

---

# **Lecture 12**

## **Computer Architecture & Assembly programming**

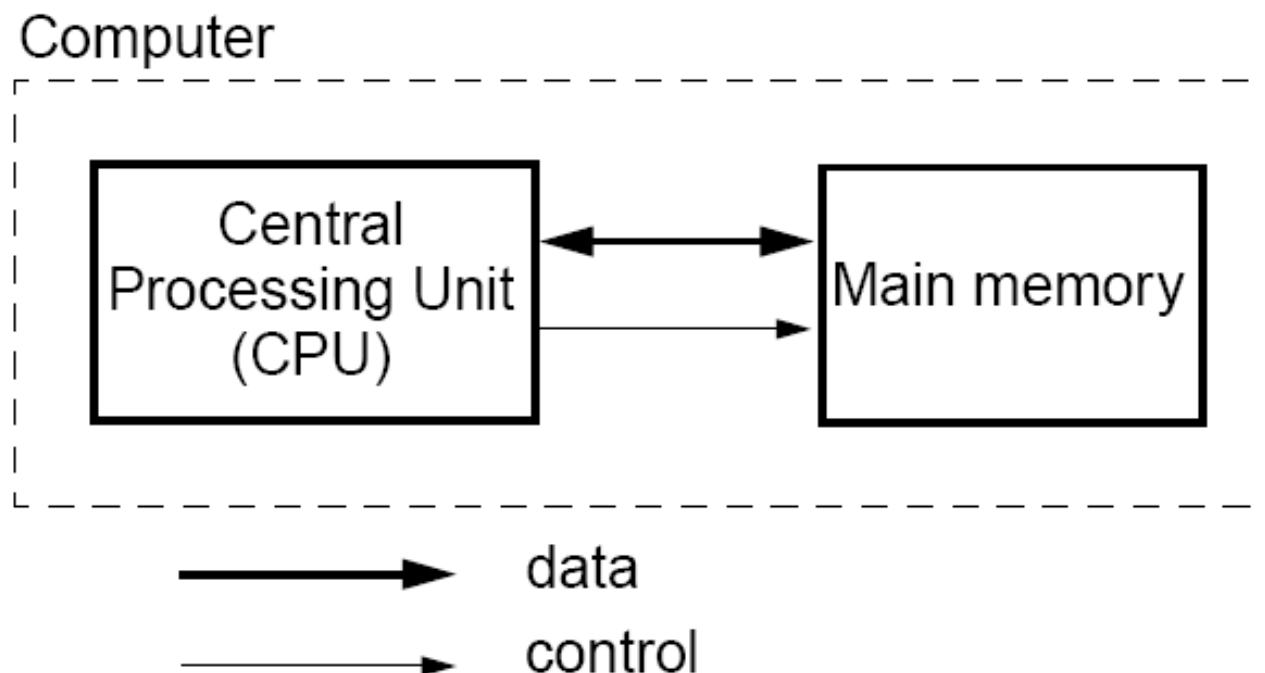
# Learning outcomes

---

- ❑ Describe the basic architecture of a computer.
- ❑ Describe the basic elements of a CPU/microprocessor and their function.
- ❑ Describe the basic CPU operation and instruction cycle.
- ❑ Discuss the different levels of computer programming languages.
- ❑ Write a simple assembly language program.
- ❑ Distinguish between CPUs/microprocessors and microcontrollers.

