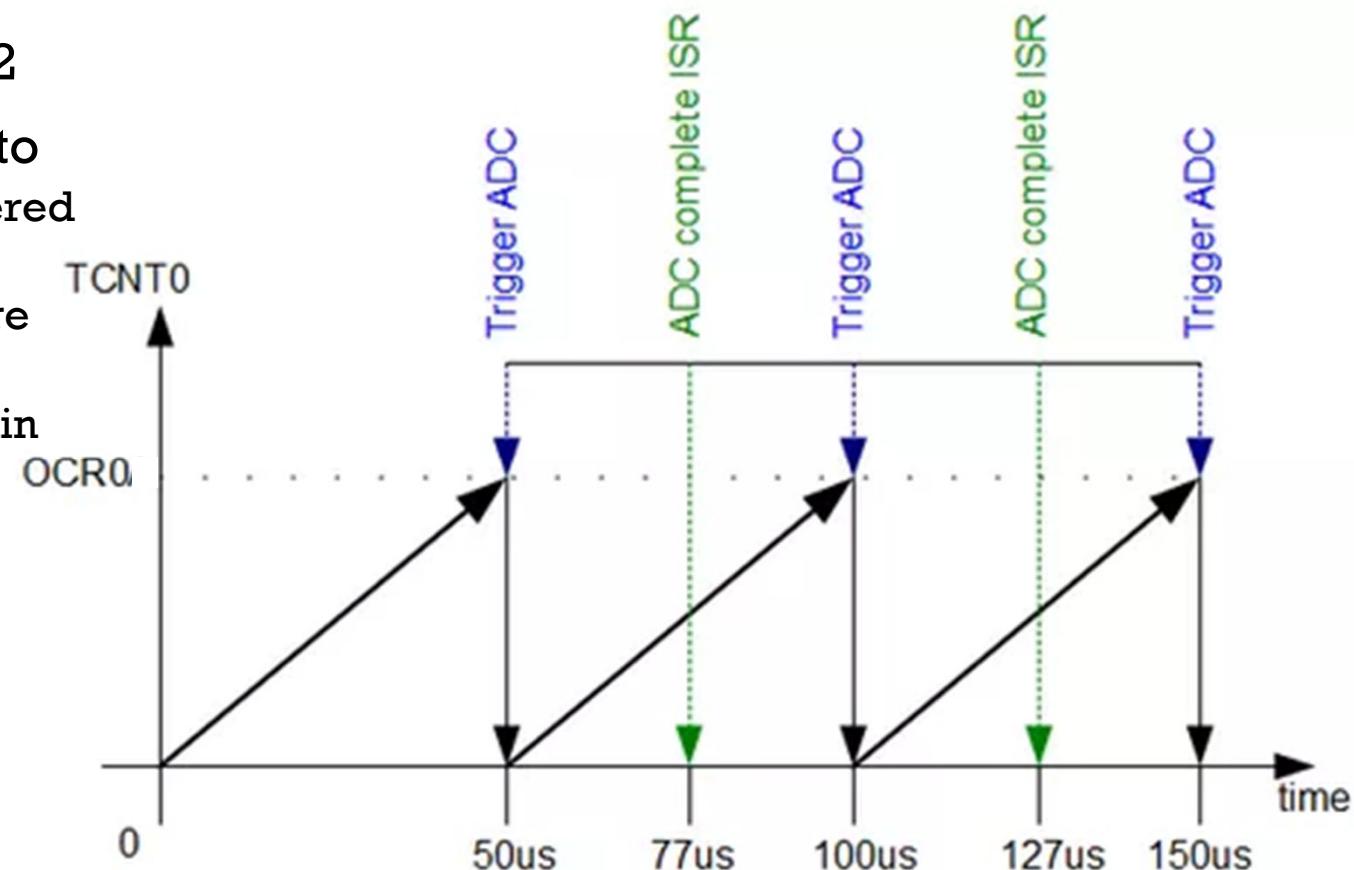
SOLUTION OVERVIEW

- To fit in to 50us window we can select prescaller up to 32
- Gives enough time to
 - Complete auto triggered ADC cycle
 - Clear Timer0 compare match flag
 - Store ADC value within one sample period
 - ...

Prescaller	ADC clock, kHz	ADC clock period, us	Total conversion time, us
4	4000	0.25	3.375
8	2000	0.5	6.75
16	1000	1	13.5
32	500	2	27
64	250	4	54
128	125	8	108

SOLUTION OVERVIEW

- ADC prescaler = 32
- Gives enough time to
 - Complete auto triggered ADC cycle
 - Clear Timer0 compare match flag
 - Store ADC value within one sample period
 - ...



COMPLETE CODE FOR ATMEGA328

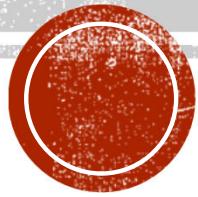
- <http://www.embedds.com/adc-on-atmega328-part-2/>



TIMER IN ATMEGA32

CSE 315

Courtesy: Muhammad Ali Nayeem, Azad Abdus Salam, Md.
Iftekharul Islam Sakib



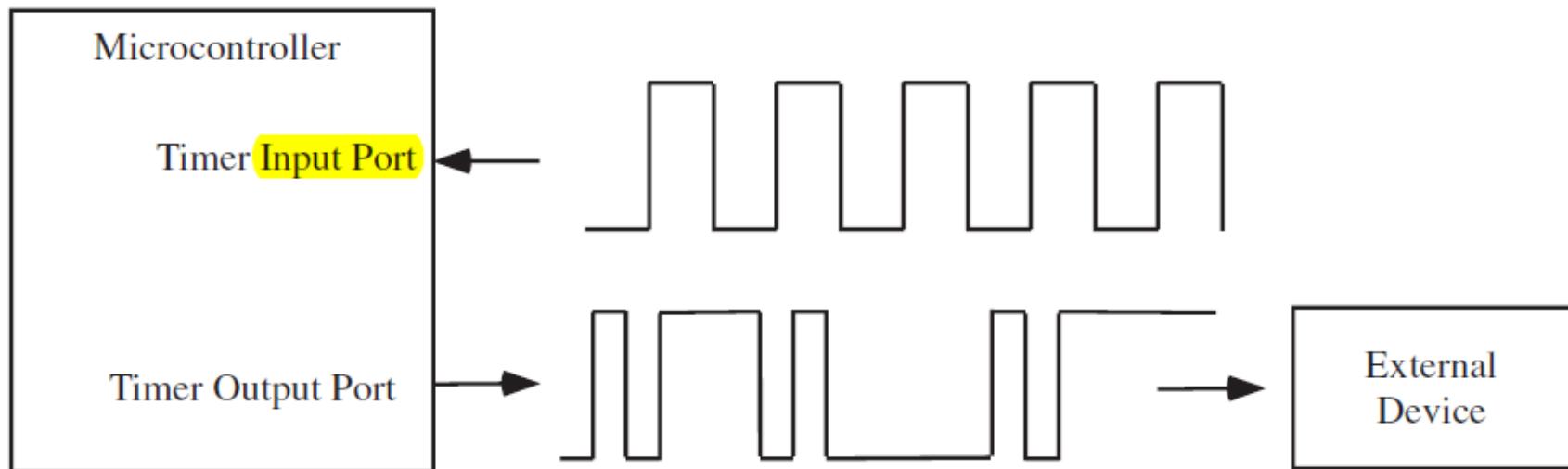
LEARNING OBJECTIVE

- What is timer?
- When, how to use timer?



TIME IS “MONEY”

- An embedded system has to perform various **time-related tasks**
 - Recording the time when an event occurs
 - Calculating the time difference between two events
 - Performing tasks at specific or periodic time instants
 - Creating accurate time delays
 - Generating waveforms of certain shape, period or duty cycle



SOFTWARE TECHNIQUE?

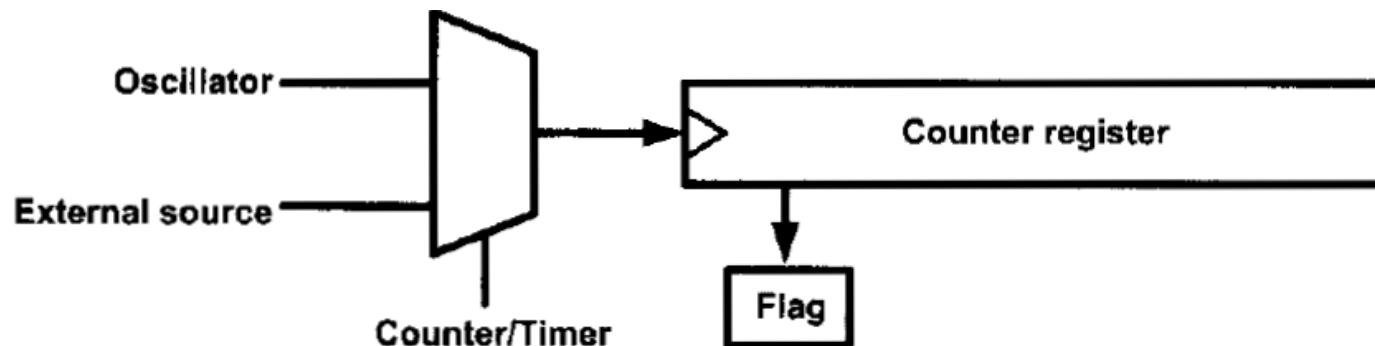
- Can we use CPU to do these tasks?

- Yes!
- But...
 - CPU will be busy
 - Timing will not be accurate

- Need something special?

- Hardware technique

$$1\mu S \times 2^{16} = x$$



PROBLEM DELAY_MS

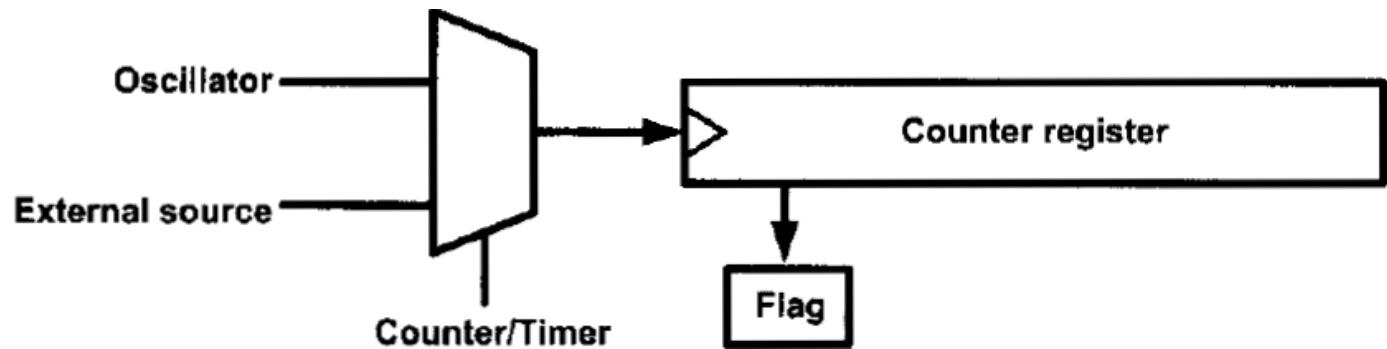
- If an ISR is fired during the execution of a `delay_ms()`,
 - ISR will be serviced first, then delay will continue
 - Resulting in more delay than specified
-
- For example,
 - If you call `_delay_ms(1)` and some ISR occurs that take 200us to be serviced, `_delay_ms(1)` takes 1.2ms



OVERVIEW

- ATmega32 has 3 timers
 - Timer 0, 1, 2
 - Timer 1 has the most functionalities
- Each timer is associated with

- A counter
- A clock signal
 - Internal
 - External



OVERVIEW

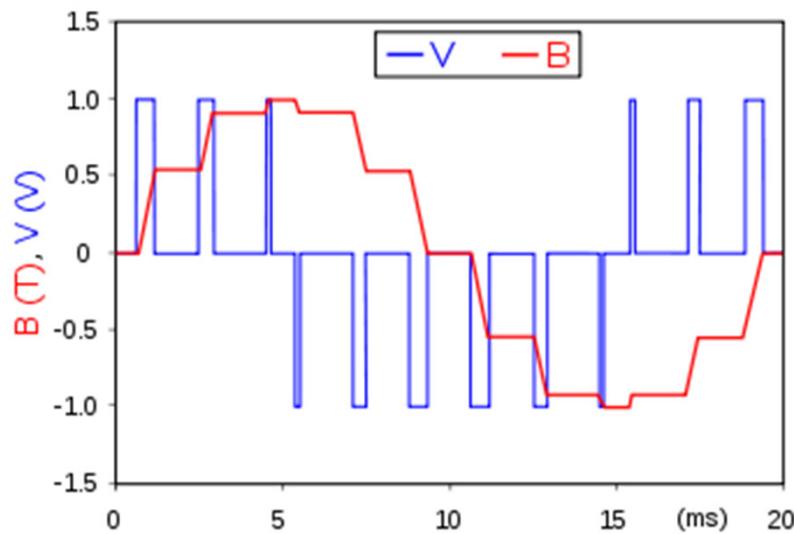
- When the **internal** system clock is used, a **prescaler** can be used to make the timer count at a slower rate
- Example:
 - Suppose the system clock rate = 1Mhz (1 μ s per cycle).
 - Suppose a timer prescaler of 64 is used.
 - Then, timer will increment every 64 μ s.



	Timer 0	Timer 1	Timer 2
Overall	<ul style="list-style-type: none"> - 8-bit counter - 10-bit prescaler 	<ul style="list-style-type: none"> - 16-bit counter - 10-bit prescaler 	<ul style="list-style-type: none"> - 8-bit counter - 10-bit prescaler
Functions	<ul style="list-style-type: none"> - PWM - Frequency generation - Event counter - Output compare 	<ul style="list-style-type: none"> - PWM - Frequency generation - Event counter - Output compare: 2 channels - Input capture 	<ul style="list-style-type: none"> - PWM - Frequency generation - Event counter - Output compare
Operation modes	<ul style="list-style-type: none"> - Normal mode - Clear timer on compare match - Fast PWM - Phase correct PWM 	<ul style="list-style-type: none"> - Normal mode - Clear timer on compare match - Fast PWM - Phase correct PWM 	<ul style="list-style-type: none"> - Normal mode - Clear timer on compare match - Fast PWM - Phase correct PWM

PWM

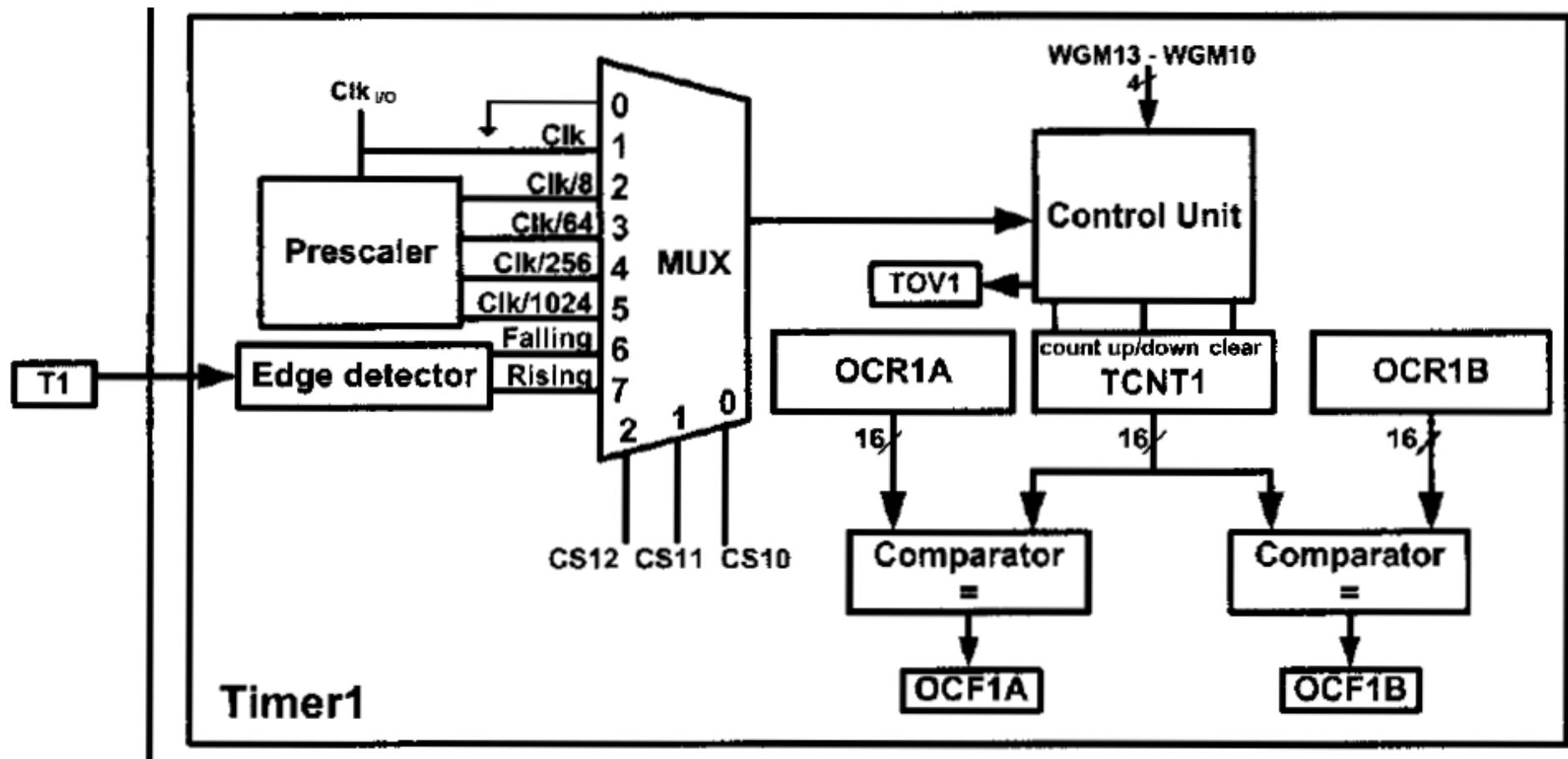
- Pulse Width Modulation
- A technique for getting analog results with digital means.



An example of PWM in an idealized inductor driven by a ■ voltage source modulated as a series of pulses, resulting in a ■ sine-like current in the inductor.



TIMER 1



TIMER 1

- 16 bit counter
 - prescaler: 8, 64, 256, 1024
- Can trigger a **timer overflow interrupt**
 - and other things.. We will learn those in the subsequent lectures
- Can trigger an **input capture interrupt** when an event occurs on the input capture pin
 - timer value is stored automatically in a register.
 - input capture pin for Timer 1 is ICP1 (D.6)

TIMER 1 – RELEVANT PINS

(XCK/T0) PB0	1	40	PA0 (ADC0)
(T1) PB1	2	39	PA1 (ADC1)
(INT2/AIN0) PB2	3	38	PA2 (ADC2)
(OC0/AIN1) PB3	4	37	PA3 (ADC3)
(SS) PB4	5	36	PA4 (ADC4)
(MOSI) PB5	6	35	PA5 (ADC5)
(MISO) PB6	7	34	PA6 (ADC6)
(SCK) PB7	8	33	PA7 (ADC7)
RESET	9	32	AREF
VCC	10	31	GND
GND	11	30	AVCC
XTAL2	12	29	PC7 (TOSC2)
XTAL1	13	28	PC6 (TOSC1)
(RXD) PD0	14	27	PC5 (TDI)
(TXD) PD1	15	26	PC4 (TDO)
(INT0) PD2	16	25	PC3 (TMS)
(INT1) PD3	17	24	PC2 (TCK)
(OC1B) PD4	18	23	PC1 (SDA)
(OC1A) PD5	19	22	PC0 (SCL)
(ICP1) PD6	20	21	PD7 (OC2)

used in this
lecture



RELATED REGISTERS – TIMER1

1. TIMER/COUNTER1

- TCNT1
- 16 bit register that stores the **current value** of the counter

2. Timer/Counter 1 Control Registers

- TCCR1A and TCCR1B
- To **configure** the operations of Timer 1.

3. Interrupt registers

- TIMSK to **enable** timer interrupts
- TIFR to monitor **status** of timer interrupts

RELATED REGISTERS – TIMER1

4. Input Capture Register

- ICR1

- to store timer value when an event occurs on input capture pin

5. Output Compare Registers

TIMER RELATED INTERRUPTS

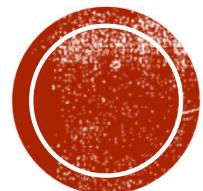
4	TIMER2_COMP_vect	Timer/Counter2 Compare Match
5	TIMER2_OVF_vect	Timer/Counter2 Overflow
6	TIMER1_CAPT_vect	Timer/Counter1 Capture Event
7	TIMER1_COMPA_vect	Timer/Counter1 Compare Match A
8	TIMER1_COMPB_vect	Timer/Counter1 Compare Match B
9	TIMER1_OVF_vect	Timer/Counter1 Overflow
10	TIMER0_OVF_vect	Timer/Counter0 Overflow



TODAY'S MISSION

- Creating an accurate delay using timer overflow interrupt
- Measuring elapsed time between two events





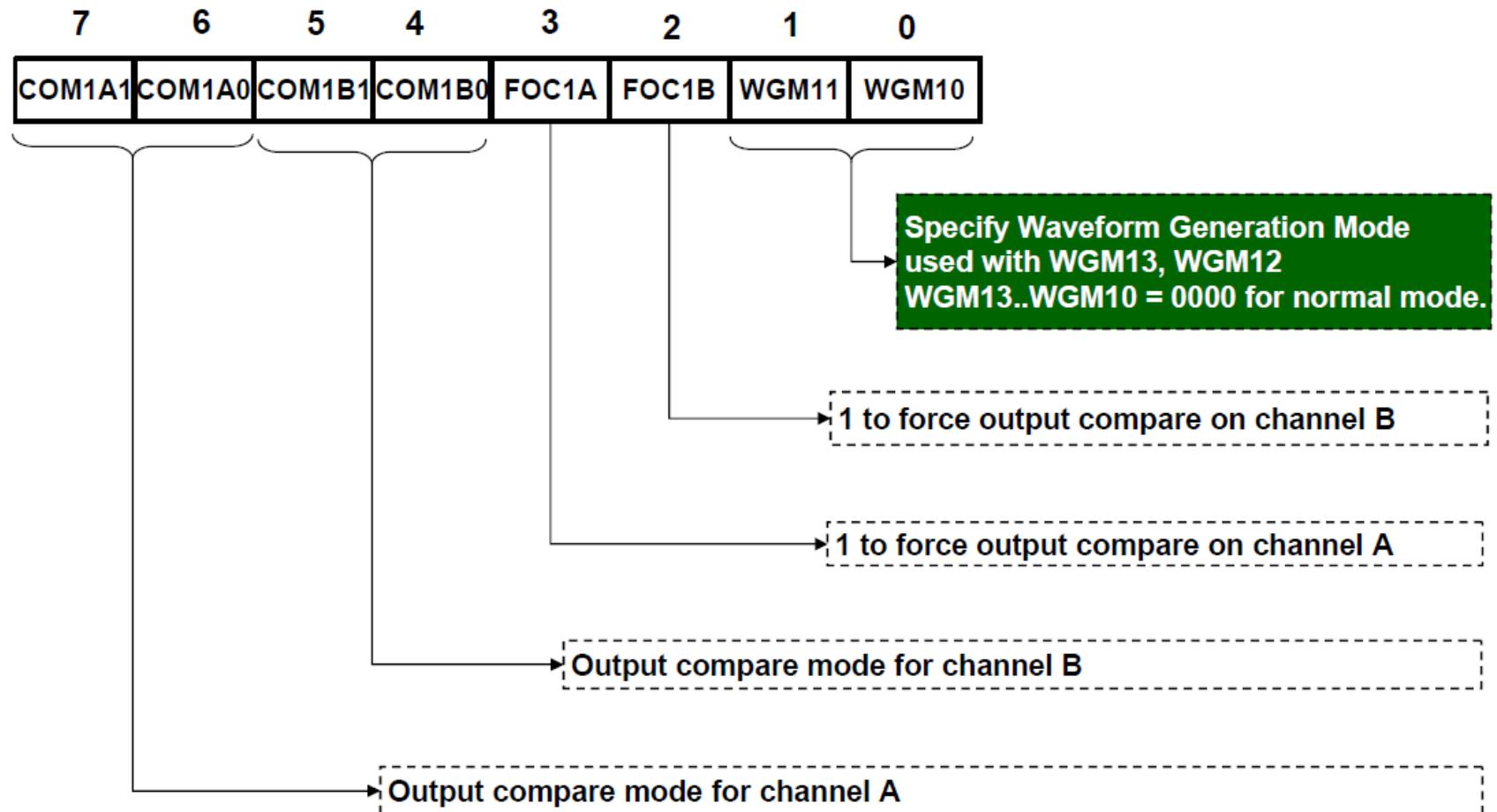
LET'S START!!!