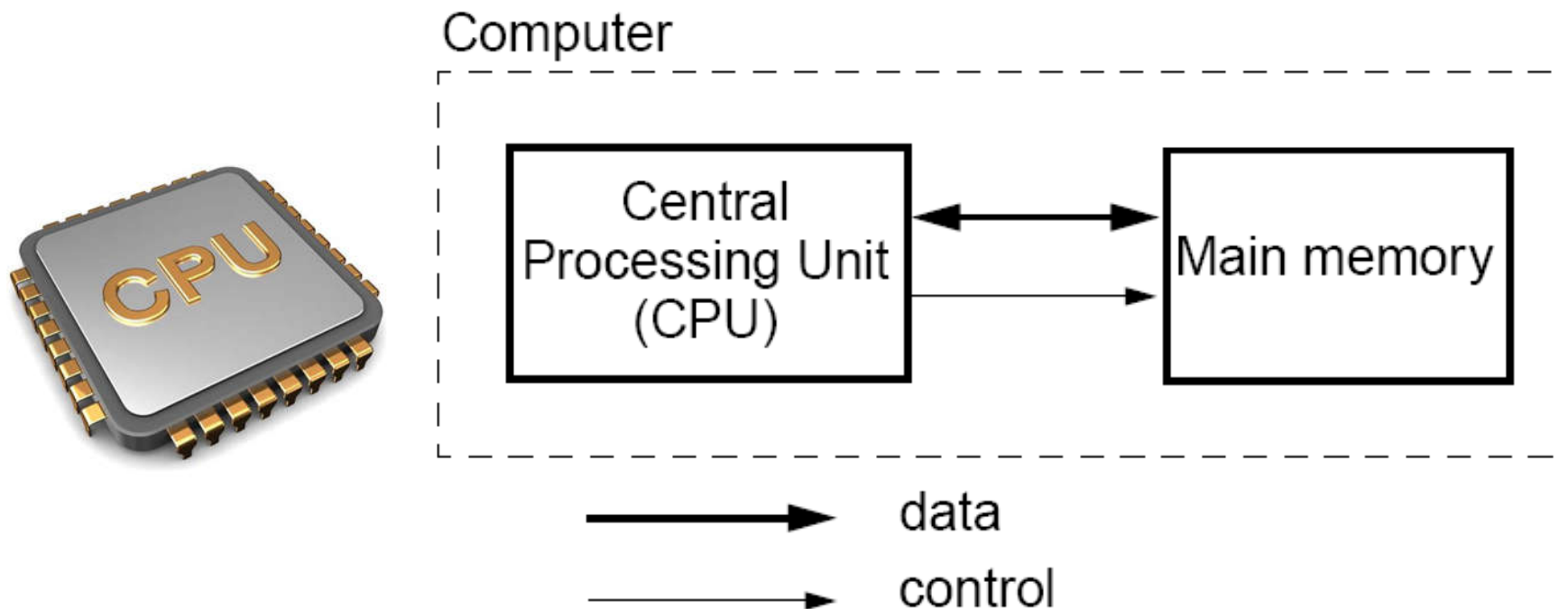
# What is a computer?

- ❑ A computer is a general-purpose data processing machine which is operated automatically under the control of a list of instructions (called a program) stored in its main memory.
- ❑ It can solve very different problems depending on the program they got to execute!