MODES

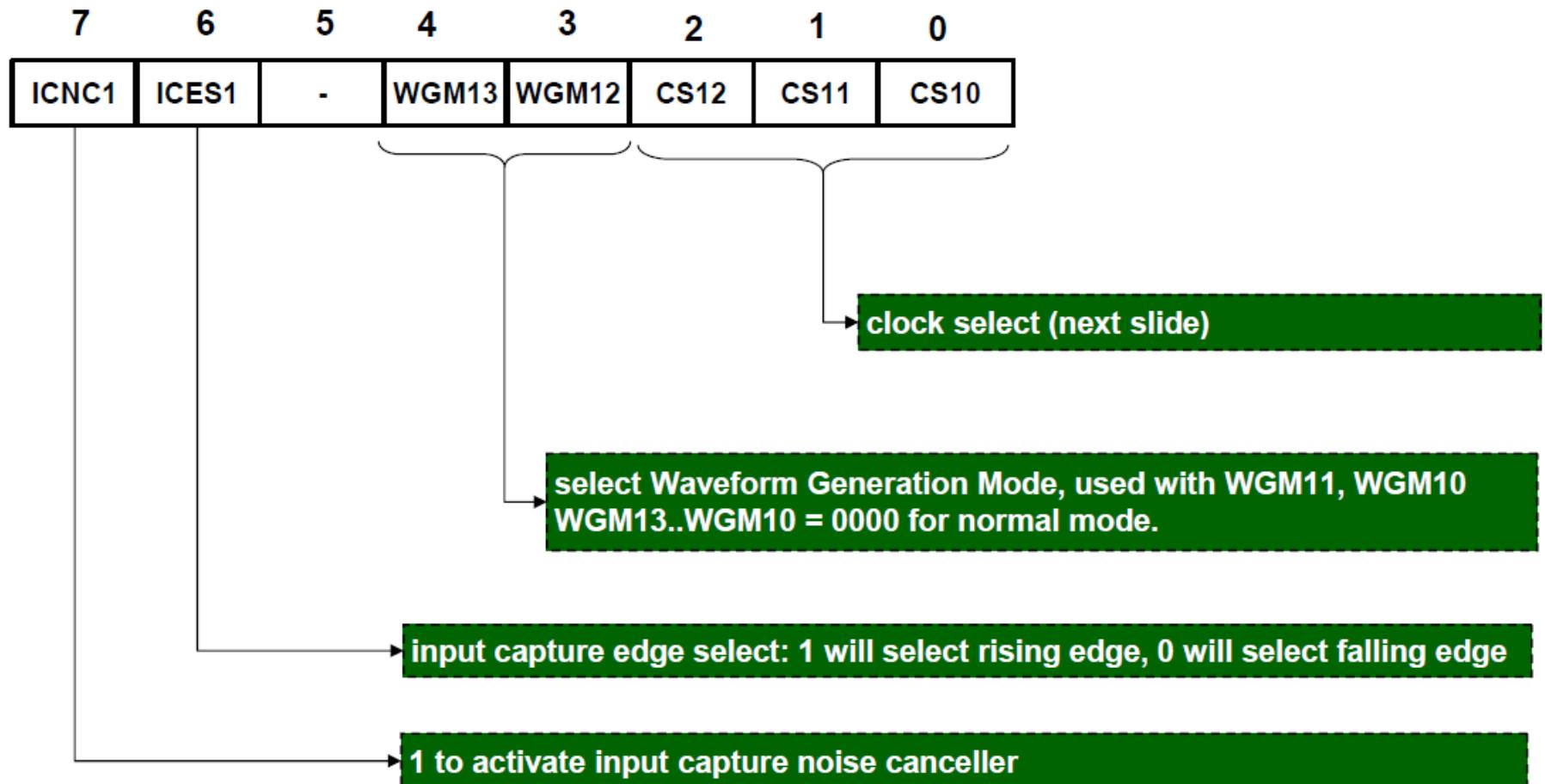
Mode	WGM13	WGM12 (CTC1)	WGM11 (PWM11)	WGM10 (PWM10)	Timer/Counter Mode of Operation	TOP	Update of OCR1x	TOV1 Flag Set on
0	0	0	0	0	Normal	0xFFFF	Immediate	MAX
1	0	0	0	1	PWM, Phase Correct, 8-bit	0x00FF	TOP	BOTTOM
2	0	0	1	0	PWM, Phase Correct, 9-bit	0x01FF	TOP	BOTTOM
3	0	0	1	1	PWM, Phase Correct, 10-bit	0x03FF	TOP	BOTTOM
4	0	1	0	0	CTC	OCR1A	Immediate	MAX
5	0	1	0	1	Fast PWM, 8-bit	0x00FF	TOP	TOP
6	0	1	1	0	Fast PWM, 9-bit	0x01FF	TOP	TOP
7	0	1	1	1	Fast PWM, 10-bit	0x03FF	TOP	TOP
8	1	0	0	0	PWM, Phase and Frequency Correct	ICR1	BOTTOM	BOTTOM
9	1	0	0	1	PWM, Phase and Frequency Correct	OCR1A	BOTTOM	BOTTOM
10	1	0	1	0	PWM, Phase Correct	ICR1	TOP	BOTTOM
11	1	0	1	1	PWM, Phase Correct	OCR1A	TOP	BOTTOM
12	1	1	0	0	CTC	ICR1	Immediate	MAX
13	1	1	0	1	Reserved	-	-	-
14	1	1	1	0	Fast PWM	ICR1	TOP	TOP
15	1	1	1	1	Fast PWM	OCR1A	TOP	TOP



TIMER/COUNTER 1 CONTROL REGISTER A (TCCR1A)



TIMER/COUNTER 1 CONTROL REGISTER B (TCCR1B)



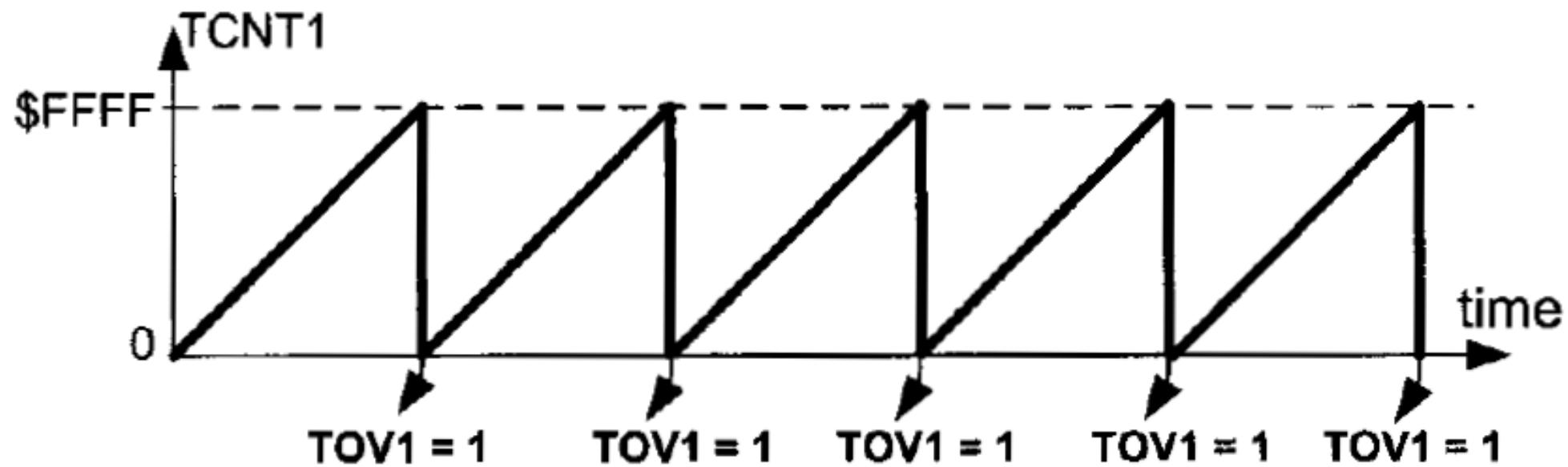
CLOCK SELECT

CS12	CS11	CS10	Description
0	0	0	No clock source (Timer/Counter stopped).
0	0	1	$\text{clk}_{\text{IO}}/1$ (No prescaling)
0	1	0	$\text{clk}_{\text{IO}}/8$ (From prescaler)
0	1	1	$\text{clk}_{\text{IO}}/64$ (From prescaler)
1	0	0	$\text{clk}_{\text{IO}}/256$ (From prescaler)
1	0	1	$\text{clk}_{\text{IO}}/1024$ (From prescaler)
1	1	0	External clock source on T1 pin. Clock on falling edge.
1	1	1	External clock source on T1 pin. Clock on rising edge.

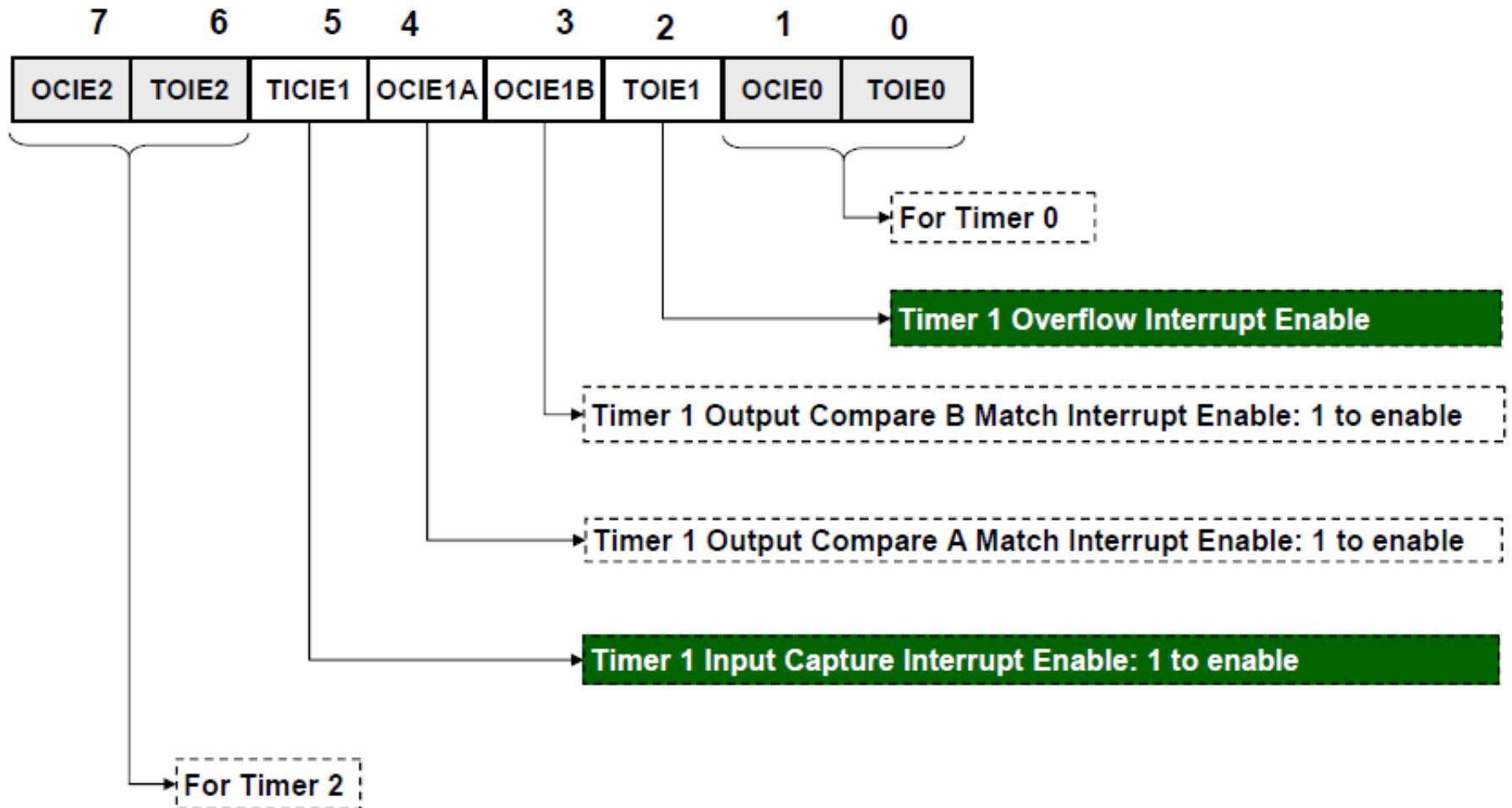
- For ATmega16, the internal clock is set by default at $\text{clk}_{\text{IO}} = 1\text{MHz}$.
- Timer 1 can run using the internal or external clock.
- If using the internal clock, we can set Timer 1 to run at a speed that is 8, 64, 256 or 1024 times slower than the internal clock.



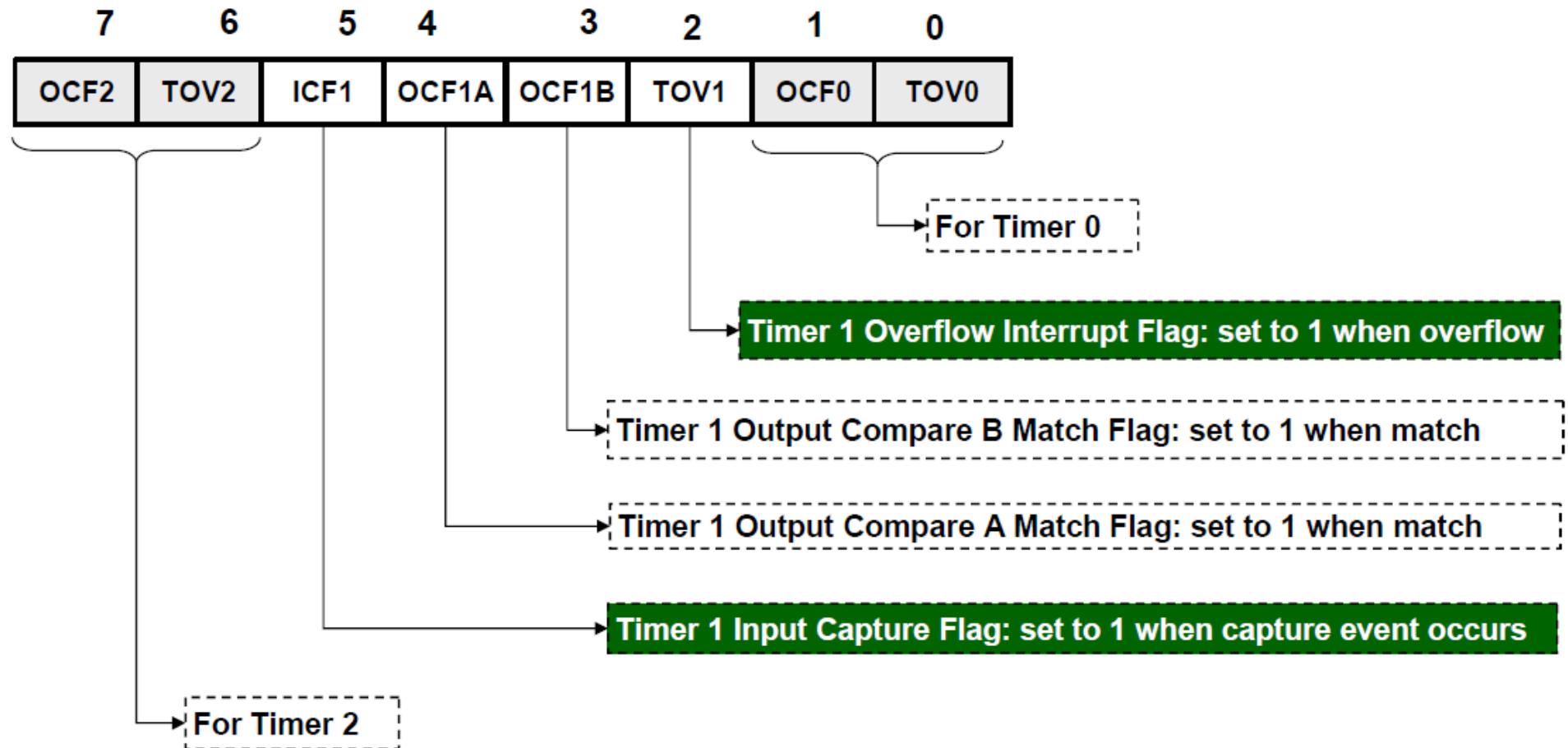
NORMAL MODE

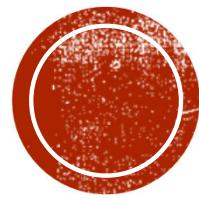


TIMER/COUNTER INTERRUPT MASK REGISTER (TIMSK)



TIMER/COUNTER INTERRUPT FLAG REGISTER (TIFR)





**WRITE A C CODE TO TOGGLE PORTB
EVERY 2 SECONDS
USING TIMER 1 OVERFLOW
INTERRUPT**

CREATING AN ACCURATE DELAY OF 2S

- System Default Frequency: 1Mhz
- Period: $1\mu\text{s}$
- With no prescaler TIMER 1 will increment every $1\mu\text{s}$
- TCNT1 is 16 bit
- Timer will overflow every $2^{16} \mu\text{s}$.
- How many times TIMER1 will overflow ??
- $2^{16}\mu\text{s} = 2^{16} \times 10^{-6}\text{s}$
- In 2S, total TIMER1 overflow = $\frac{2}{2^{16} \times 10^{-6}} = 30.51 \approx 31$

Bit	7	6	5	4	3	2	1	0	
Read/Write	R/W	R/W	R/W	R/W	W	W	R/W	R/W	TCCR1A
Initial Value	0	0	0	0	0	0	0	0	
Bit	7	6	5	4	3	2	1	0	
Read/Write	R/W	R/W	R	R/W	R/W	R/W	R/W	R/W	TCCR1B
Initial Value	0	0	0	0	0	0	0	0	

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

```
#include <avr/io.h>
#include <avr/interrupt.h>

volatile int overflowCount;
ISR(TIMER1_OVF_vect)
{
    overflowCount++;
    if(overflowCount==31)
    {
        PORTB = ~PORTB;
        overflowCount = 0;
    }
}
```

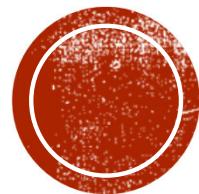
```
int main(void)
{
    overflowCount = 0;
    DDRB = 0xFF;
    PORTB = 0xFF;
    //configure timer
    TCCR1A = 0b00000000; // normal mode
    TCCR1B = 0b00000001; // no prescaler, internal
    clock

    TIMSK = 0b00000100; //Enable Overflow Interrupt
    sei(); //Global Interrupt Enable

    while(1);
}
```

APPENDIX

- Why volatile variable?
 - <http://www.avrfreaks.net/forum/whats-volatile-variable>

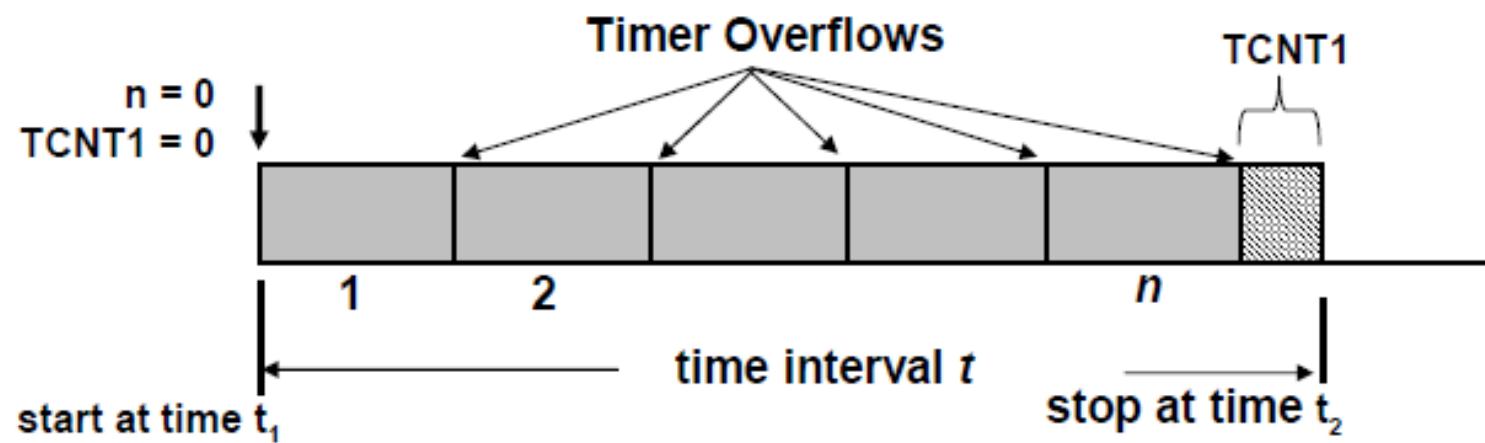


**MEASURE THE TIME FOR A
DEMO C CODE TO EXECUTE**



MEASURING ELAPSED TIME

$$t = n \times 65536 + \text{TCNT1} \quad (\mu\text{s}) \quad \text{When no prescaling}$$