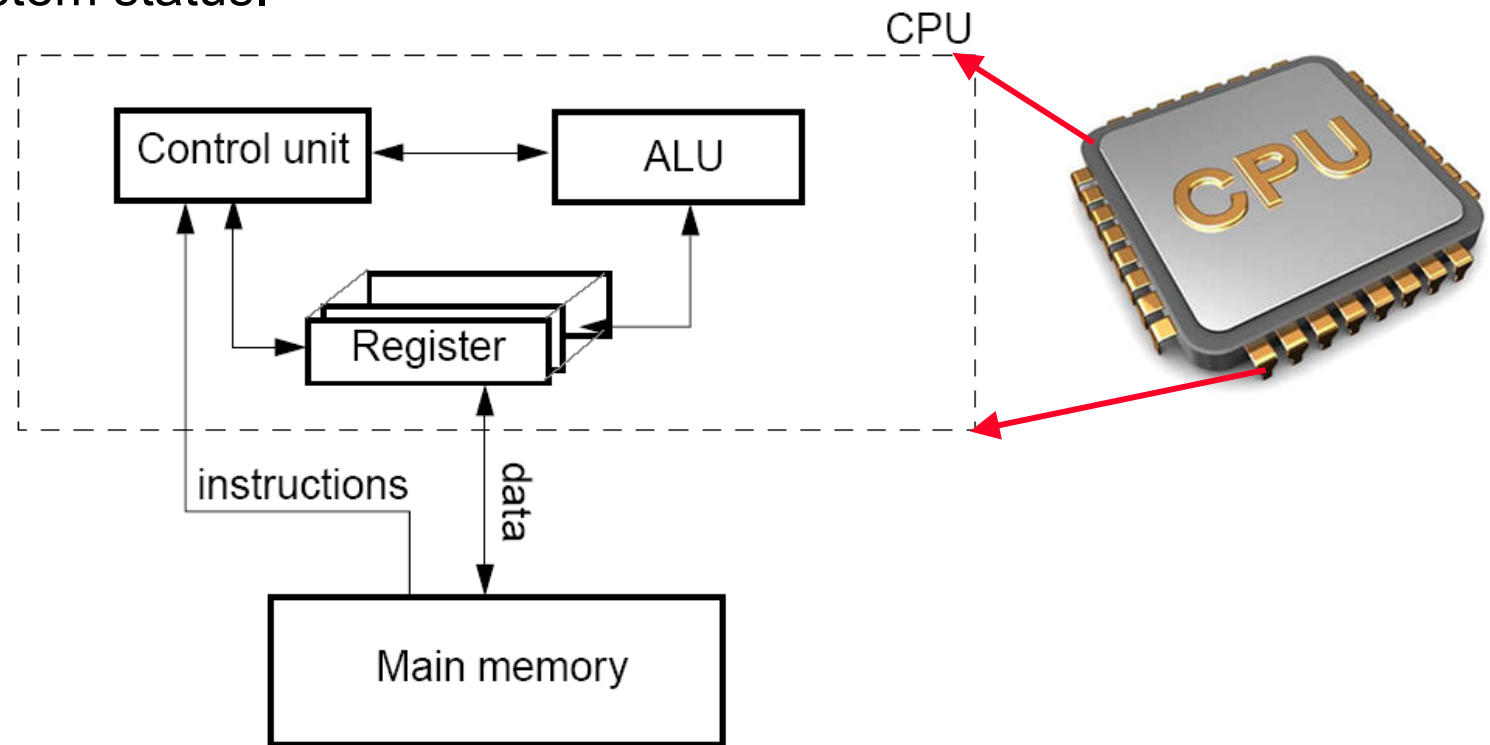
# Central Processing Unit

- ❑ The central processing unit (CPU) is considered the brains of the computer.
- ❑ The CPU handles reading instructions from memory, decoding them to understand which instruction is being performed, and executing the necessary steps to complete the instruction.



# Inside a CPU

- ❑ The Control unit is the one which interprets (decodes) the instruction to be executed and which "tells" the different other components what to do.
- ❑ The Arithmetic Logic Unit (ALU) is the system that performs all mathematical (i.e., addition, subtraction, multiplication, and division) and logic operations (i.e., and, or, not, shifts, etc.).
- ❑ The ALU operates on data being held in CPU registers.
- ❑ The CPU includes a set of registers which are temporary storage devices typically used to hold intensively used data and intermediate results as well as data on system status.



- ❑ The CPU has registers (e.g. memory locations) used for specific purposes:
  - **Instruction Register (IR)** – The instruction register holds the current binary code of the instruction being executed. This code is read from program memory as the first part of instruction execution.
  - **Memory Address Register (MAR)** – The MAR stores the memory location for data or instructions that needs to be fetched from memory or stored into memory.
  - **Program Counter (PC)** – The program counter holds the address of the current instruction being executed in program memory. The program counter will increment sequentially through the program memory reading instructions until a dedicated instruction is used to set it to a new location.
  - **General-Purpose Registers** – These registers are available for temporary storage by the program. Instructions exist to move information from memory into these registers and to move information from these registers into memory. Instructions also exist to perform arithmetic and logic operations on the information held in these registers.
  - **Condition Code Register (CCR) or Status Register** – Holds status flags that provide information about the arithmetic and logic operations performed in the CPU. The most common flags are negative (N), zero (Z), two's complement overflow (V), and carry (C).

# CPUs are not just in computers!

---

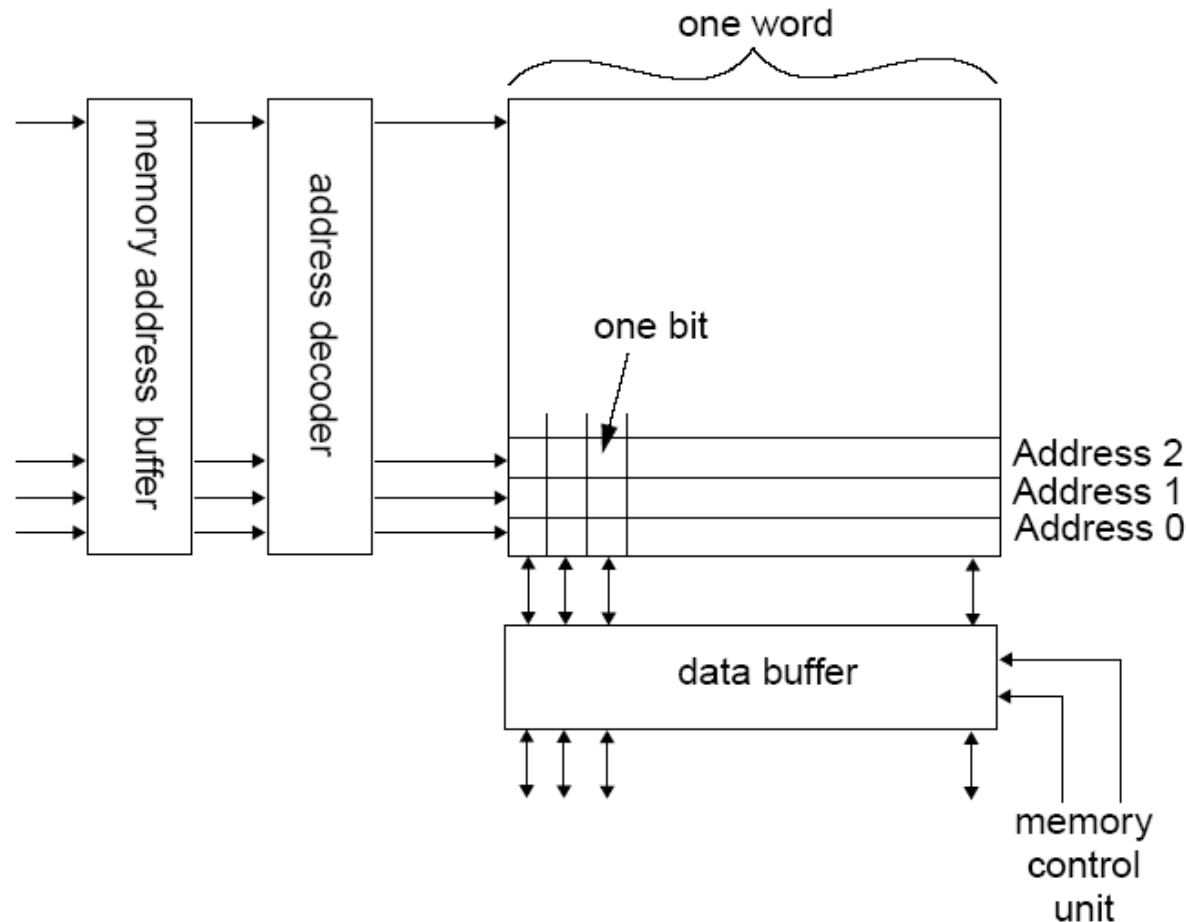
7

- ❑ **Most sophisticated digital** systems available today. Found in a wide range of products
- ❑ **General-purpose and programmable**, found in a wide range of products.
- ❑ **Microprocessor**: CPU on a single chip, with or w/o memory and I/O.
- ❑ **Microcontroller**: CPU, memory and I/O on a single chip, controls the operation of a system



# The Main memory

- ❑ The main memory is used to store the program and data which are currently manipulated by the CPU.



- ❑ Can be viewed as a set of storage cells, each of which is assigned a unique address.