MEASURING ELAPSED TIME

- 16 Bits
 - 2^{16} Counts to overflow
 - Find times between increments (Period)
 - For no prescaler, frequency, $F = 10^6 / 1 = 10^6$
 - Period, $P = 1 / F = 10^{-6} = 1\mu s$
-
- Number of overflows = n
 - Total count, $C = 2^{16} \times n + \text{TCNT1}$
 - Total time = $C \times P$



Bit	7	6	5	4	3	2	1	0	
	COM1A1	COM1A0	COM1B1	COM1B0	FOC1A	FOC1B	WGM11	WGM10	TCCR1A
Read/Write	R/W	R/W	R/W	R/W	W	W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Bit	7	6	5	4	3	2	1	0	
	ICNC1	ICES1	-	WGM13	WGM12	CS12	CS11	CS10	TCCR1B
Read/Write	R/W	R/W	R	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Bit	7	6	5	4	3	2	1	0	
	OCIE2	TOIE2	TICIE1	OCIE1A	OCIE1B	TOIE1	OCIE0	TOIE0	TIMSK
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	

```

#include <avr/io.h>
#include <avr/interrupt.h>
#include <inttypes.h>

volatile uint32_t n;

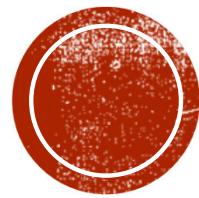
ISR(TIMER1_OVF_vect){ // handler for Timer1 overflow interrupt
    n++; // increment overflow count
}

int main(void) {
    int i, j;
    uint32_t elapse_time;

    TCCR1A = 0b00000000; // normal mode
    TCCR1B = 0b00000001; // no prescaler, internal clock
    TIMSK  = 0b00000100; // enable Timer 1 overflow interrupt

    n = 0; // reset n
    TCNT1 = 0; // reset Timer 1
    sei(); // enable interrupt subsystem globally
    // ----- start code -----
    for (i = 0; i < 100; i++)
        for (j = 0; j < 1000; j++){}
    // ----- end code -----
    elapse_time = n * 65536 + (uint32_t) TCNT1; // In micro seconds
    cli(); // disable interrupt subsystem globally
    return 0;
}

```



**MEASURE THE PERIOD OF
A SQUARE WAVE**

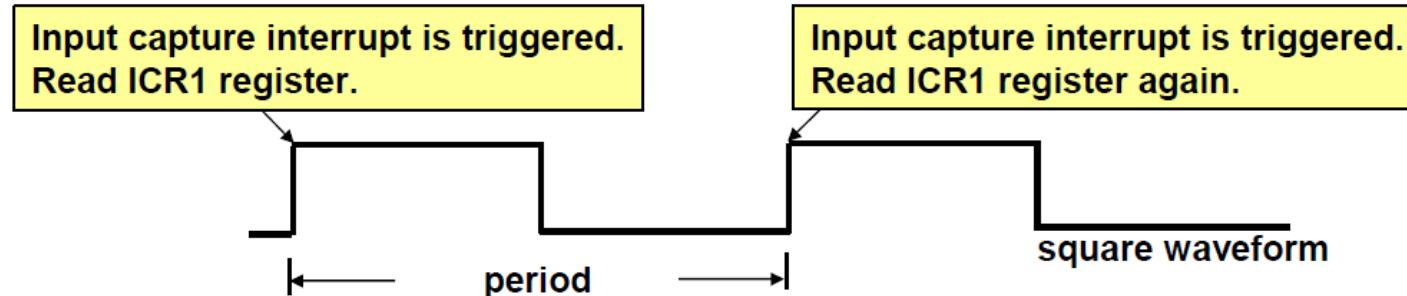


MEASURE PERIOD

■ Analysis:

- The period of a square wave = the time difference between two consecutive rising edges.
- Connect the square wave to input capture pin of Timer 1. →
- Configure input capture module to trigger on a rising edge.

PDIP	
(XCK/T0)	PB0
(T1)	PB1
(INT2/AIN0)	PB2
(OC0/AIN1)	PB3
(SS)	PB4
(MOSI)	PB5
(MISO)	PB6
(SCK)	PB7
RESET	
VCC	10
GND	11
XTAL2	12
XTAL1	13
(RXD)	PD0
(TXD)	PD1
(INT0)	PD2
(INT1)	PD3
(OC1B)	PD4
(OC1A)	PD5
(ICP1)	PD6
1	40
2	39
3	38
4	37
5	36
6	35
7	34
8	33
9	32
10	31
11	30
12	29
13	28
14	27
15	26
16	25
17	24
18	23
19	22
20	21
PA0 (ADC0)	
PA1 (ADC1)	
PA2 (ADC2)	
PA3 (ADC3)	
PA4 (ADC4)	
PA5 (ADC5)	
PA6 (ADC6)	
PA7 (ADC7)	
AREF	
GND	
AVCC	
PC7 (TOSC2)	
PC6 (TOSC1)	
PC5 (TDI)	
PC4 (TDO)	
PC3 (TMS)	
PC2 (TCK)	
PC1 (SDA)	
PC0 (SCL)	
PD7 (OC2)	



When a capture is triggered, the 16-bit value of the counter (TCNT1) is written to the *Input Capture Register (ICR1)*.



MEASURE PERIOD

- Assumption: the input signal has a high frequency, hence timer overflow can be ignored.

Say, period of input signal is less than $2^{16}\mu\text{s}$.

- ## ■ Implementation:

- ☐ **Select timer operations:** normal, no prescaler, internal clock 1MHz, noise canceller enabled, input capture for rising edges.

```
TCCR1A = 0b00000000;
```

TCCR1B = 0b11000001;

- #### Enable input capture interrupt:

```
TIMSK = 0b00100000;
```

```
#include <avr/io.h>
#include <avr/interrupt.h>
#include <inttypes.h>

volatile uint16_t period;

ISR(TIMER1_CAPT_vect){ // handler for Timer1 input capture interrupt
    period = ICR1;          // period = value of Timer 1 stored in ICR1
    TCNT1 = 0;              // reset Timer 1
    PORTB = ~(period >> 8); // display top 8-bit of period on PORT B
}

int main(void) {
    DDRB = 0xFF;           // set port B for output

    TCCR1A = 0b00000000;   // normal mode
    TCCR1B = 0b11000001;   // no prescaler, rising edge, noise canceller
    TIMSK = 0b00100000;    // enable Timer 1 input capture interrupt
    sei();                 // enable interrupt subsystem globally
    while (1){;}          // infinite loop
    return 0;
}
```



MEASURE THE PERIOD OF A SQUARE WAVE

- Exercise:
- No assumption on frequency
- Period of input signal is greater than $2^{16}\mu s$.

- Need to consider overflow count.
- Left as an exercise to the reader.



Microprocessors and Microcontrollers

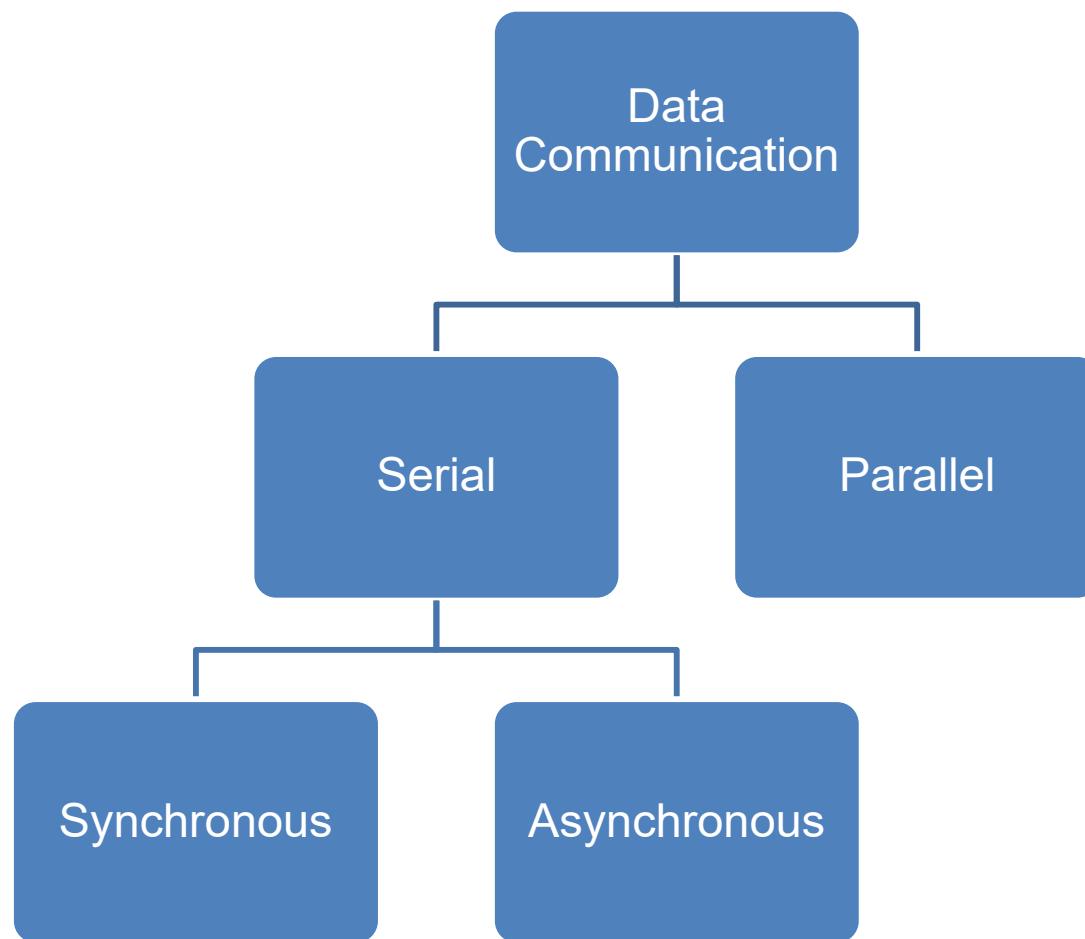
SERIAL COMMUNICATION USING ATMEGA16/32

Md. Tareq Mahmood

Original slides made by Azad Abdus Salam



Types of Data Communication



Serial vs. Parallel Communication

- Computers transfer data in two ways: parallel and serial.
- Parallel: Several data bits are transferred simultaneously, e.g. printers and hard disks.
- Serial: A single data bit is transferred at one time, e.g. bluetooth and USB.



Why Serial Communication ???

- longer distances (fewer wires/cables)
- easier to synchronize
- fewer IO pins
- lower cost



Synchronous Communication

- The clocks of the sender and receiver are synchronized.
- A block of characters, enclosed by synchronizing bytes, is sent at a time.
- Faster transfer and less overhead
- Example:
 - Serial Peripheral Interface (SPI) by Motorola

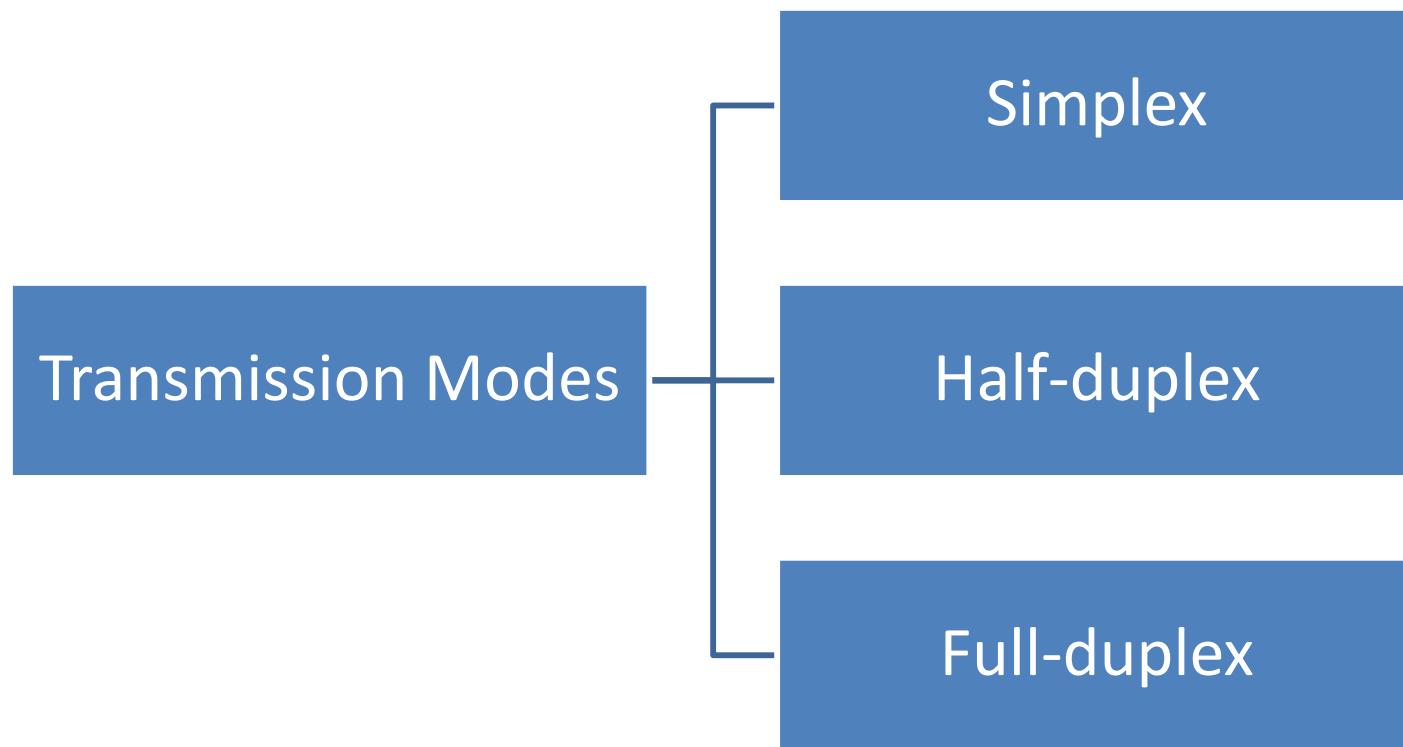


Asynchronous Communication

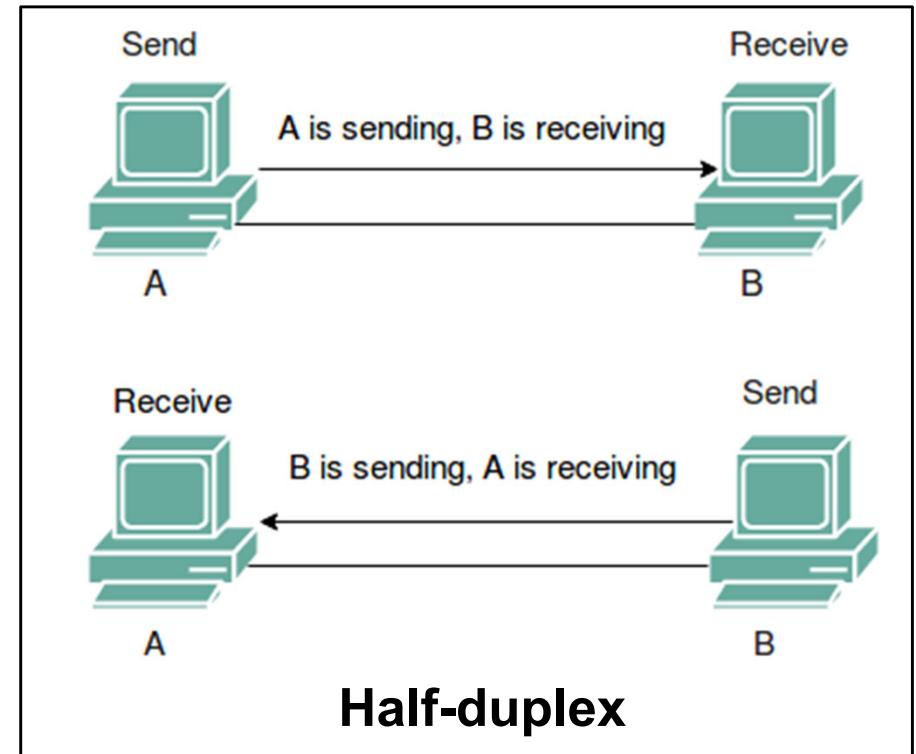
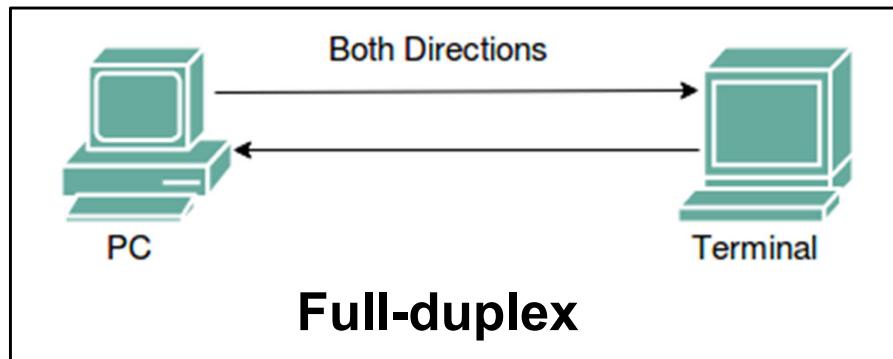
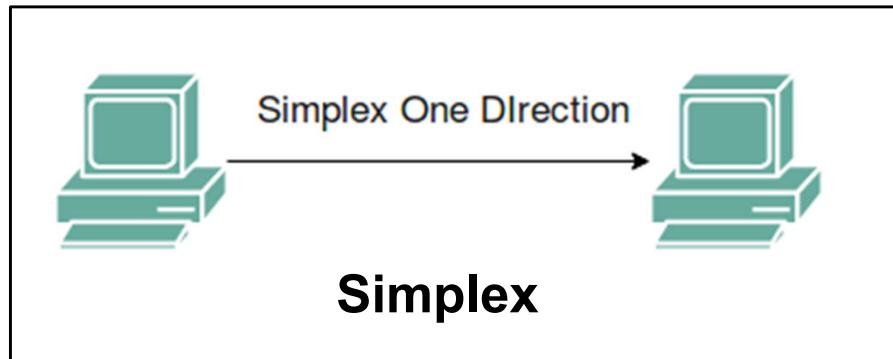
- The clocks of the sender and receiver are not synchronized.
- One character (5-9 bits) is sent at a time, enclosed between a start bit and one or two stop bits. A parity bit may be included.
- Examples:
 - RS232 (part of) by Electronic Industry Alliance.
 - USART of ATmega16



Transmission Modes/ Channel Types



Transmission Modes/ Channel Types



Serial communications in ATmega16

- ATmega16 provides three subsystems for serial communications
 - Universal Synchronous & Asynchronous Serial Receiver & Transmitter (USART)
 - Serial Peripheral Interface (SPI)
 - Two-wire Serial Interface (TWI)



USART

- Supports **full-duplex mode** between a receiver and transmitter
- Typically used in asynchronous communication.
 - We focus more on the **asynchronous communication!!** Often called UART
 - An asynchronous UART can operate using 2 wires:
Transmit (Tx) and Receive (Rx)
- Typical speed: 960bps up to 57.6kbps



USART

ATmega32 (PDIP-40) Pinout	
(XCK/T0) PB0	1
(T1) PB1	2
(INT2/AIN0) PB2	3
(OC0/AIN1) PB3	4
(SS) PB4	5
(MOSI) PB5	6
(MISO) PB6	7
(SCK) PB7	8
RESET	9
VCC	10
GND	11
XTAL2	12
XTAL1	13
(RXD) PD0	14
(TXD) PD1	15
(INT0) PD2	16
(INT1) PD3	17
(OC1B) PD4	18
(OC1A) PD5	19
(ICP1) PD6	20
	40
	39
	38
	37
	36
	35
	34
	33
	32
	31
	30
	29
	28
	27
	26
	25
	24
	23
	22
	21
PA0 (ADC0)	
PA1 (ADC1)	
PA2 (ADC2)	
PA3 (ADC3)	
PA4 (ADC4)	
PA5 (ADC5)	
PA6 (ADC6)	
PA7 (ADC7)	
AREF	
GND	
AVCC	
PC7 (TOSC2)	
PC6 (TOSC1)	
PC5 (TDI)	
PC4 (TDO)	
PC3 (TMS)	
PC2 (TCK)	
PC1 (SDA)	
PC0 (SCL)	
PD7 (OC2)	

Serial Peripheral Interface (SPI)

- The receiver and transmitter share a **common clock** line.
- The device considered the 'Master' provides a clock signal used to synchronize data transactions between the two devices.
- Supports **higher data rates**.
- A SPI interface uses a 3-wire system
 - MOSI, MISO, SCK



Serial Peripheral Interface (SPI)

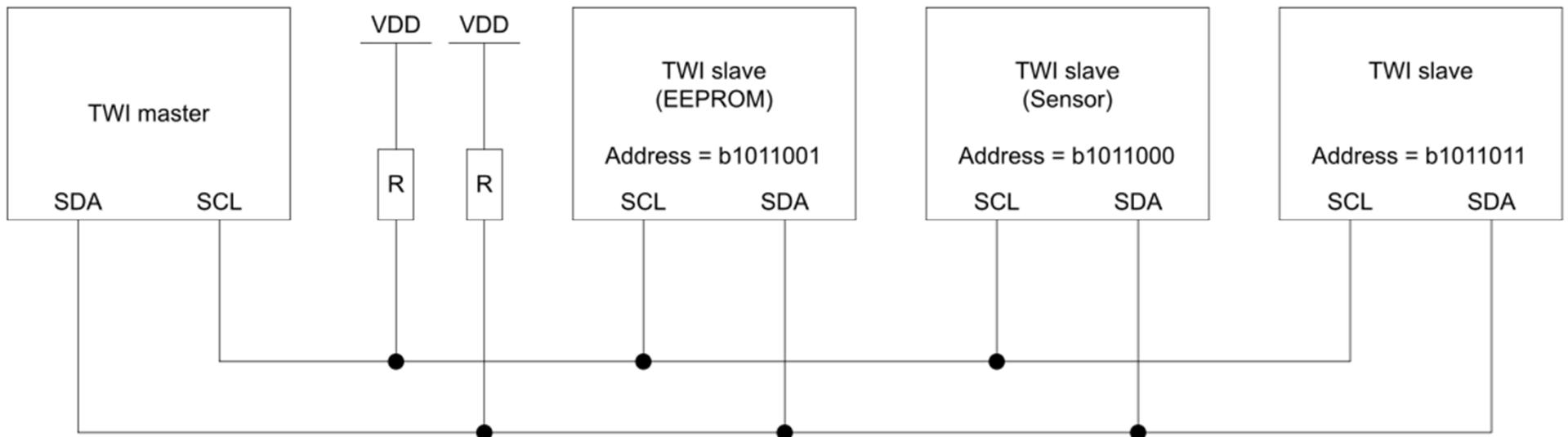
ATmega32 (PDIP-40) Pinout	
(XCK/T0) PB0	1
(T1) PB1	2
(INT2/AIN0) PB2	3
(OC0/AIN1) PB3	4
(SS) PB4	5
(MOSI) PB5	6
(MISO) PB6	7
(SCK) PB7	8
RESET	9
VCC	10
GND	11
XTAL2	12
XTAL1	13
(RXD) PD0	14
(TXD) PD1	15
(INT0) PD2	16
(INT1) PD3	17
(OC1B) PD4	18
(OC1A) PD5	19
(ICP1) PD6	20
40	PA0 (ADC0)
39	PA1 (ADC1)
38	PA2 (ADC2)
37	PA3 (ADC3)
36	PA4 (ADC4)
35	PA5 (ADC5)
34	PA6 (ADC6)
33	PA7 (ADC7)
32	AREF
31	GND
30	AVCC
29	PC7 (TOSC2)
28	PC6 (TOSC1)
27	PC5 (TDI)
26	PC4 (TDO)
25	PC3 (TMS)
24	PC2 (TCK)
23	PC1 (SDA)
22	PC0 (SCL)
21	PD7 (OC2)

Two-wire Serial Interface (TWI)

- Network several devices such as microcontrollers and display boards, using a two-wire bus.
- 7 bit address
- Up to 128 devices are supported
- Each **device has a unique address** and can exchange data with other devices in a small network.
- TWI is ATmega32 version of Phillips' I²C
 - Inter-Integrated Circuit, pronounced "I squared C"



Two-wire Serial Interface (TWI)



A sample TWI network connecting 3 devices with TWI master.