# Data Representation

---

- ❑ Inside a computer, data and control information (instructions) are all represented in binary format which uses only two basic symbols: "0" and "1".
- ❑ The two basic symbols are represented by electrical signals.
- ❑ CPUs are organized to process information in groups of bits, called Words. Word sizes typically range from 8 bits (1 byte) to 64 bits ( 8bytes).
- ❑ The larger the word size, the more the CPU can do.

# Registers

---

- ❑ Most of the activity in a microprocessor involves the transfer of a byte of data from one register to another.
- ❑ Even arithmetic and logical operations such as add, subtract, and, or are performed while transferring data from one register to another.
- ❑ The transfer operation from one register X to another Y is represented by:

$X \leftarrow Y$  or more often as  $Y \leftarrow X$

- ❑ X is called the Source Register, Y the Destination Register. Note that the data in the source register is not affected. A transfer corresponds to a **copy operation**.

# Machine instructions

- ❑ Each computer has a set of specific machine instructions which its CPU is able to recognize and execute.
- ❑ A machine instruction is represented as a sequence of bits (binary digits). For example, below is a 8-bit instruction word divided into 3 fields: a 4-bit Operation Code to uniquely specifies one of  $2^4 = 16$  possible operations; a 2-bit Source field, which specifies one of 4 possible registers: as the source of the data; a 2 bit Destination field. Assume codes (00, 01, 10, 11) for registers (A, B, C, D)



- ❑ These bits define:
  - What has to be done (the operation code)
  - To whom the operation applies (source operands)
  - Where does the result go (destination operand)
  - How to continue after the operation is finished.

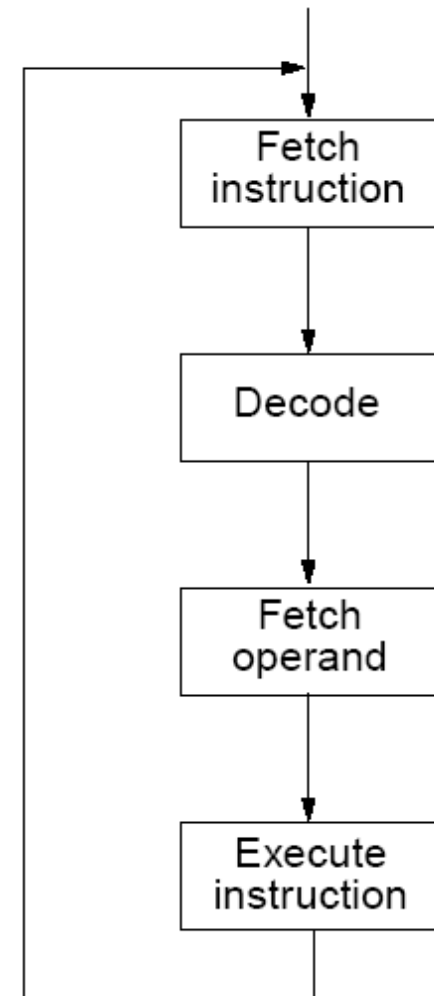
## Machine instructions (4 types)

---

- ❑ Data transfer between memory and CPU registers
- ❑ Arithmetic and logic operations
- ❑ Program control (test and branch)
- ❑ I/O transfer

# The Instruction Cycle

- ❑ The sequence of operations required to execute an instruction are:
  - Instruction Fetch: retrieve the instruction word from memory and transfer it to the instruction register.
  - The instruction decoding logic generates the necessary control signals to Execute the instruction.
  - This sequence, called a Fetch-Execute cycle, is repeated for every instruction.
- ❑ An operand is additional information for the instruction that may be required.
- ❑ An instruction may have any number of operands including zero.