Two-wire Serial Interface (TWI)

ATmega32 (PDIP-40) Pinout	
(XCK/T0) PB0	1
(T1) PB1	2
(INT2/AIN0) PB2	3
(OC0/AIN1) PB3	4
(SS) PB4	5
(MOSI) PB5	6
(MISO) PB6	7
(SCK) PB7	8
RESET	9
VCC	10
GND	11
XTAL2	12
XTAL1	13
(RXD) PD0	14
(TXD) PD1	15
(INT0) PD2	16
(INT1) PD3	17
(OC1B) PD4	18
(OC1A) PD5	19
(ICP1) PD6	20
	40
	39
	38
	37
	36
	35
	34
	33
	32
	31
	30
	29
	28
	27
	26
	25
	24
	23
	22
	21
PA0 (ADC0)	
PA1 (ADC1)	
PA2 (ADC2)	
PA3 (ADC3)	
PA4 (ADC4)	
PA5 (ADC5)	
PA6 (ADC6)	
PA7 (ADC7)	
AREF	
GND	
AVCC	
PC7 (TOSC2)	
PC6 (TOSC1)	
PC5 (TDI)	
PC4 (TDO)	
PC3 (TMS)	
PC2 (TCK)	
PC1 (SDA)	
PC0 (SCL)	
PD7 (OC2)	

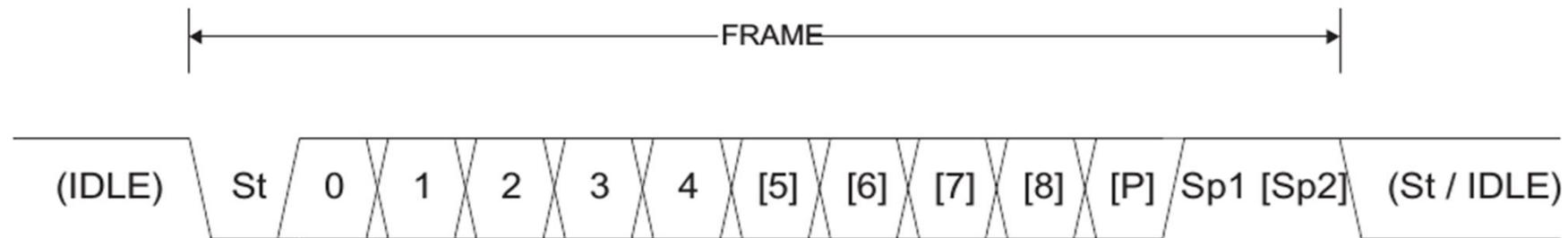
Serial communications terminology

- **Parity bit:** a single bit that is sent together with data bits to make the total number of 1's even (for even parity) or odd (for odd parity)
 - used for error checking.
- **Start bit:** to indicate the start of a character. Its typical value is 0.
- **Stop bit:** to indicate the end of a character. Its typical value is 1



Data Framing - Asynchronous

Figure 72. Frame Formats



St Start bit, always low.

(n) Data bits (0 to 8).

P Parity bit. Can be odd or even.

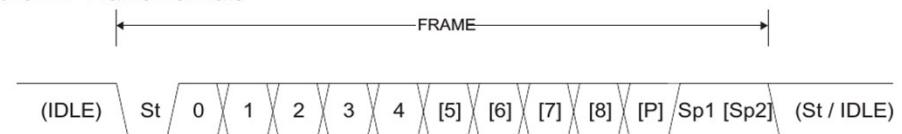
Sp Stop bit, always high.

IDLE No transfers on the communication line (RxD or TxD). An IDLE line must be high.

A frame starts with the start bit followed by the least significant data bit.

Data Framing - Asynchronous

Figure 72. Frame Formats



St Start bit, always low.

(n) Data bits (0 to 8).

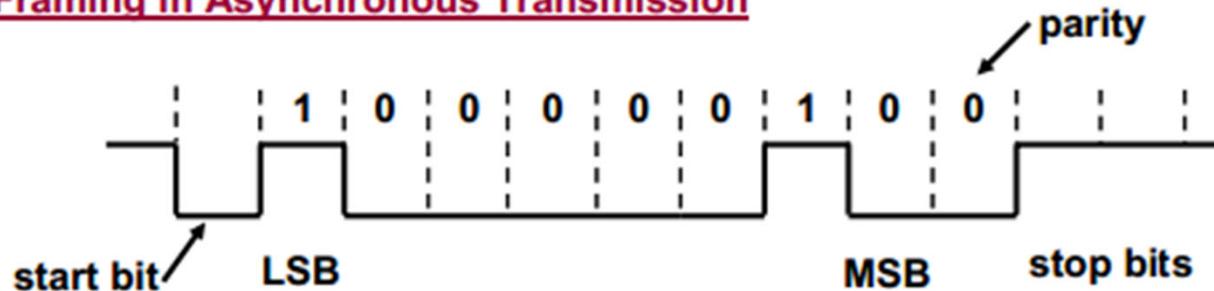
P Parity bit. Can be odd or even.

Sp Stop bit, always high.

IDLE No transfers on the communication line (RxD or TxD). An IDLE line must be high.

Example:

Data Framing in Asynchronous Transmission



Sending character "A" (41h = 0100 0001)
8-bit data, 1 start bit, 2 stop bits, even-parity



USART in ATmega16 – An Overview

- baud rates from 960bps up to 57.6kbps
 - technically baud rate refers to symbols per second
 - for our case baud rate is equal to bit-rate
- character size: 5 to 9 bits
- 1 start bit
- 1 or 2 stop bits
- parity bit (optional: even or odd parity)



USART – Hardware Elements

- USART Transmitter:
 - to send a character through TxD pin.
 - to handle start/stop bit framing, parity bit, shift register.
- USART Receiver:
 - to receive a character through RxD pin.
 - to perform the reverse operation of the transmitter



PDIP

(XCK/T0)	PB0	<input type="checkbox"/>	1		40	<input type="checkbox"/>	PA0 (ADC0)
(T1)	PB1	<input type="checkbox"/>	2		39	<input type="checkbox"/>	PA1 (ADC1)
(INT2/AIN0)	PB2	<input type="checkbox"/>	3		38	<input type="checkbox"/>	PA2 (ADC2)
(OC0/AIN1)	PB3	<input type="checkbox"/>	4		37	<input type="checkbox"/>	PA3 (ADC3)
(SS)	PB4	<input type="checkbox"/>	5		36	<input type="checkbox"/>	PA4 (ADC4)
(MOSI)	PB5	<input type="checkbox"/>	6		35	<input type="checkbox"/>	PA5 (ADC5)
(MISO)	PB6	<input type="checkbox"/>	7		34	<input type="checkbox"/>	PA6 (ADC6)
(SCK)	PB7	<input type="checkbox"/>	8		33	<input type="checkbox"/>	PA7 (ADC7)
<u>RESET</u>		<input type="checkbox"/>	9		32	<input type="checkbox"/>	AREF
VCC		<input type="checkbox"/>	10		31	<input type="checkbox"/>	GND
GND		<input type="checkbox"/>	11		30	<input type="checkbox"/>	AVCC
XTAL2		<input type="checkbox"/>	12		29	<input type="checkbox"/>	PC7 (TOSC2)
XTAL1		<input type="checkbox"/>	13		28	<input type="checkbox"/>	PC6 (TOSC1)
(RXD)	PD0	<input type="checkbox"/>	14		27	<input type="checkbox"/>	PC5 (TDI)
(TXD)	PD1	<input type="checkbox"/>	15		26	<input type="checkbox"/>	PC4 (TDO)
(INT0)	PD2	<input type="checkbox"/>	16		25	<input type="checkbox"/>	PC3 (TMS)
(INT1)	PD3	<input type="checkbox"/>	17		24	<input type="checkbox"/>	PC2 (TCK)
(OC1B)	PD4	<input type="checkbox"/>	18		23	<input type="checkbox"/>	PC1 (SDA)
(OC1A)	PD5	<input type="checkbox"/>	19		22	<input type="checkbox"/>	PC0 (SCL)
(ICP1)	PD6	<input type="checkbox"/>	20		21	<input type="checkbox"/>	PD7 (OC2)

PINOUT

- RXD
 - To receive
- TXD
 - To transmit



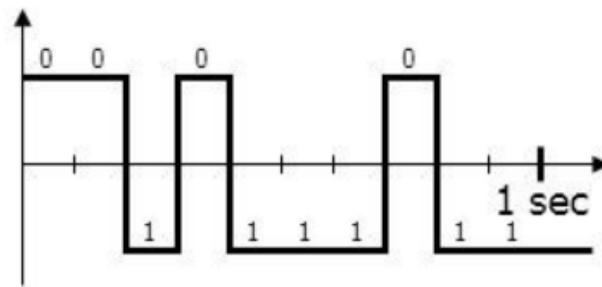
USART - Registers

- Three set of registers
- USART Baud Rate Registers
 - UBRRH and UBRRL
 - Sets the speed of communication
- USART Control and Status Registers
 - UCSRA, UCSRB, and UCSRC
- USART Data Register
 - UDR
 - Used to read and write actual data!

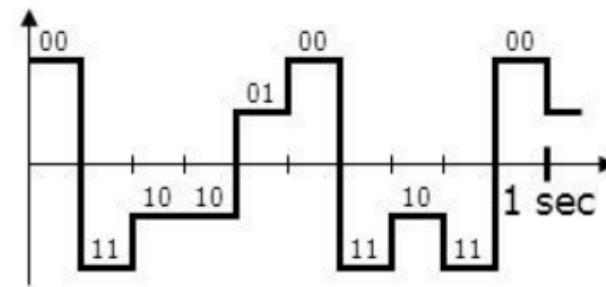


Baud rate

- The number of bits sent per second (**bps**).
- Strictly speaking, baud rate is the number of symbols per second.
- Or, how many times signal changes states
- As we have only two symbols 0 and 1, here baud rate is equal to **bps**



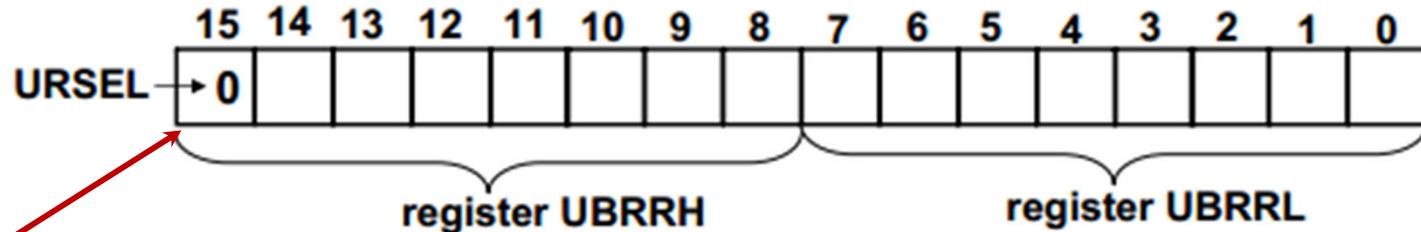
Baud = 10
Bit rate = 10 bps



Baud = 10
Bit rate = 20 bps



Setting Baud Rate



Must be 0 when
setting the baud
rate

$$\text{baud rate} = \frac{\text{system clock frequency (Hz)}}{16(\text{UBRR} + 1)}$$

$$\text{UBRR} = \frac{\text{system clock frequency (Hz)}}{16 \times \text{baud rate}} - 1$$

- Two register UBRRH and UBRLL are used together to set the baud rate
- There is also a **double speed transmission mode**, where the 16 is replaced by 8 in the formula

Setting Baud Rate – An Example

Find UBRR registers for baud rate of 1200bps, assuming system clock is 1MHz

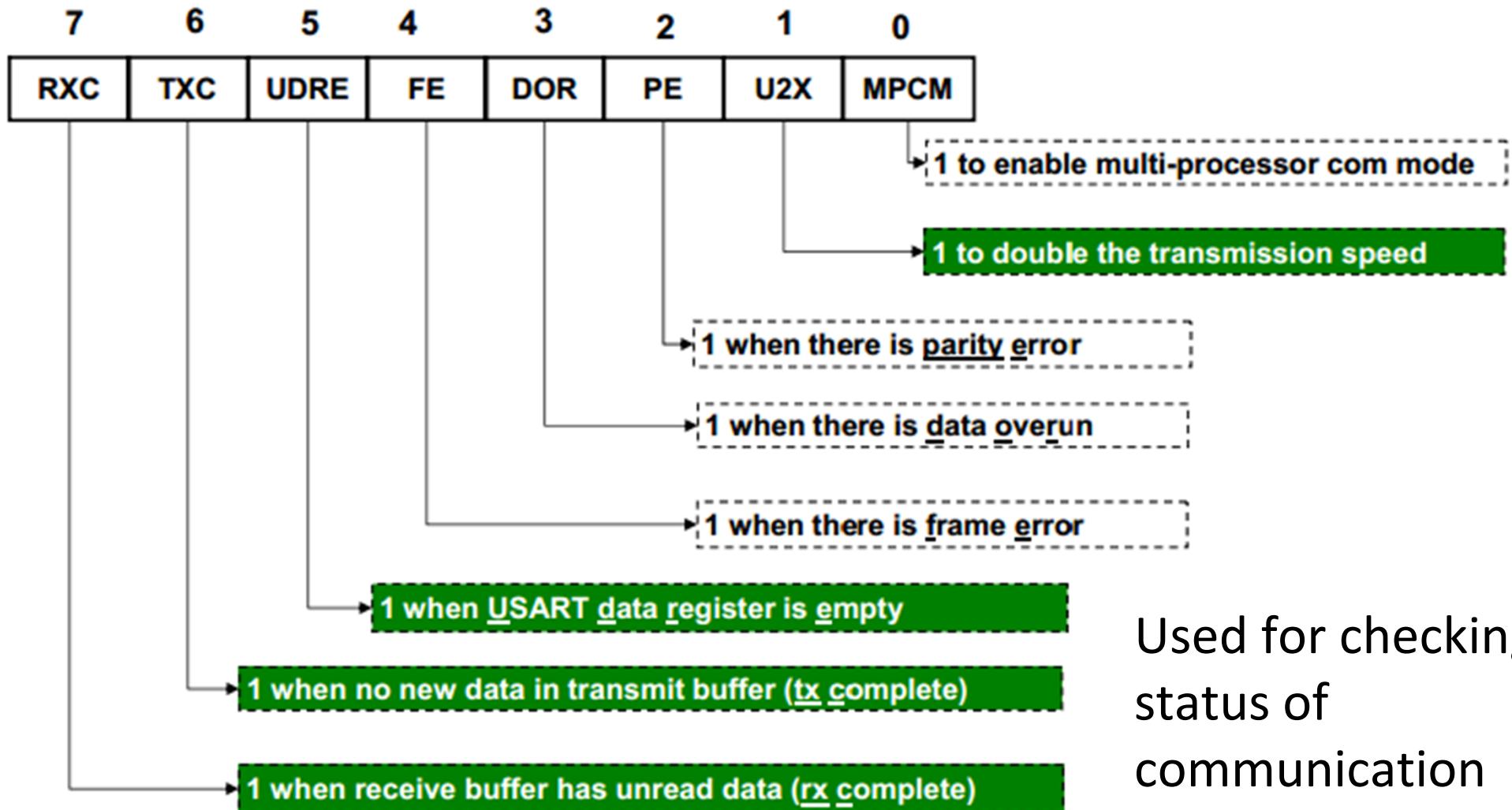
$$\text{baud rate} = \frac{\text{system clock frequency (Hz)}}{16(\text{UBRR} + 1)}$$

$$\text{UBRR} = \frac{\text{system clock frequency (Hz)}}{16 \times \text{baud rate}} - 1$$

- $\text{UBRR} = 1000000 / (16 \times 1200) - 1 = 51.08 \approx 51\text{d} = 0033\text{H}.$
- Therefore, $\text{UBRRH} = 0x00$ and $\text{UBRRL} = 0x33$



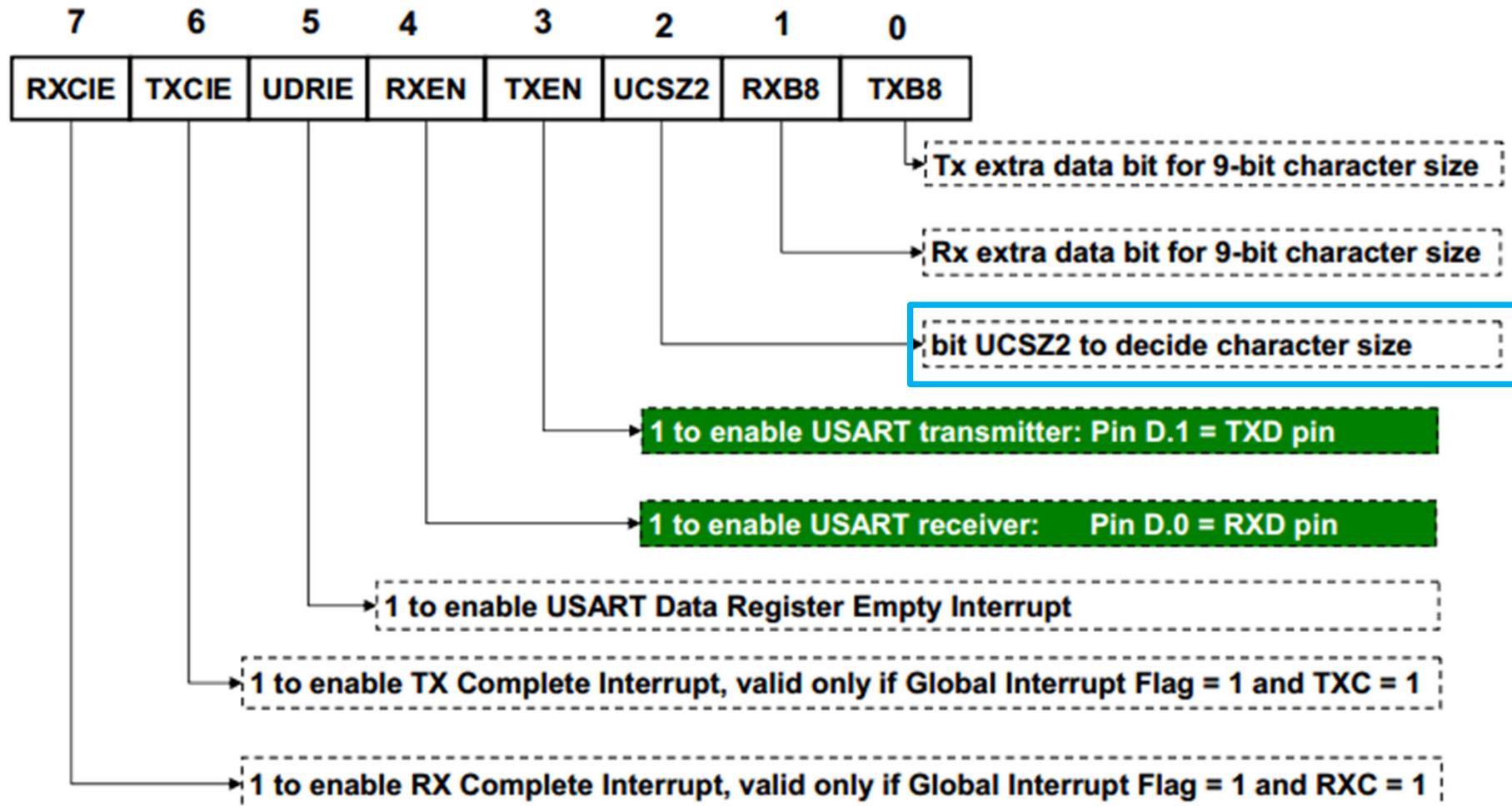
USART Control and Status Register A (UCSRA)