# Opcodes and Operands

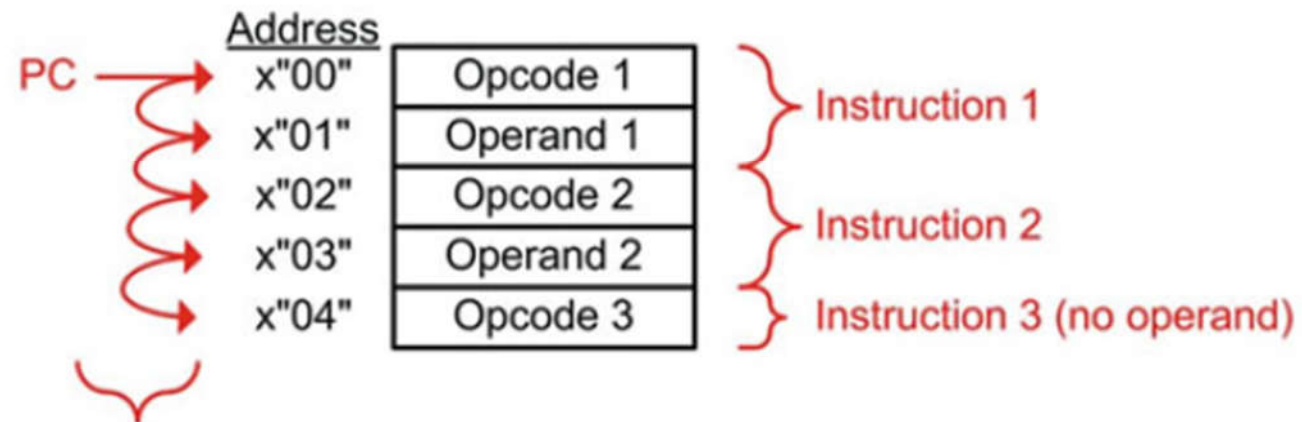
An instruction consists of a unique opcode and potentially one or more operands.

Opcode, Operand

Each instruction in the set is given a unique code.

An operand (optional) provides additional information needed for the instruction.

The following is an example of how instructions may reside in program memory. Each opcode is decoded to know which instruction is to be executed. The opcode additionally tells the CPU whether or not there are operands required in the instruction.



The program counter contains the address of where to read the instruction from. Each time a part of an instruction is read, it is incremented to point to the next location in memory.

# Data Access Methods

---

- ❑ An addressing mode describes the way in which the operand of an instruction is accessed. For example:
  - data could be stored in the instruction itself
  - data could be stored in a register
  - address of data could be stored in instruction
  - address of data could be stored in a register
  
- ❑ Modern computer systems provide numerous addressing modes.

# Instruction set

---

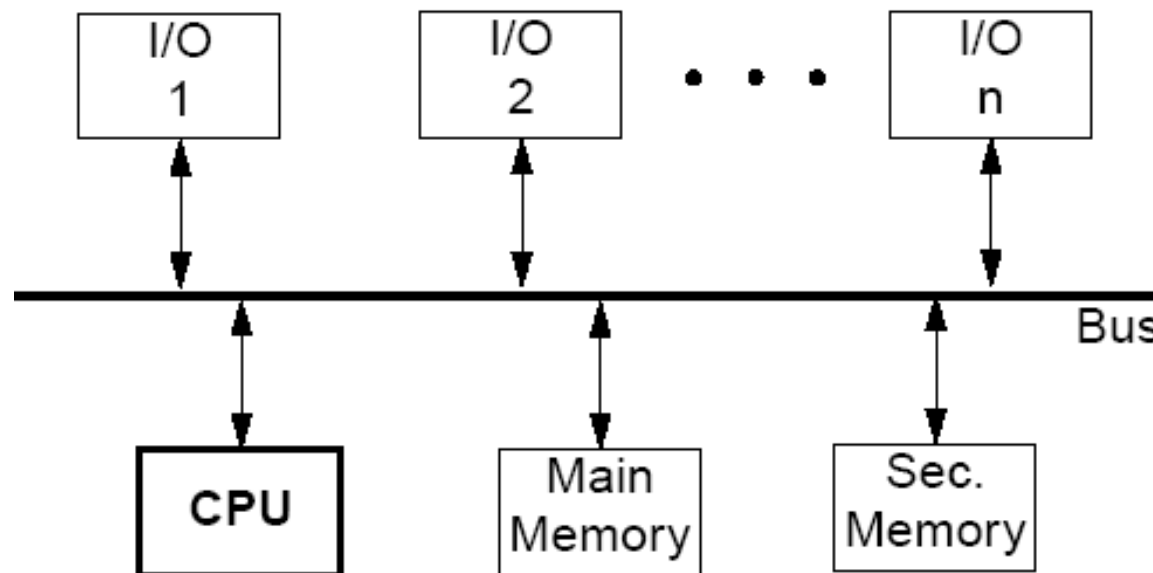
- ❑ The specific group of instructions that a computer can execute is known as its instruction set.
- ❑ Some computer systems have **a very small number of instructions** in order to reduce the physical size of the circuitry needed in the CPU. This allows the CPU to execute the instructions very quickly but requires a large number of operations to accomplish a given task. This architectural approach is called **a reduced instruction set computer (RISC)**.
- ❑ The alternative to this approach is to make an instruction set with **a large number of dedicated instructions** that can accomplish a given task in fewer CPU operations. The drawback of this approach is that the physical size of the CPU must be larger in order to accommodate the various instructions. **This architectural approach is called a complex instruction set computer (CISC)**.