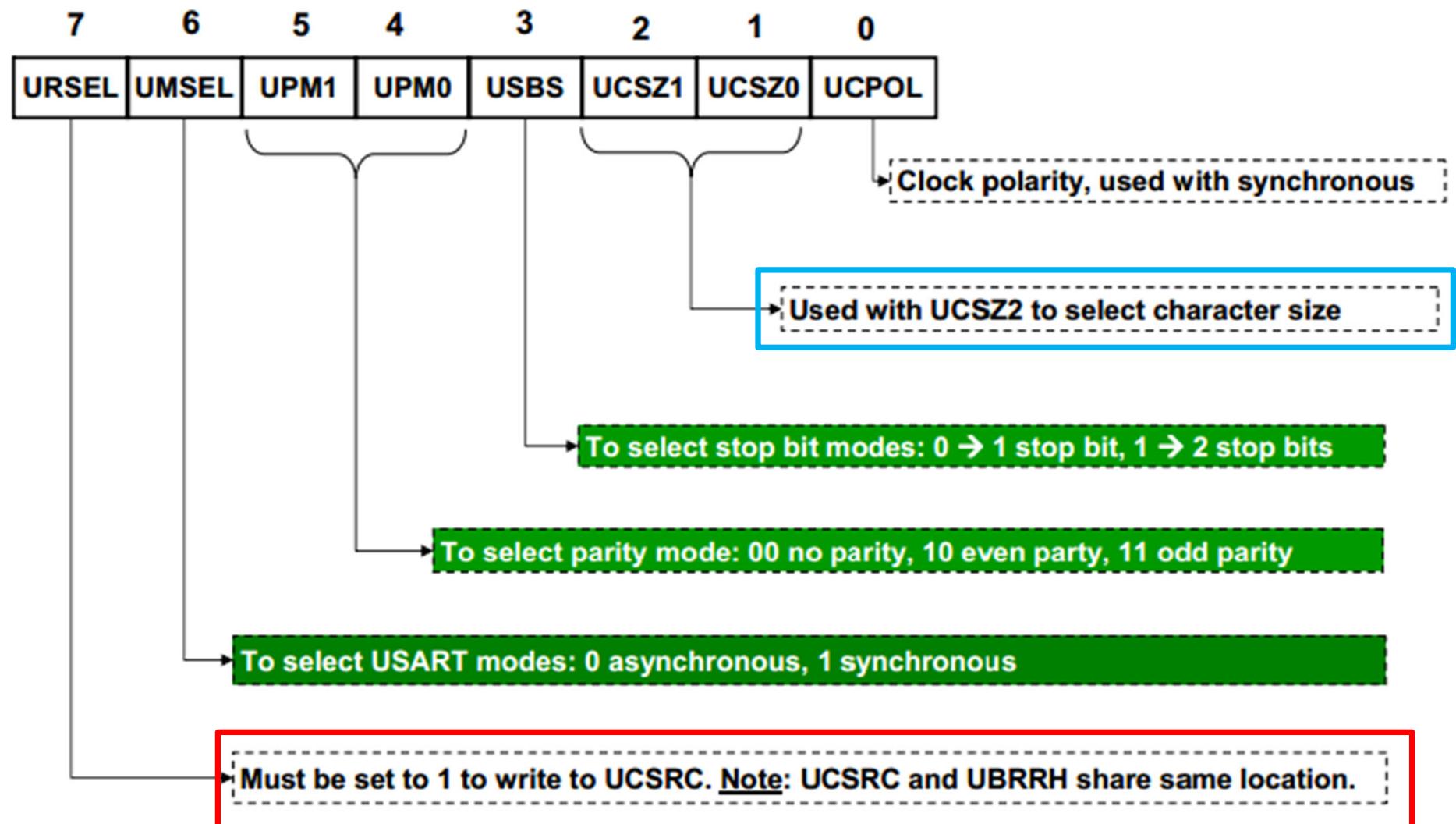
Used for checking status of communication



USART Control and Status Register B (UCSRB)



USART Control and Status Register C (UCSRC)



Setting character size

- Characters can be 5 to 9 bits long
- Three bits are designated to configure the character size
 - bit UCSZ2 (in register UCSRB)
 - bit UCSZ1 and UCSZ0 (in register UCSRC)



Setting character size

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



USART Data Register - UDR

- Register UDR is the buffer for characters sent or received through the serial port

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



USART Data Register - UDR

- Register UDR is the buffer for characters sent or received through the serial port

To start sending a character, we write it to UDR.

```
unsigned char data;  
data = 'a';  
UDR = data;           // start sending character
```

To check a received character, we read it from UDR.

```
unsigned char data;  
data = UDR;          // this will clear UDR
```

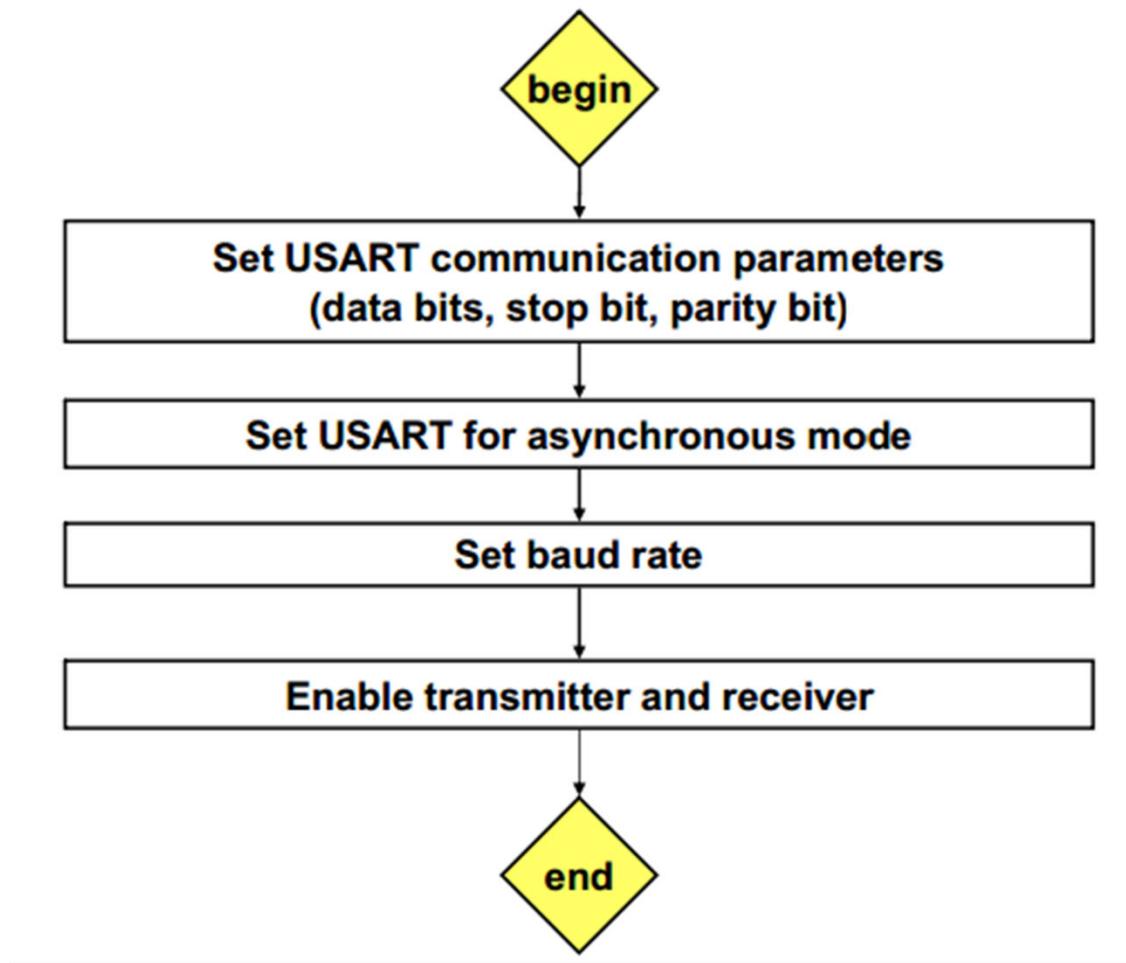


Learning Goals

- Initializing the serial port.
- Sending and receiving a character.
- Sending/receiving formatted strings



Initializing Serial Port



**Initialize ATmega32 for UART with
the following parameters: baud
rate 1200 bps, no parity,
1 stop bit, 8 data bits.**

**Assume a clock speed of 1MHz and
polling approach.**



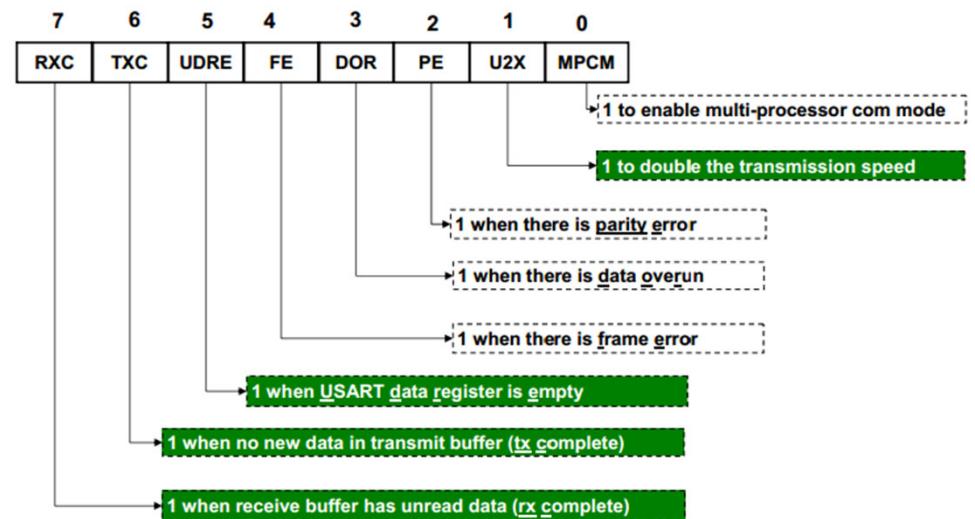
C Code

```
void UART_init(void){  
    // Normal speed, disable multi-proc  
    UCSRA = 0b00000000;  
    // Enable Tx and Rx, disable interrupts  
    UCSRB = 0b00011000;  
    // Asynchronous mode, no parity, 1 stop bit, 8 data bits  
    UCSRC = 0b10000110;  
    // Baud rate 1200bps, assuming 1MHz clock  
    UBRRL = 0x33;  
    UBRRH = 0x00;  
}
```



C Code

```
void UART_init(void){  
    // Normal speed, disable multi-proc  
    UCSRA = 0b00000000;  
  
    // Enable Tx and Rx, disable interrupts  
    UCSRB = 0b00011000;  
  
    // Asynchronous mode, no parity, 1 stop bit, 8 data bits  
    UCSRC = 0b10000110;  
  
    // Baud rate 1200bps, assuming 1MHz clock  
    UBRRH = 0x33;  
    UBRLR = 0x00;  
}
```



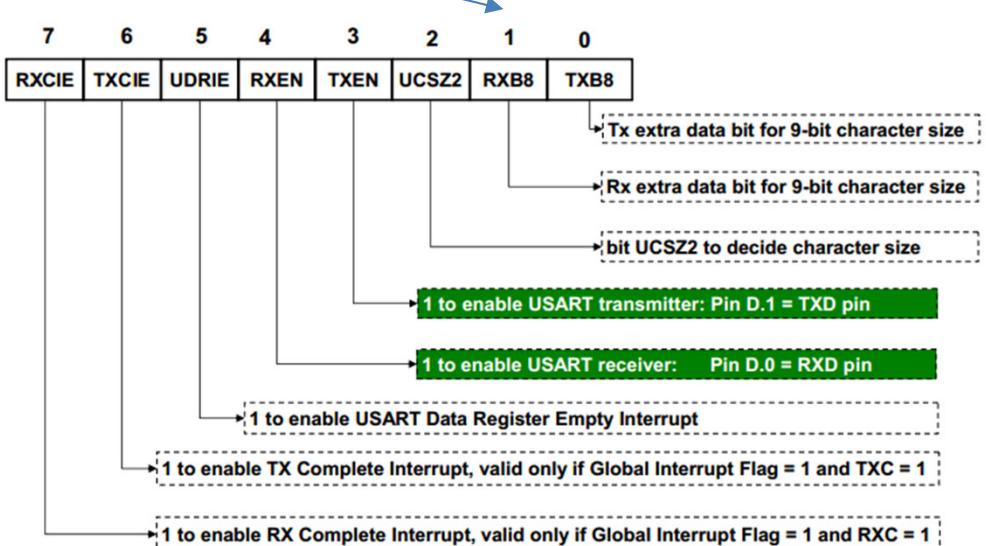
C Code

```

void UART_init(void){
    // Normal speed, disable multi-proc
    UCSRA = 0b00000000;
    // Enable Tx and Rx, disable interrupts
    UCSRB = 0b00011000;
    // Asynchronous mode, no parity, 1 stop bit, 8 data bits
    UCSRC = 0b10000110;
    // Baud rate 1200bps, assuming 1MHz clock
    UBRRH = 0x33;
    UBRLR = 0x00;
}

```

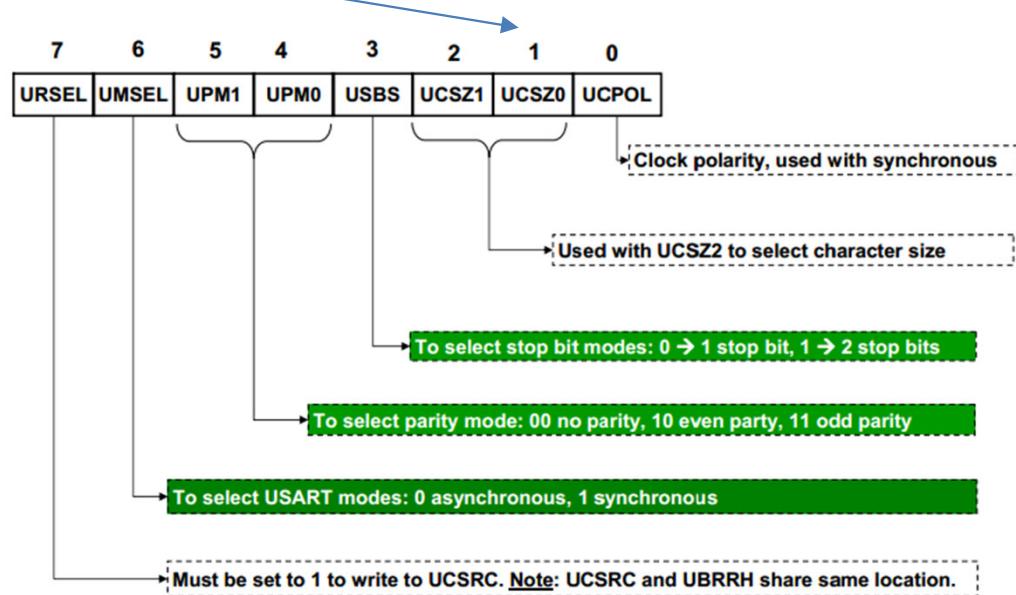
UCSZ2	UCSZ1	UCSZ0	Character Size
0	0	0	5-bit
0	0	1	6-bit
0	1	0	7-bit
0	1	1	8-bit
1	0	0	Reserved
1	0	1	Reserved
1	1	0	Reserved
1	1	1	9-bit



C Code

```
void UART_init(void){  
    // Normal speed, disable multi-proc  
    UCSRA = 0b00000000;  
    // Enable Tx and Rx, disable interrupts  
    UCSRB = 0b00011000;  
    // Asynchronous mode, no parity, 1 stop bit, 8 data bits  
    UCSRC = 0b10000110;  
    // Baud rate 1200bps, assuming 1MHz clock  
    UBRRH = 0x33;  
    UBRRH = 0x00;  
}
```

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



C Code

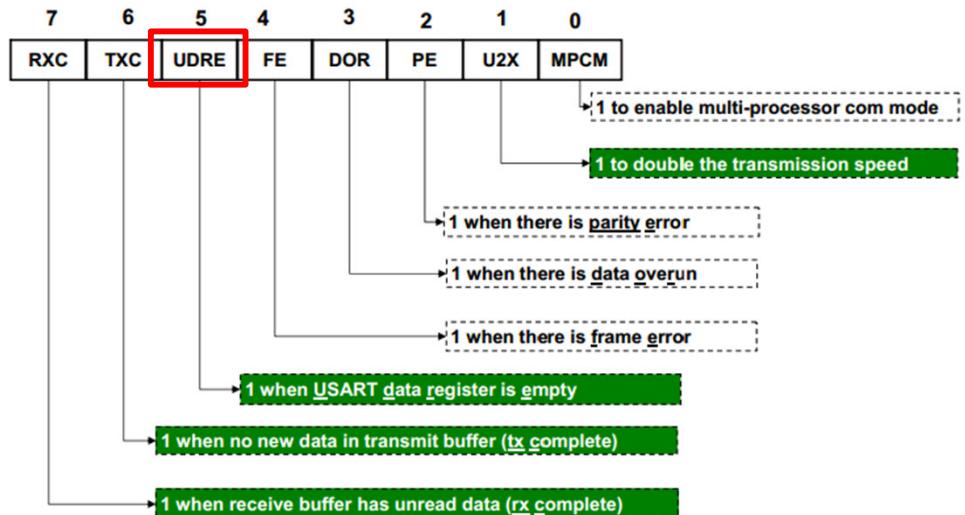
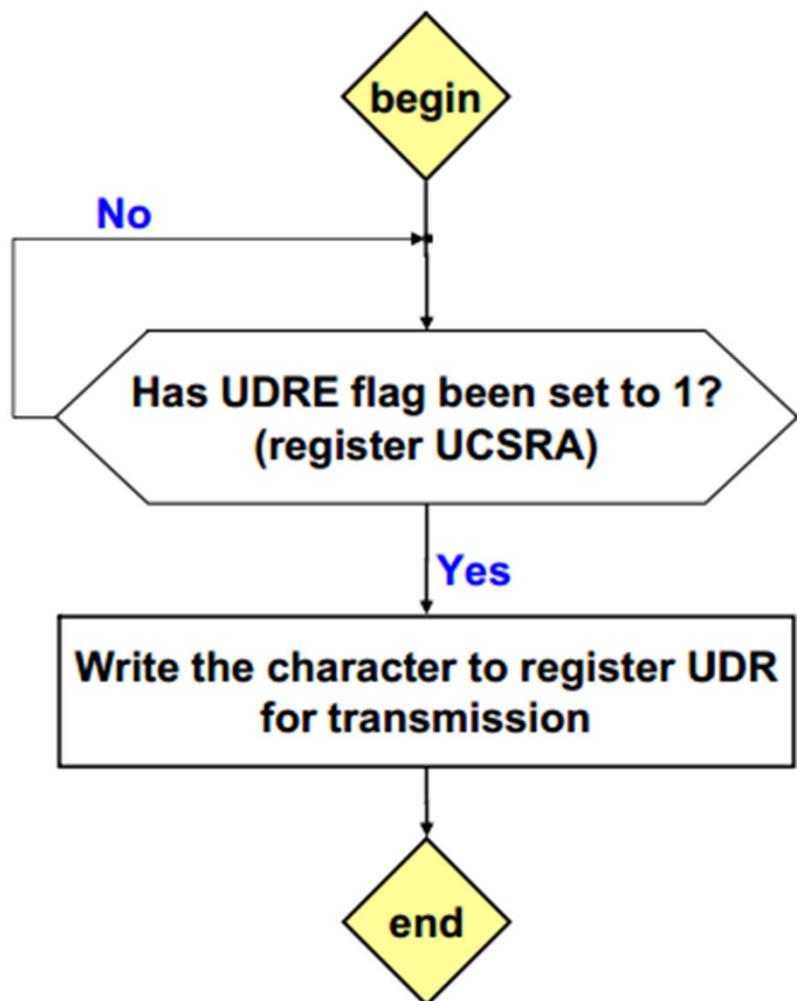
```
void UART_init(void){  
    // Normal speed, disable multi-proc  
    UCSRA = 0b00000000;  
    // Enable Tx and Rx, disable interrupts  
    UCSRB = 0b00011000;  
    // Asynchronous mode, no parity, 1 stop bit, 8 data bits  
    UCSRC = 0b10000110;  
    // Baud rate 1200bps, assuming 1MHz clock  
    UBRRH = 0x00;  
    UBRLL = 0x33;  
}
```

$$\text{baud rate} = \frac{\text{system clock frequency (Hz)}}{16(\text{UBRR} + 1)}$$

$$\text{UBRR} = \frac{\text{system clock frequency (Hz)}}{16 \times \text{baud rate}} - 1$$

- UBRR = $1000000 / (16 \times 1200) - 1 = 51d = 0033H$.
- Therefore, UBRRH = 0x00 and UBRLL = 0x33

Sending a character (Polling)



**Write a C function to send a
character through the serial port of
ATmega16 (Polling)**

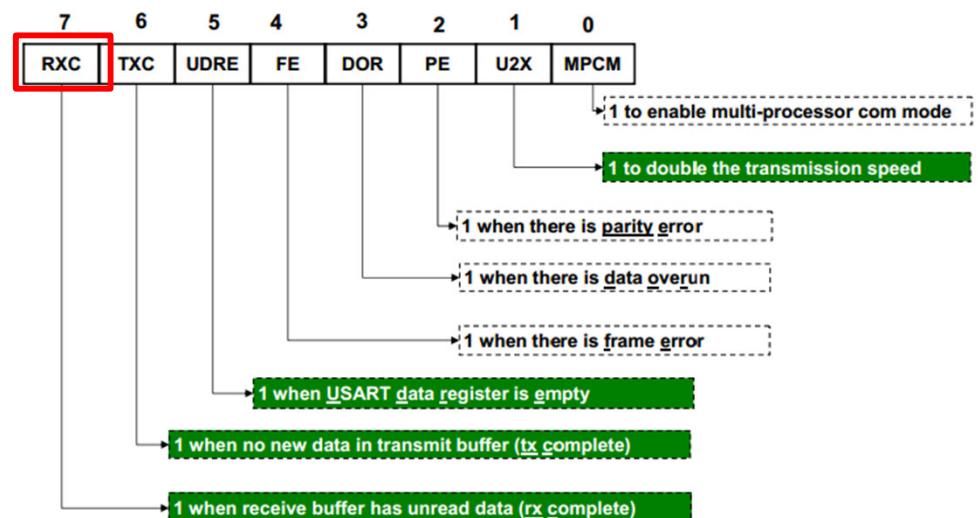
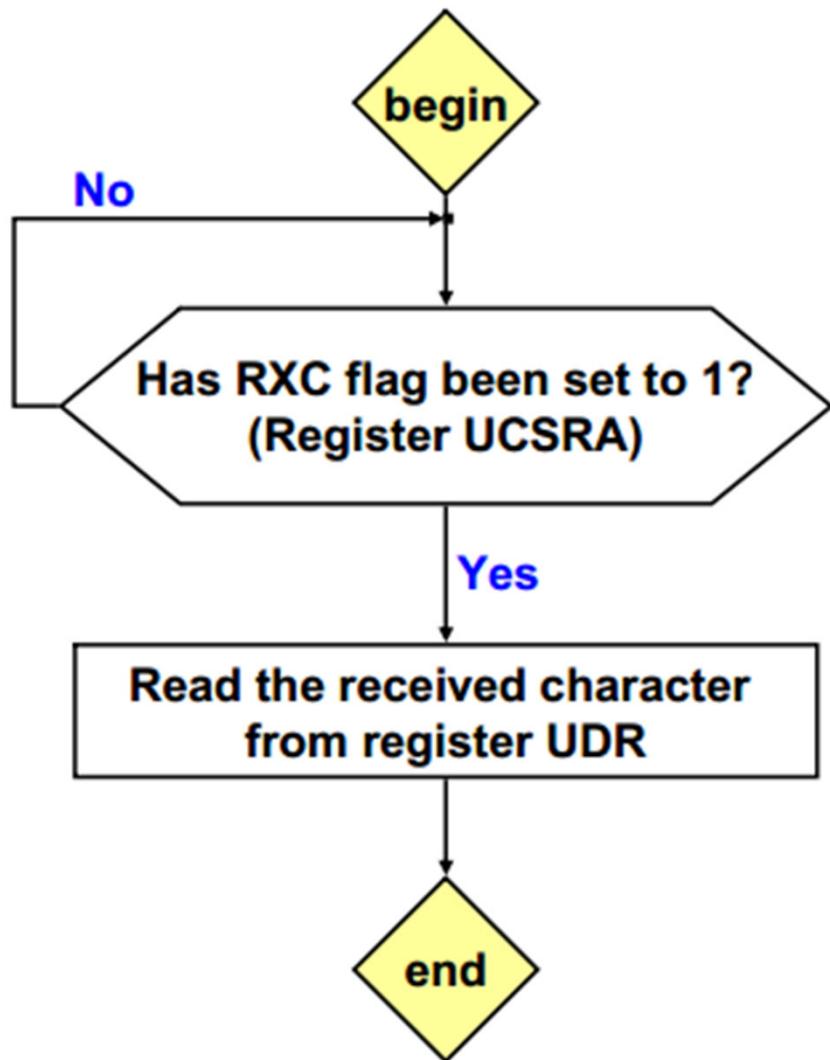


C Code

```
void UART_send(unsigned char data){  
    // wait until UDRE flag is set to logic 1  
    while ((UCSRA & (1<<UDRE)) == 0x00);  
    UDR = data; // Write character to UDR for  
                // transmission  
}
```



Receiving a character (Polling)



**Write a C function to receive a
character through the serial port of
ATmega16 (Polling)**



C Code

```
unsigned char UART_receive(void){  
    // Wait until RXC flag is set to logic 1  
    while (((UCSRA & (1<<RXC)) == 0x00);  
    return UDR; // Read the received  
                character from UDR  
}
```



Sending/receiving formatted strings

- 1) Write two functions to send and receive a character through serial port.
 - Already done - `UART_send` and `UART_receive`
- 2) In main program, call `fdevopen()` to designate the two functions as the handlers for standard output and standard input device.
- 3) Use `printf`/`scanf` as usual. Formatted strings will be sent/received through serial port.