# The Computer System

---

- ❑ **A computer system** consists usually of a computer and its peripherals.
- ❑ **Computer peripherals** include input devices, output devices, and secondary memories.
- ❑ **CPU + main memory** constitute the "core" of the computer system.
- ❑ Communication between different components of the system is usually performed using **one or several buses**.



# Buses

---

- ❑ A **bus** is a collection of wires that carry data.
- ❑ The **Address Bus** is the means by which a processor specifies the location from which data are to be read or to which data are to be written. The address bus is a “one-way street” or unidirectional. If the address bus is 32 bits wide,  $2^{32}$  or 4,294,967,296 memory locations can be accessed.
- ❑ The **Data Bus** consists of signal lines over which the computer system transfers information from one device to another. Because the processor can both read data from and write data to system devices, each data line is bidirectional. Modern processors have 64-bit data buses.
- ❑ The **Control Bus** is the collection of signals that controls the transfer of data within the system and coordinates the operation of system hardware. Control signals can be unidirectional or bidirectional, can function individually or with other control signals.

# Input-output Devices

---

- ❑ Input and output devices provide a means for people to make use of a computer.
- ❑ Typical input devices include keyboard, light pen, Mouse, Joystick, Graphics tablet, Voice input, Scanner.
- ❑ Typical output devices: Display screen, Laser/Inkjet printer, voice output.
- ❑ Some I/O devices function also as an interface between a computer system and other physical systems. Such interface usually consists of A/D and D/A converters.