C Code

```
#include <avr/io.h>
#include <stdio.h>

int main(void)
{
    unsigned char a;
    // ... Code to initialise baudrate, TXD, RXD, and so on is
not shown here
    // Initialise the standard IO handlers
    stdout = fdevopen(UART_send, NULL);
    stdin = fdevopen(NULL, UART_receive);
    // Start using printf, scanf as usual
    while (1){
        printf("\n\rEnter a = ");
        scanf("%d", &a); printf("%d", a);
    }
}
```



Now, it is time to implement our
knowledge in a sample
application



Program a camera to rotate repetitively

- Connection Parameters
 - 8 data bit
 - 1 stop bit
 - no parity bit
 - baud rate 9600bps
- Control: Sending character “4” or “6” turns the camera left or right, respectively



Setting Baud Rate

$$\text{baud rate} = \frac{\text{system clock frequency (Hz)}}{16(\text{UBRR} + 1)}$$

$$\text{UBRR} = \frac{\text{system clock frequency (Hz)}}{16 \times \text{baud rate}} - 1$$

- In normal mode, the calculation leads to a UBR of $6.51 - 1 = 5.51$
- Hence the UBR would have to be set to 5 or, 6
- Thus we will use double speed mode to be more precise



Setting Baud Rate

U2X = 0

$$UBRR = \frac{\text{system clock frequency (Hz)}}{16 \times \text{baud rate}} - 1$$
$$= \frac{10^6}{16 \times 9600} - 1 = 5.51 \approx 6$$

U2X = 1

$$UBRR = \frac{\text{system clock frequency (Hz)}}{8 \times \text{baud rate}} - 1$$
$$= \frac{10^6}{8 \times 9600} - 1 = 12.02 \approx 12 \text{ (more precise)}$$



C Code - Initialize

```
void USART_init(void)
{
    UCSRA = 0b00000010; // double speed
    UCSRB = 0b00011000; // Enable Tx and Rx, polling
    UCSRC = 0b10000110; // Async mode, no parity, 1 stop bit,
                        // 8 data bits
    //in double-speed mode, UBRR = clock/(8xbaud rate) - 1
    UBRRH = 0;
    UBRLR = 12; // Baud rate 9600bps, assuming 1MHz clock
}
```



C Code – Main Loop

```
int main(void)
{
    unsigned char i;
    USART_init(); // initialise USART
    while (1) {
        for (i=0; i<10; i++)// rotate left 10 times
        {
            UART_send('4');
            delay();
        }
        for (i=0; i < 10; i++)//rotate right 10 times
        {
            UART_send('6');
            delay();
        }
    }
}
```



Sending 9 bit characters

- The ninth bit must be written to the TXB8 bit in UCSRB before the low byte of the character is written to UDR

```
void USART_Transmit( unsigned int data )
{
    /* Wait for empty transmit buffer */
    while ( !( UCSRA & (1<<UDRE)) ) ;
    /* Copy 9th bit to TXB8 */
    UCSRB &= ~(1<<TXB8);
    if ( data & 0x0100 )
        UCSRB |= (1<<TXB8);
    /* Put data into buffer, sends the data */
    UDR = data;
}
```



Reception of 9 bit characters and status flags

- Reading UDR changes the RXB8, FE, DOR and PE bits
- Always read status from UCSRA and the 9th bit from RXB8 in UCSRB **BEFORE**
 - reading data from UDR.



Reception of 9 bit characters and status flags

```
unsigned int USART_Receive( void )
{
    unsigned char status, resh, resl;
    /* Wait for data to be received */
    while ( !(UCSRA & (1<<RXC)) )
        ;
    /* Get status and 9th bit, then data */
    /* from buffer */
    status = UCSRA;
    resh = UCSRB;
    resl = UDR;
    /* If error, return -1 */
    if ( status & (1<<FE) | (1<<DOR) | (1<<PE) )
        return -1;
    /* Filter the 9th bit, then return */
    resh = (resh >> 1) & 0x01;
    return ((resh << 8) | resl);
}
```



Transmission

- To start transmission data is written in the buffer (UDR)
- The buffered data will be moved to the Shift Register when the Shift Register is ready to send a new frame, i.e, it has no bits left to send.
 - if it is in idle state (no ongoing transmission) or
 - immediately after the last stop bit of the previous frame is transmitted.



Reception

- The receiver starts data reception when it detects a valid start bit.
- Each bit that follows the start bit will be sampled at the baud rate or XCK clock, and shifted into the receive Shift Register until the first stop bit of a frame is received.
- A second stop bit will be ignored by the receiver.



Reception

- When the first stop bit is received, a complete serial frame is present in the receive Shift Register, the contents of the Shift Register is then moved into the receive buffer.
- The receive buffer can then be read from UDR.



Frame Error

- A UART will detect a framing error when it does not see a "stop" bit at the expected "stop" bit time.
- A FE (Frame Error) will be detected in the cases where the first stop bit is zero.
 - The receiver ignores the second stop bit



Data OverRun Error (DOR)

- The Data OverRun (DOR) Flag indicates data loss due to a receiver buffer full condition.
- A Data OverRun occurs when the receive buffer (UDR) is full (two characters), it is a new character waiting in the receive Shift Register, and a new start bit is detected.
- If the DOR Flag is set there was one or more serial frame lost between the frame last read from UDR, and the next frame read from UDR.



Checking Transmission Status

- The USART transmitter has two flags that indicate its state:
 - USART Data Register Empty (UDRE)
 - Transmit Complete (TxC).
- The Data Register Empty (UDRE) Flag indicates whether the transmit buffer is ready to receive new data.
 - This bit is set when the transmit buffer is empty
 - and not set when not empty (has unmoved data to shift register).



Checking Transmission Status

- The Transmit Complete (TXC) Flag bit is set one when
 - the entire frame in the transmit Shift Register has been shifted out, and
 - there are no new data currently present in the transmit buffer.
- The TXC Flag bit is automatically cleared when
 - a transmit complete interrupt is executed, or
 - it can be cleared by writing a one to its bit location



Checking TXC flag to check transmission status

- The TXC Flag must be cleared before each transmission (before UDR is written)
- If, not, then what will be the problem ???



Transmission using UDRIE Interrupt

- The USART Data Register Empty Interrupt will be executed as long as UDRE is set.
 - UDRE is cleared by writing UDR.
- When interrupt-driven data transmission is used, the Data Register Empty Interrupt routine must
 - either write new data to UDR in order to clear UDRE or
 - disable the Data Register empty Interrupt
- No such problem with TXCIE interrupt



Reception using RXCIE interrupt

- When the Receive Complete Interrupt Enable (RXCIE) in UCSRB is set, the USART Receive Complete Interrupt will be executed as long as the RXC Flag is set
- The receive complete routine must read the received data from UDR in order to clear the RXC Flag
 - otherwise a new interrupt will occur once the interrupt routine terminates.



Double Speed Mode Disadvantage

- The receiver will use half the number of samples (reduced from 16 to 8) for data sampling and clock recovery (syncing the internal clock to the incoming serial frames)
- Hence, a more accurate baud rate setting and system clock are required.
- For the transmitter, there are no downsides.



Disabling Transmission and Reception

- The disabling of the transmitter (setting the TXEN to zero) will not become effective until ongoing and pending transmissions are completed
- The disabling of reception (setting the RXEN to zero) is immediate
 - Disabling reception flushes the receive buffer immediately



Asynchronous Data Reception

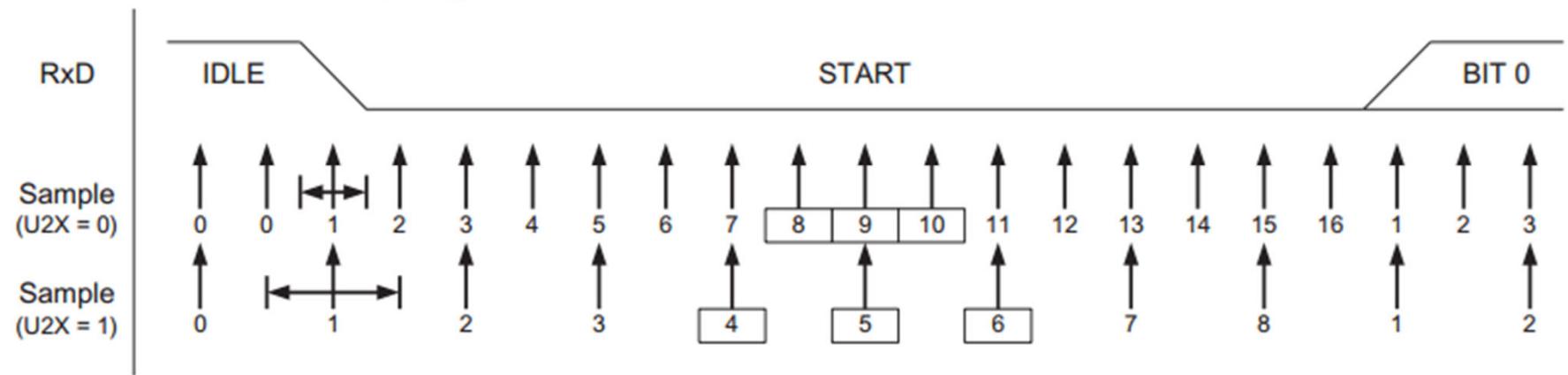
- Clock Recovery
 - Detect Start Bit
- Data Recovery
 - Detect Data Bits, Parity Bit, Stop Bit



Asynchronous Clock Recovery

- The clock recovery logic synchronizes internal clock to the incoming serial frames.
- The sample rate is 16 times the baud rate for Normal mode, and 8 times the baud rate for Double Speed mode.

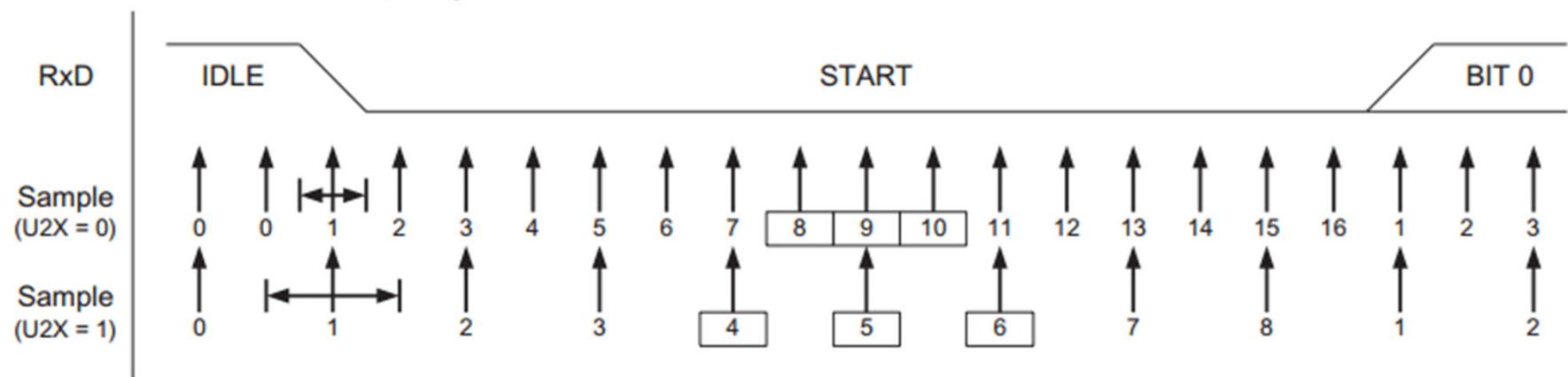
Figure 73. Start Bit Sampling



Asynchronous Clock Recovery

- Samples denoted zero are samples done when the RxD line is idle (that is, no communication activity).
- When the clock recovery logic detects a high (idle) to low (start) transition on the RxD line, the start bit detection sequence is initiated.

Figure 73. Start Bit Sampling



Asynchronous Clock Recovery

- The clock recovery logic uses samples 8, 9, and 10 for Normal mode, and samples 4, 5, and 6 for Double Speed mode
- If two or more of these three samples have logical high levels (the majority wins), the start bit is rejected as a noise spike and the receiver starts looking for the next high to low-transition.



Asynchronous Clock Recovery

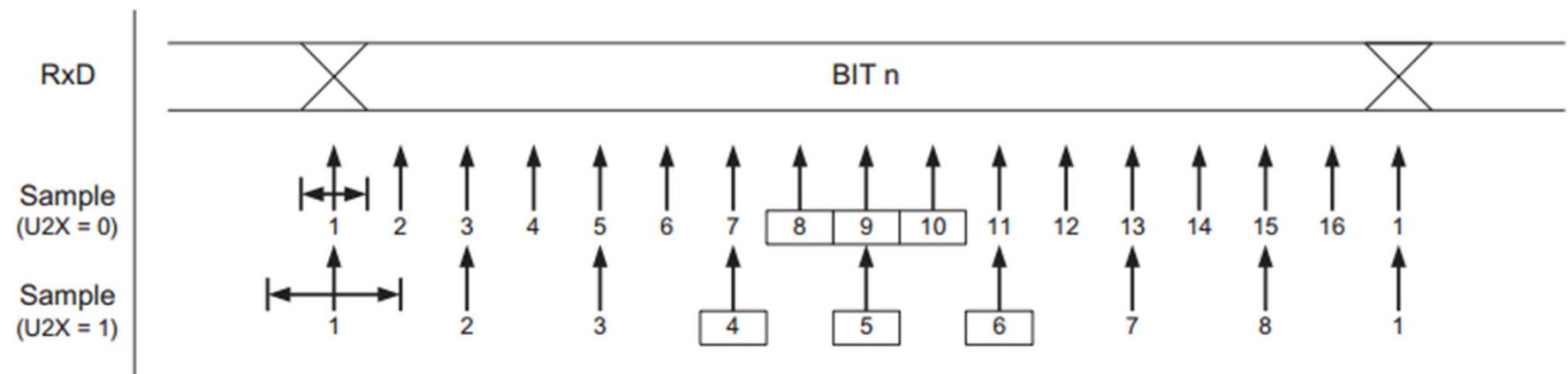
- If however, a valid start bit is detected, the clock recovery logic is synchronized and the data recovery can begin.
- The synchronization process is repeated for each start bit.



Asynchronous Data Recovery

- The decision of the logic level of the received bit is taken by doing a majority voting of the logic value to the three samples in the center of the received bit.
- The parity bit and the first stop bit is also sampled this way

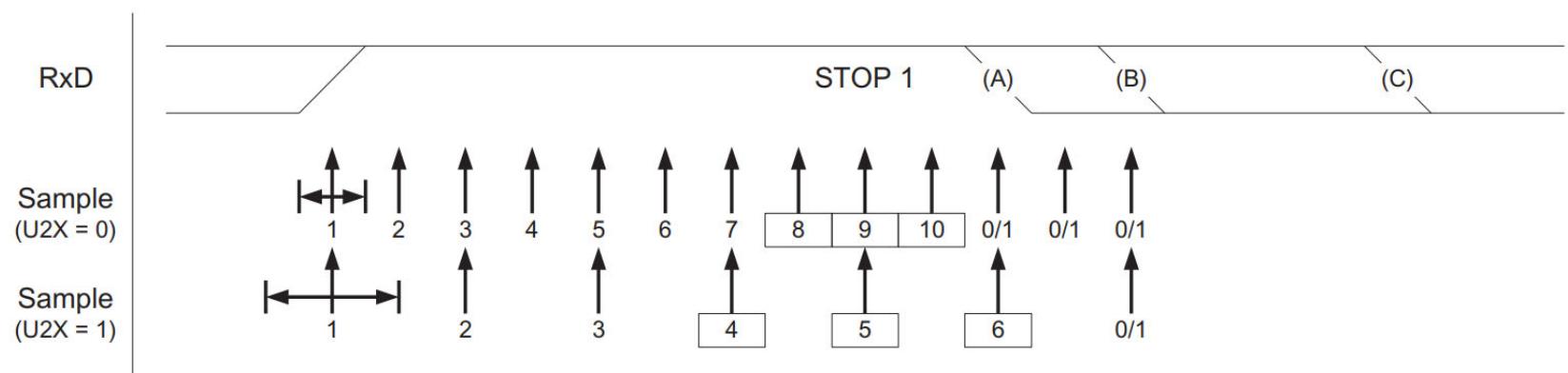
Figure 74. Sampling of Data and Parity Bit



Asynchronous Data Recovery

- The same majority voting is done to the stop bit as done for the other bits in the frame.
- If the stop bit is registered to have a logic 0 value, the Frame Error (FE) Flag will be set.

Figure 75. Stop Bit Sampling and Next Start Bit Sampling



SPI



Serial Peripheral Interface (SPI)

- The receiver and transmitter share a common clock line.
- The device considered the 'Master' provides a clock signal used to synchronize data transactions between the two devices.
- Supports **higher data rates** than UART.
- A SPI interface uses a 3-wire system
 - MOSI, MISO, SCK
 - There is also a slave select pin, which is used to activate slave



PIN Description

SPI uses 4 pins for communications

1. MISO – **MISO stands for Master In Slave Out.**
MISO is the input pin for Master, and output pin for Slave device. Data transfer from Slave to Master takes place through this channel.
2. MOSI – **MOSI stands for Master Out Slave In.**
This pin is the output pin for Master and input pin for Slave. Data transfer from Master to Slave takes place through this channel.

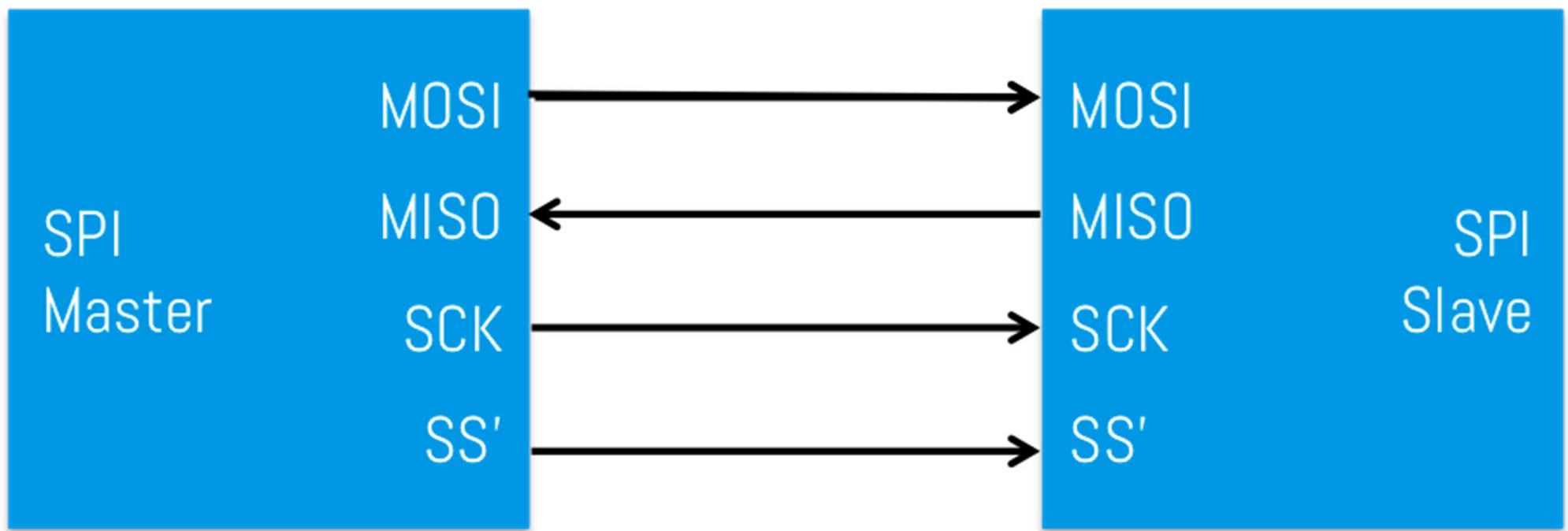


PIN Description

3. SCK – This is the SPI clock line
 - Master generates a clock signal in this PIN which is fed to the slave
4. SS – This stands for Slave Select.
 - This pin is used to activate/select a slave



SPI Connection: Single Slave



maxEmbedded.com
a guide to robotics and embedded systems

SPI Bus: Single Master
Single Slave



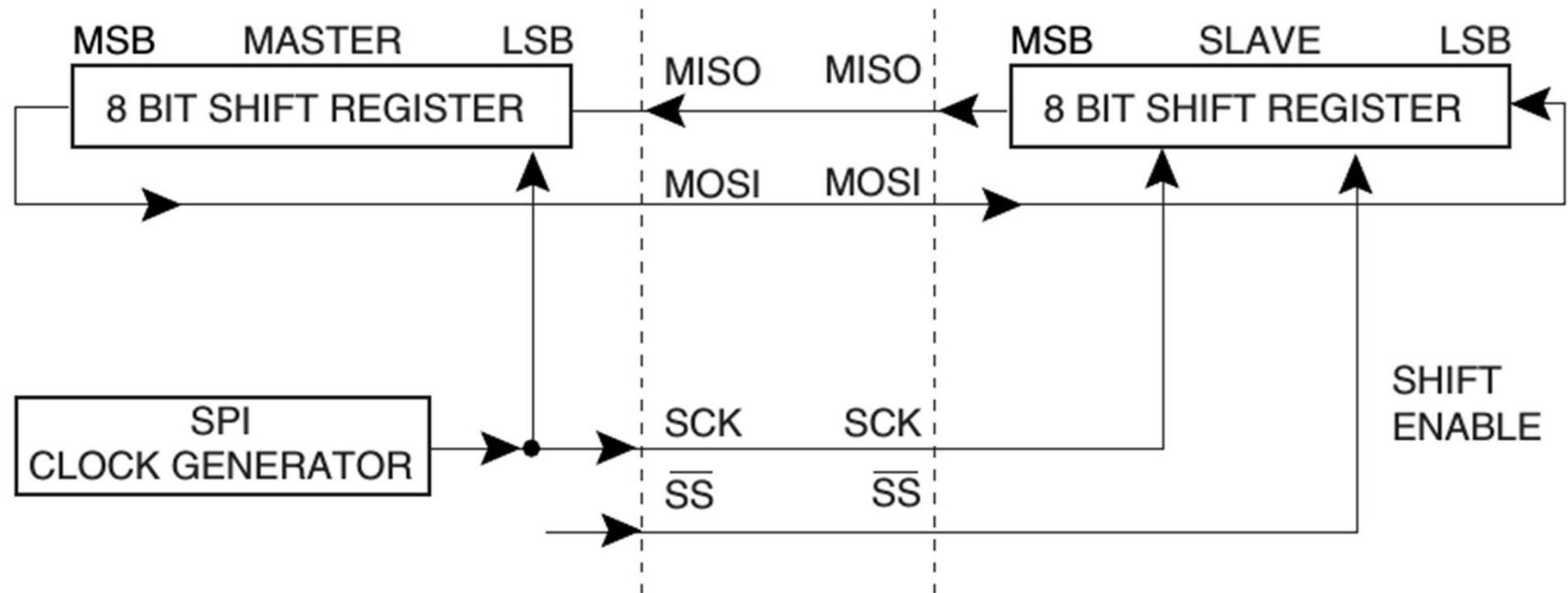
SPI

- Both, Master and Slave **place the data (byte)** they wish to transfer in their **respective shift registers** before the communication starts.
- Master generates 8 clock pulses. After each clock pulse, one bit of information is transferred from Master to Slave and vice-versa.
- After 8 clock pulses, Master would have received Slave's data, whereas Slave would have Master's data. And that's why this is a **full-duplex** communication.



SPI Master-slave Interconnection

Figure 66. SPI Master-slave Interconnection



- Slave select line should activate slave before starting transmission and deactivate it after finishing communication.
- Any free pin of the Master can be used to drive the Slave Select line of the slave
- The SS line of Master, if configured as output is typically used to drive a slave



SPI Master-slave Interchange

- Example of Master Slave byte exchange per clock pulse
- This example assumes MSB is transferred first (DORD=0)

Pulse	Master	Slave
0	10100110	11001011
1	01001101	10010111
2	10011011	00101110
3	00110110	01011101
4	01101100	10111010
5	11011001	01110100
6	10110010	11101001
7	01100101	11010011
8	11001011	10100110



Slave Select Line Functionality - Slave Mode

- When the SPI is configured as a Slave, the Slave Select (SS) pin is always input.
- When **SS is held low, the SPI is activated**
- When SS is driven high, it will not receive incoming data.



Slave Select Line Functionality - Master Mode

- When the SPI is configured as a Master (MSTR in SPCR is set), the user can determine the direction of the SS pin.
- If SS is configured as an output, the pin is a general output pin which does not affect the SPI system.
- Typically, the pin will be driving the SS pin of the SPI Slave.



Slave Select Line Functionality - Master Mode

- When configured as a Master, the SPI interface has **no automatic control of the SS line.**
- This must be handled by user software before communication can start.



SPI Pin Overrides

When the SPI is enabled, the data direction of the MOSI, MISO, SCK, and SS pins is overridden according to following table

Pin	Direction, Master SPI	Direction, Slave SPI
MOSI	User Defined	Input
MISO	Input	User Defined
SCK	User Defined	Input
SS	User Defined	Input



Register Descriptions

Three registers deal with SPI:

- SPCR – SPI Control Register
 - It contains the bits to initialize SPI and control it.
- SPSR – SPI Status Register
 - This register is used to read the status of the bus lines.
- SPDR – SPI Data Register
 - The SPI Data Register is the read/write register where the actual data transfer takes place.



SPI - Transmission Steps

1. The SPI Master initiates the communication cycle by pulling low the Slave Select pin of the desired Slave.
2. Master writes the byte to send in SPDR
3. Step 2 triggers the the SPI clock generator of the Master to generate the required clock pulses on the SCK line to interchange data.



SPI - Transmission Steps

4. After shifting one byte, the SPI clock generator stops, setting the end of Transmission Flag (SPIF). If the SPI Interrupt is enabled an interrupt is issued.
5. The Master may continue to shift the next byte by writing it into SPDR, or signal the end of packet by pulling high the Slave Select line.



SPI Control Register - SPCR

Bit	7	6	5	4	3	2	1	0	SPCR
	SPIE	SPE	DORD	MSTR	CPOL	CPHA	SPR1	SPR0	
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	

- Bit 7 – SPIE: SPI Interrupt Enable
 - This bit causes the SPI interrupt to be executed if SPIF bit in the SPSR Register is set
- Bit 6 – SPE: SPI Enable
 - written one to enable the SPI
- Bit 5 – DORD: Data Order
 - When the DORD bit is written to one, the LSB of the data word is transmitted first.
 - When the DORD bit is written to zero, the MSB of the data word is transmitted first.



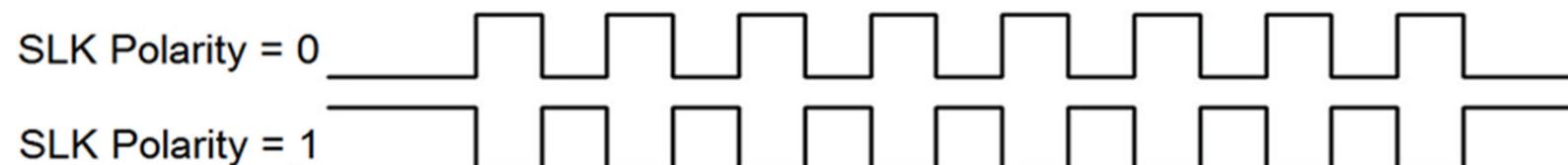
SPI Control Register - SPCR

Bit	7	6	5	4	3	2	1	0	SPCR
	SPIE	SPE	DORD	MSTR	CPOL	CPHA	SPR1	SPR0	
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	

- Bit 4 – MSTR: Master/Slave Select
 - 1 for Master and 0 for Slave
- Bit 3 – CPOL: Clock Polarity
 - The CPOL functionality is summarized below:

Table 56. CPOL Functionality

CPOL	Leading Edge	Trailing Edge	Idle State
0	Rising	Falling	LOW
1	Falling	Rising	HIGH



SPI Control Register - SPCR

Bit	7	6	5	4	3	2	1	0	SPCR
	SPIE	SPE	DORD	MSTR	CPOL	CPHA	SPR1	SPR0	
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	

- Bit 2 – CPHA: Clock Phase
 - The settings of the Clock Phase bit (CPHA) determine if data is sampled on the leading (first) or trailing (last) edge of SCK.

Table 57. CPHA Functionality

CPHA	Leading Edge	Trailing Edge
0	Sample	Setup
1	Setup	Sample



SPI Control Register - SPCR

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

- Bits 1, 0 – SPR1, SPR0: SPI Clock Rate Select 1 and 0

These two bits control the SCK rate of the device configured as a Master. SPR1 and SPR0 have no effect on the Slave. The relationship between SCK and the Oscillator Clock frequency f_{osc} is shown in the following table:

Table 58. Relationship Between SCK and the Oscillator Frequency

SPI2X	SPR1	SPR0	SCK Frequency
0	0	0	$f_{osc}/4$
0	0	1	$f_{osc}/16$
0	1	0	$f_{osc}/64$
0	1	1	$f_{osc}/128$
1	0	0	$f_{osc}/2$
1	0	1	$f_{osc}/8$
1	1	0	$f_{osc}/32$
1	1	1	$f_{osc}/64$

SPI Status Register - SPSR

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

- Bit 7 – SPIF: SPI Interrupt Flag
 - When a serial transfer is complete, the SPIF Flag is set. Interrupt is generated if enabled
 - SPIF is cleared by hardware when executing the corresponding interrupt handling vector.
 - Alternatively, the SPIF bit is cleared by first reading the SPI Status Register with SPIF set, then accessing the SPI Data Register (SPDR).



SPI Status Register - SPSR

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

- Bit 6 – WCOL: Write Collision Flag
 - The WCOL bit is set if the SPI Data Register (SPDR) is written during a data transfer.
- Bit 0 – SPI2X: Double SPI Speed Bit
 - When this bit is written logic one the SPI speed (SCK Frequency) will be doubled when the SPI is in Master mode



SPI Data Register - SPDR

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

- The SPI Data Register is used for data transfer.
- Writing to the register initiates data transmission.
- Reading the register causes the Shift Register Receive buffer to be read.



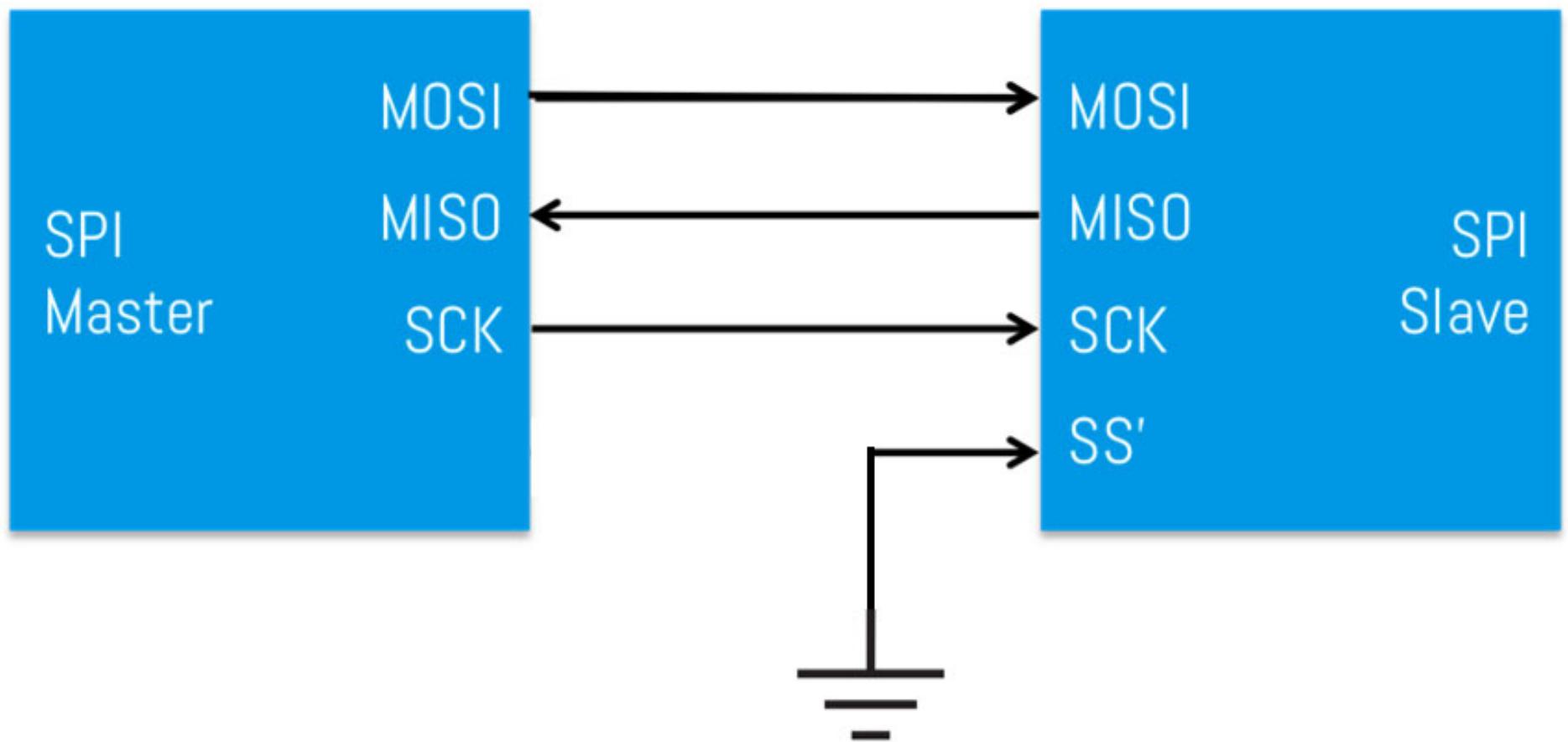
PINOUT

PDIP

(XCK/T0)	PB0	1	40	PA0 (ADC0)
(T1)	PB1	2	39	PA1 (ADC1)
(INT2/AIN0)	PB2	3	38	PA2 (ADC2)
(OC0/AIN1)	PB3	4	37	PA3 (ADC3)
(SS)	PB4	5	36	PA4 (ADC4)
(MOSI)	PB5	6	35	PA5 (ADC5)
(MISO)	PB6	7	34	PA6 (ADC6)
(SCK)	PB7	8	33	PA7 (ADC7)
RESET		9	32	AREF
VCC		10	31	GND
GND		11	30	AVCC
XTAL2		12	29	PC7 (TOSC2)
XTAL1		13	28	PC6 (TOSC1)
(RXD)	PD0	14	27	PC5 (TDI)
(TXD)	PD1	15	26	PC4 (TDO)
(INT0)	PD2	16	25	PC3 (TMS)
(INT1)	PD3	17	24	PC2 (TCK)
(OC1B)	PD4	18	23	PC1 (SDA)
(OC1A)	PD5	19	22	PC0 (SCL)
(ICP1)	PD6	20	21	PD7 (OC2)



SPI – Sample Setup

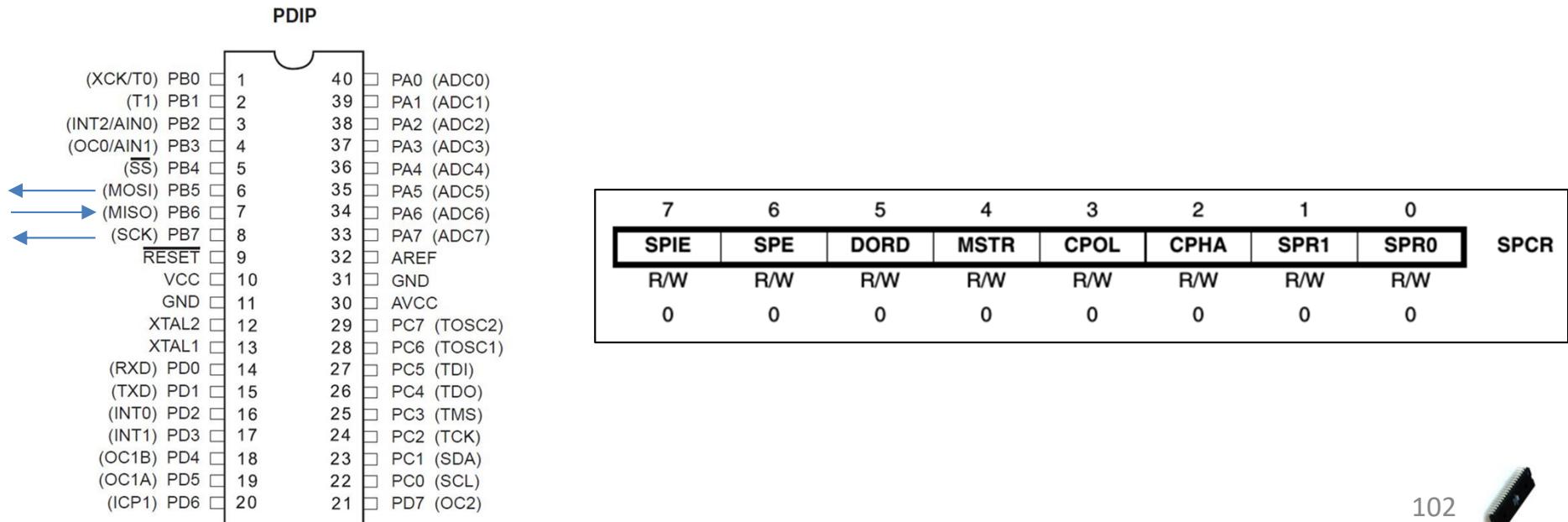


SPI – Master Config

```

void SPI_MasterInit(void)
{
    /* Set MOSI and SCK output, all others input */
    DDRB = (1<<DDB5)|(1<<DDB7);
    /* Enable SPI, Master, set clock rate fck/16 */
    SPCR = (1<<SPE)|(1<<MSTR)|(1<<SPR0);
}

```

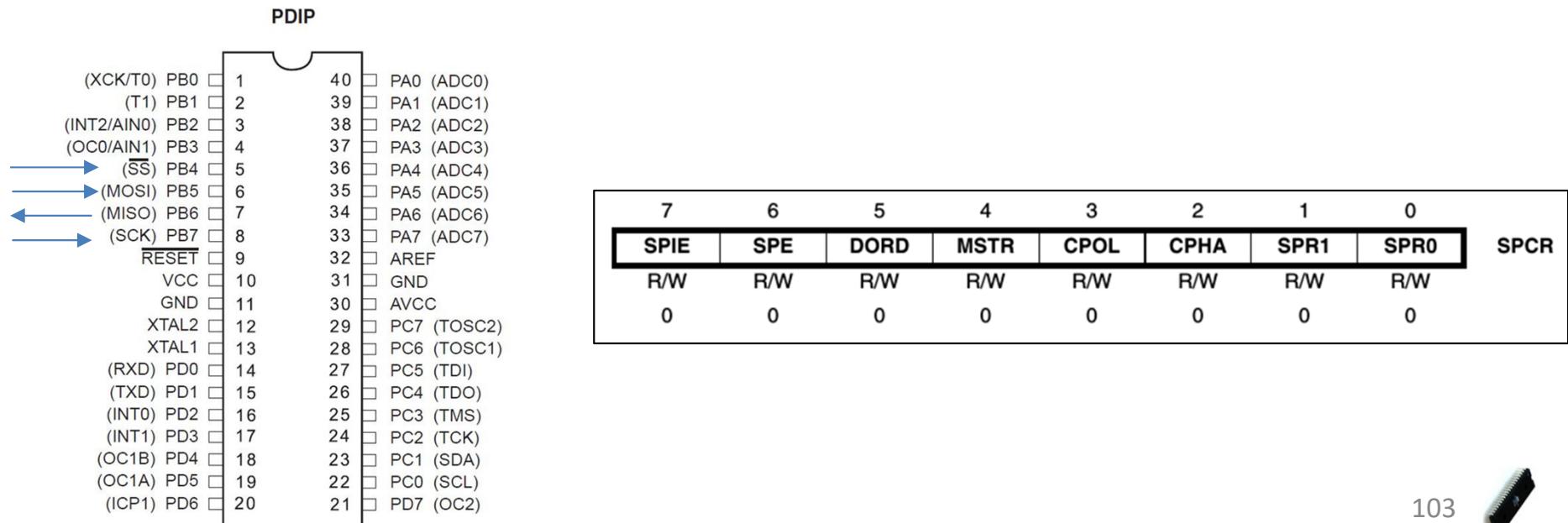


SPI – Slave Config

```

void SPI_SlaveInit(void)
{
    /* Set MISO output, all others input */
    DDRB = (1<<DDB6);
    /* Enable SPI */
    SPCR = (1<<SPE);
}

```



SPI - Data sending

```
void SPI_MasterTransmit(char cData)
{
    /* Start transmission */
    SPDR = cData;
    /* Wait for transmission complete */
    while(!(SPSR & (1<<SPIF)));
}
```



SPI - Data Receiving

```
char SPI_SlaveReceive(void)
{
    /* Wait for reception complete */
    while(!(SPSR & (1<<SPIF)));
    /* Return data register */
    return SPDR;
}
```



How to perform duplex
operation: send and receive
simultaneously ?



SPI – Data Exchange

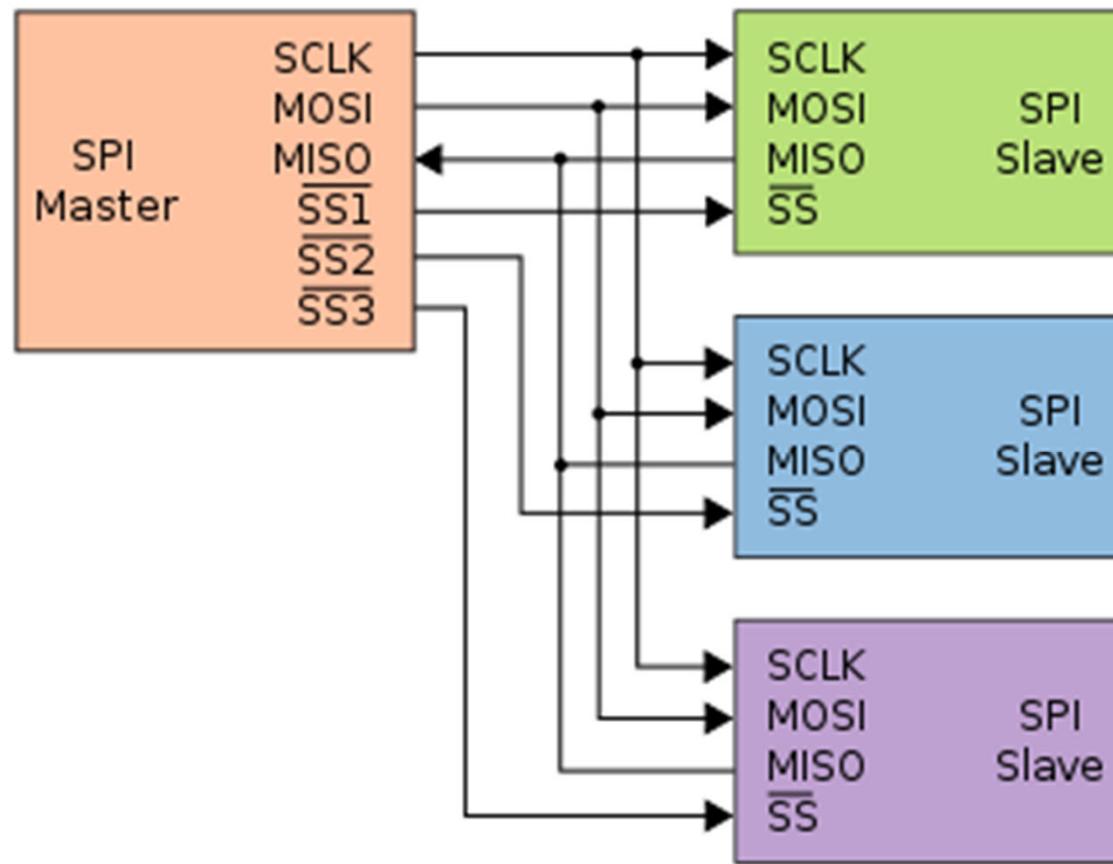
```
char SPI_Transceiver(char cData)
{
    /* Start transmission */
    SPDR = cData;
    /* Wait for transmission complete */
    while(!(SPSR & (1<<SPIF)));
    /* Return newly received data */
    Return SPDR;
}
```

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

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



Working with Multiple Slave



- All the Slaves share the same MOSI, MISO and SCK signals.
- The SS' signal is responsible for choosing a particular Slave.
- The Slave gets enabled only when its input SS' signal goes LOW.



SPDR buffer

- The system is single buffered in the transmit direction and double buffered in the receive direction.
- This means that bytes to be transmitted cannot be written to the SPI Data Register before the entire shift cycle is completed.
- When receiving data, however, a received character must be read from the SPI Data Register before the next character has been completely shifted in. Otherwise, the first byte is lost.

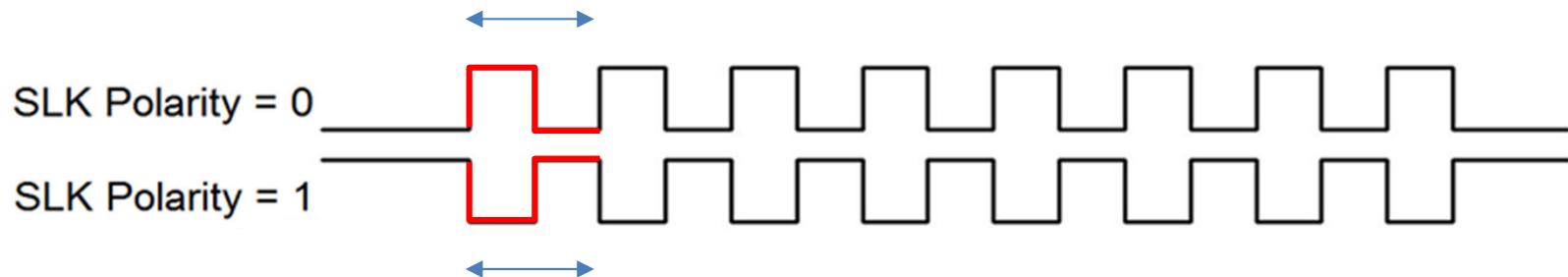


Data Modes

- There are four combinations of SCK phase and polarity with respect to serial data, which are determined by control bits CPHA and CPOL.
- Data bits are shifted out and latched in on opposite edges of the SCK signal
 - ensuring sufficient time for data signals to stabilize.

Sample = Data Reading
Setup = Shifting



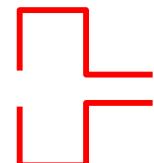


- **Bit 3 – CPOL: Clock Polarity**

When this bit is written to one, SCK is high when idle. When CPOL is written to zero, SCK is low when idle. Refer to [Figure 67](#) and [Figure 68](#) for an example. The CPOL functionality is summarized below:

Table 56. CPOL Functionality

CPOL	Leading Edge	Trailing Edge
0	Rising	Falling
1	Falling	Rising



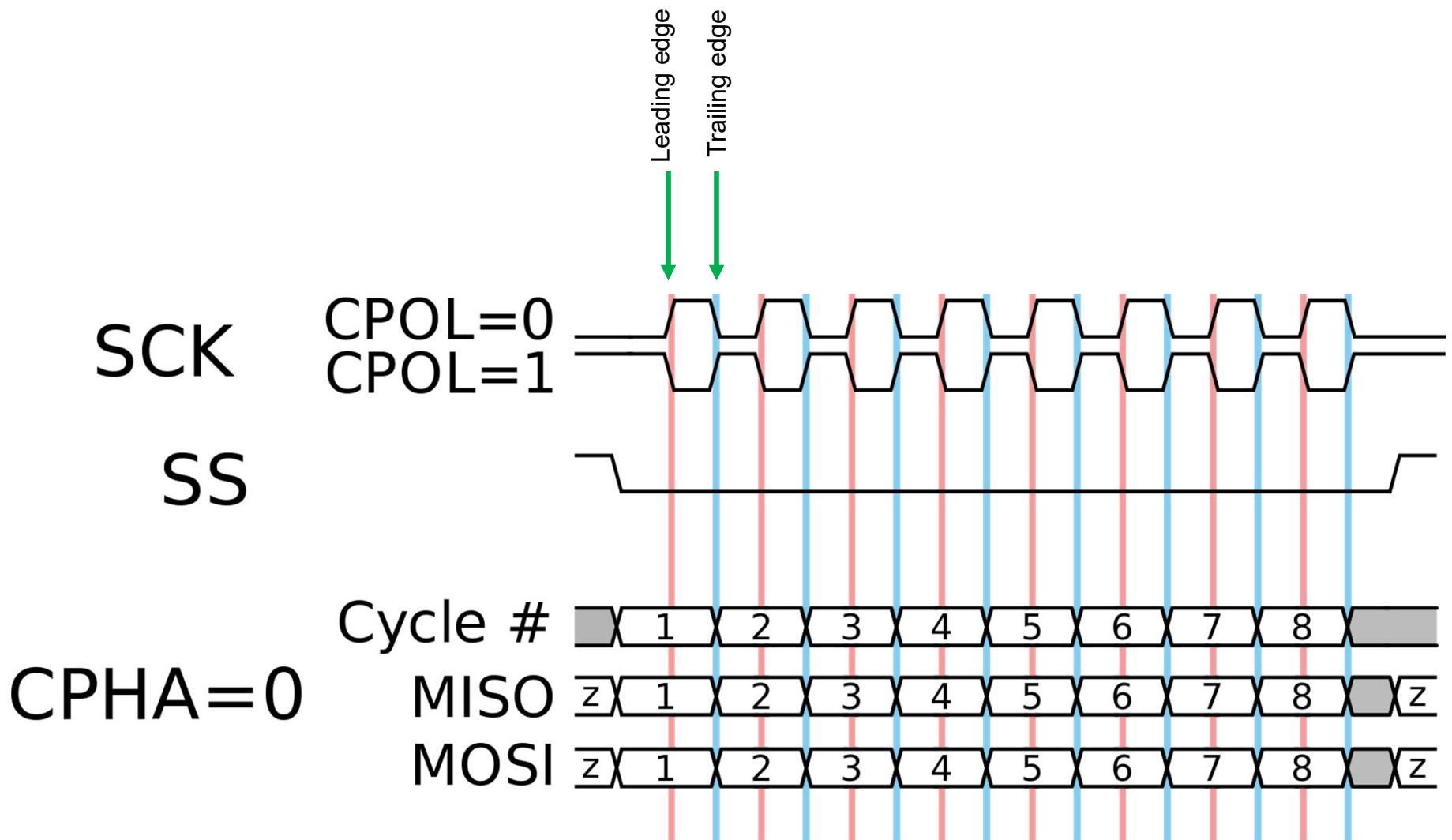
- **Bit 2 – CPHA: Clock Phase**

The settings of the Clock Phase bit (CPHA) determine if data is sampled on the leading (first) or trailing (last) edge of SCK. Refer to [Figure 67](#) and [Figure 68](#) for an example. The CPHA functionality is summarized below:

Table 57. CPHA Functionality

CPHA	Leading Edge	Trailing Edge
0	Sample	Setup
1	Setup	Sample





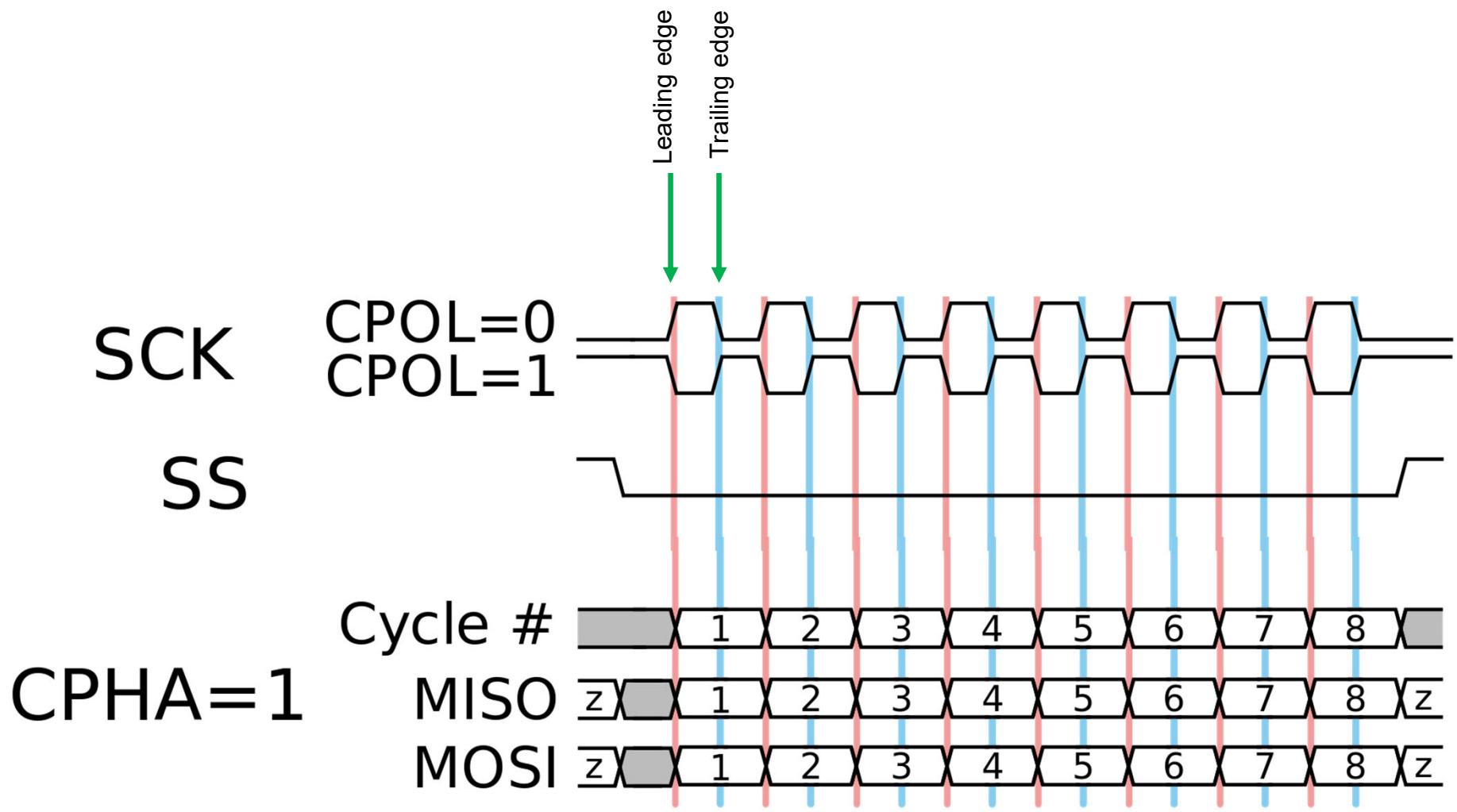


Table 59. CPOL and CPHA Functionality

	Leading Edge	Trailing Edge	SPI Mode
CPOL = 0, CPHA = 0	Sample (Rising)	Setup (Falling)	0
CPOL = 0, CPHA = 1	Setup (Rising)	Sample (Falling)	1
CPOL = 1, CPHA = 0	Sample (Falling)	Setup (Rising)	2
CPOL = 1, CPHA = 1	Setup (Falling)	Sample (Rising)	3

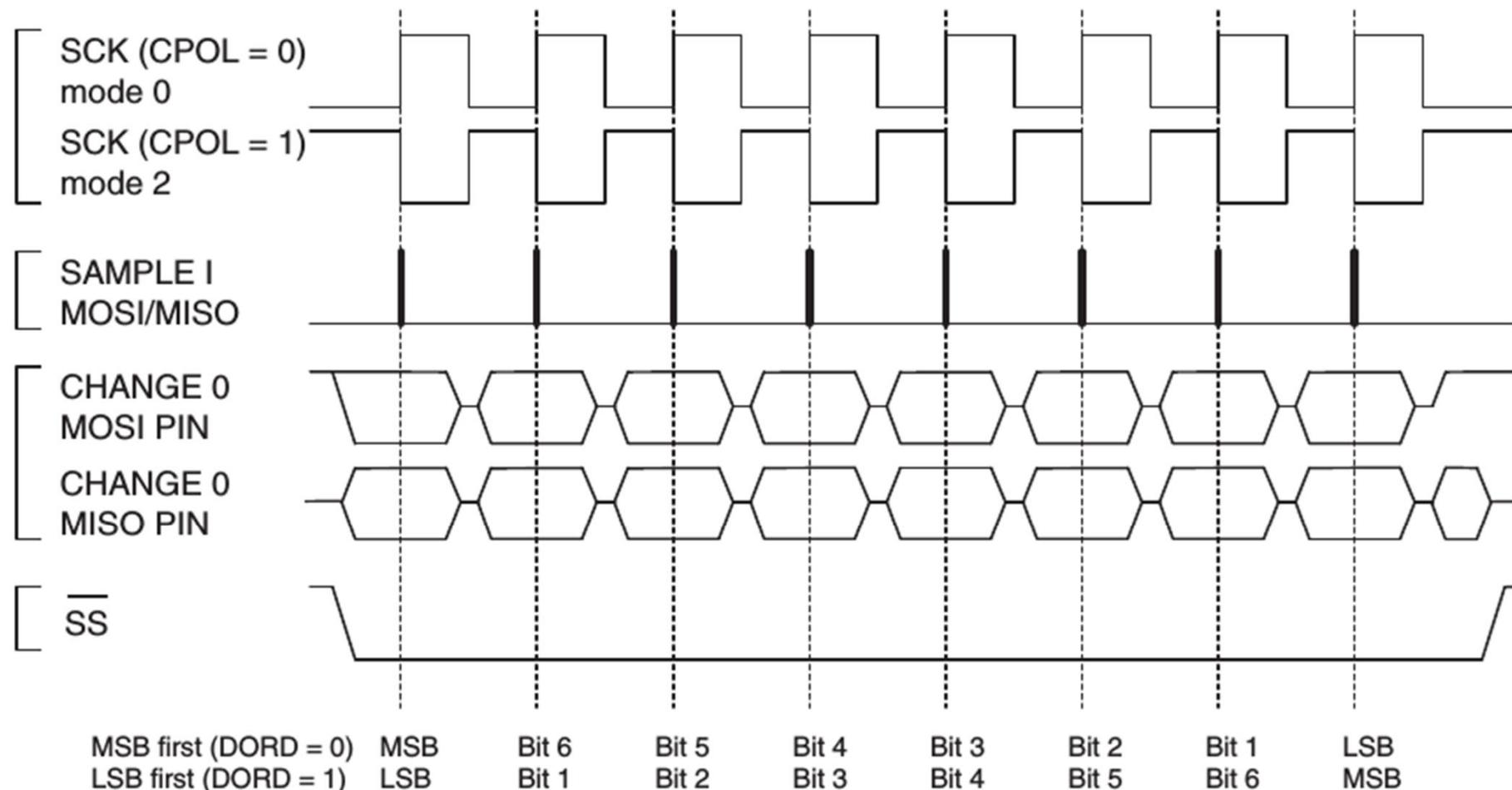
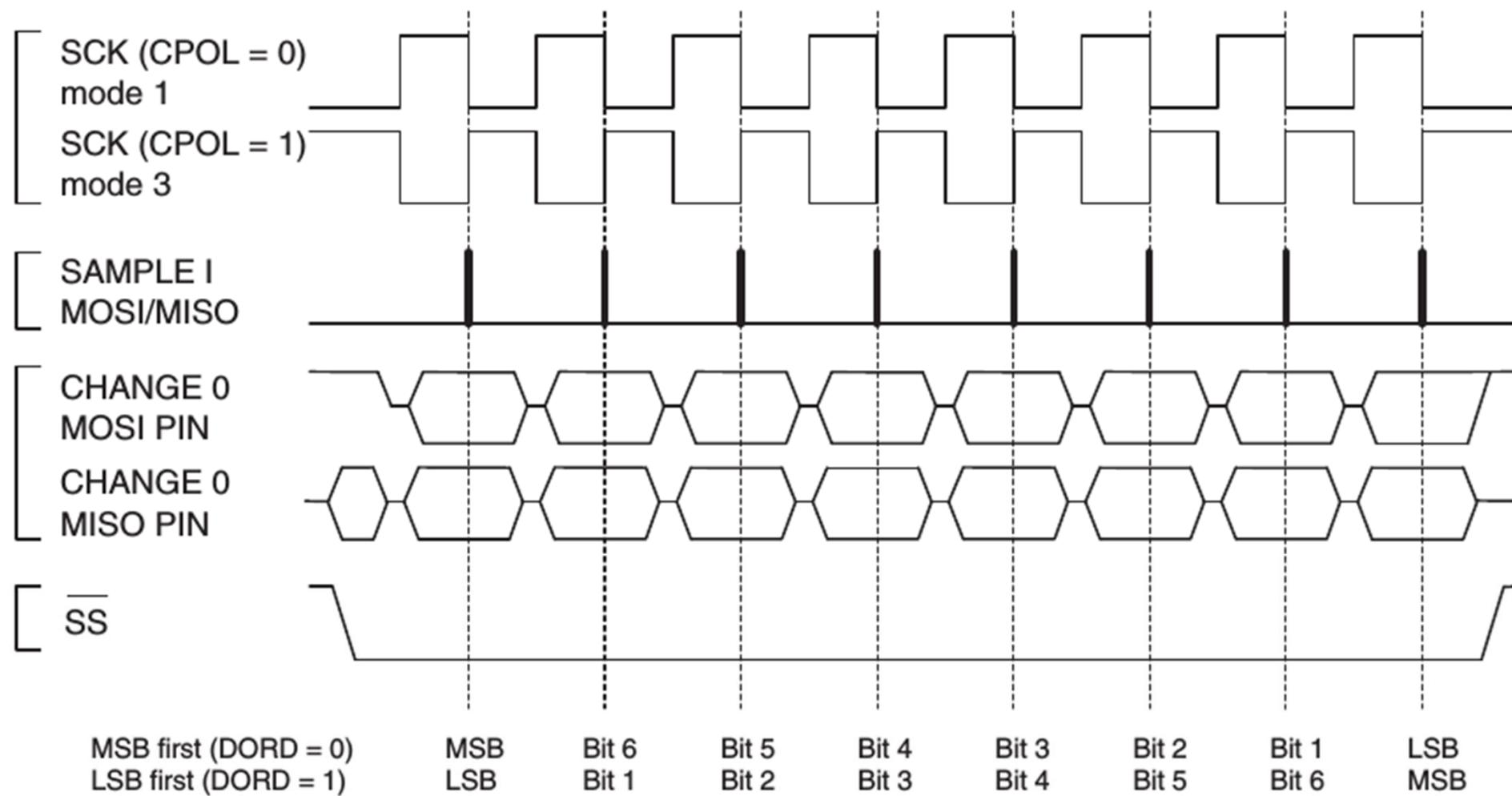
Figure 67. SPI Transfer Format with CPHA = 0

Table 59. CPOL and CPHA Functionality

	Leading Edge	Trailing Edge	SPI Mode
CPOL = 0, CPHA = 0	Sample (Rising)	Setup (Falling)	0
CPOL = 0, CPHA = 1	Setup (Rising)	Sample (Falling)	1
CPOL = 1, CPHA = 0	Sample (Falling)	Setup (Rising)	2
CPOL = 1, CPHA = 1	Setup (Falling)	Sample (Rising)	3

Figure 68. SPI Transfer Format with CPHA = 1

Bit Banging (Not for exams)

- SPI can also be done without using any dedicated hardware of the uC.
- The SPI properties are simulated by programming
- The technique is called Bit Banging



SPI resources

- <http://maxembedded.com/2013/11/the-spi-of-the-avr/>
- <http://maxembedded.com/2013/11/serial-peripheral-interface-spi-basics/>
- <http://www.avrfreaks.net/forum/atmega32-spi-sample-code>
- <http://www.circuitstoday.com/how-to-work-with-spi-in-avr-micro-controllers>
- <http://www.embedds.com/serial-peripheral-interface-in-avr-microcontrollers/>



Bit Banging

- <https://electronics.stackexchange.com/questions/44670/what-is-bit-banging>
- https://en.wikipedia.org/wiki/Bit_banging