# Secondary memory

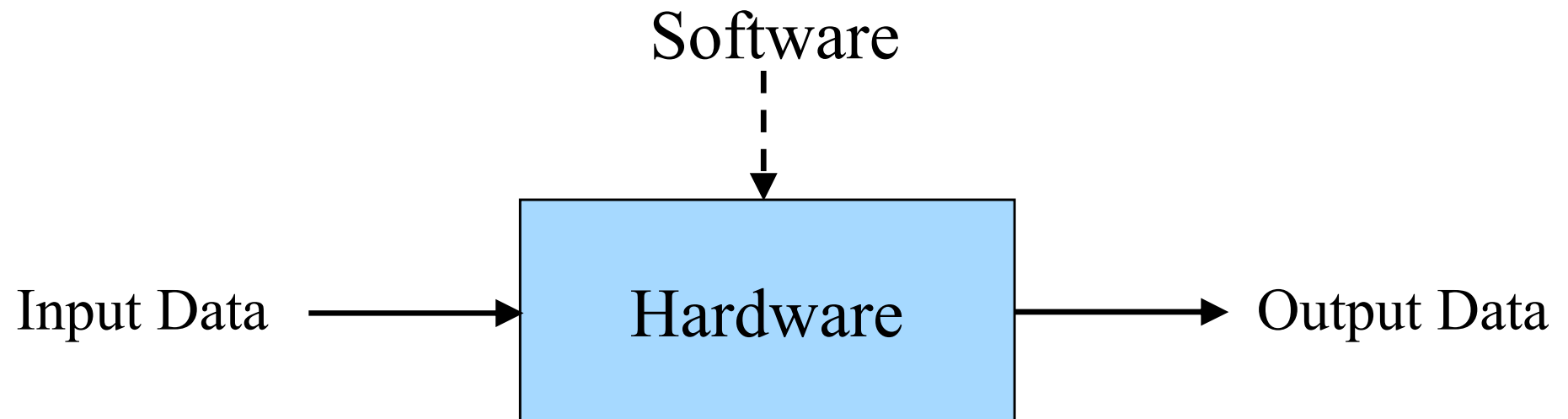
---

- ❑ The secondary memory provides the long term storage of large amounts of data and program.
- ❑ Hard disk, floppy disk, CD-ROM, DVD-ROM, are all examples of secondary memories.
- ❑ Before the data and program can be manipulated by the CPU, they must first be loaded into the main memory.
- ❑ Because of the technology used for its implementation, the secondary memory is significantly slower and larger than the main memory.

---

# **Assembly Language Programming**

- ❑ Computers must be programmed to perform some specific function or task.
- ❑ Load program on hardware for a particular task



# Programming a computer system

---

## ❑ High level programming language (C, C++, ...)

- Easier as operations described in a semi-natural language
- Interpreter and compiler convert it in machine code; reusable/portable code
- Not optimized to the underlying hardware

## ❑ Machine code

- Direct execution and optimized for the underlying hardware
- Very low-level programming using a sequence of 1's and 0's.
- Difficult and error-prone, not used in practice anymore

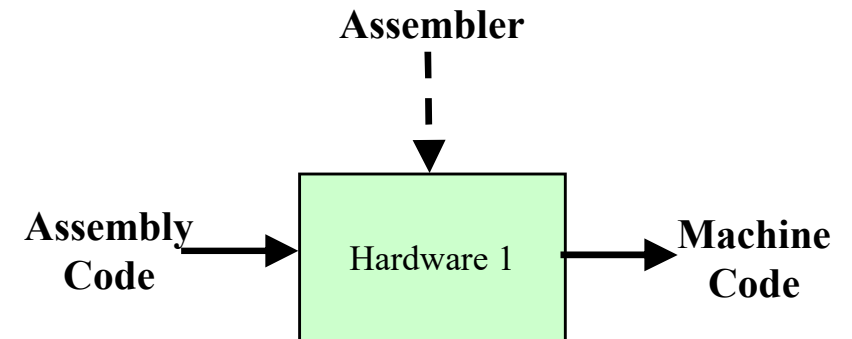
## ❑ Assembly language

- use a mnemonic (i.e. abbreviation/nickname ADD for adding, CMP for comparing) to represent the instruction. Much easier to write mnemonics than the corresponding instruction bytes. The mnemonics are then converted to the corresponding instruction bytes by looking them up in a table. A program that does this is called an **assembler**.
- Difficult and error-prone
- Only used when code needs to run very quickly and efficiently, e.g. embedded systems, network processors, etc...
- Instruction format is machine-dependent, cannot be ported!

# Assembler Example

```
main:  LDI    R16, 7
        LDI    R17, 'A'
        ADD    R16, R17
        SUBI   R16, 0x40
        LDI    R18, 3
        MUL    R18, R16
        LSR    R18
        NEG    R18
        STS    result, R18
        JMP    main
```

myfile.asm



Hex Code (machine code)

```
:0200000020000FC
:1000000007E011E4010F005423E0209F269521957D
:08001000209300010C94000094
:000000001FF
```

myfile.hex

**The Hex representation makes it more readable, easier to write and less error prone, but still far from convenient for program writing.**



# Hexadecimal Number System

- ❑ Hexadecimal uses sixteen characters to represent numbers: the numbers 0 through 9 and the alphabetic characters A through F.
- ❑ Large binary number can easily be converted to hexadecimal by grouping bits 4 at a time and writing the equivalent hexadecimal character.
- ❑ Express **1001 0110 0000 1110<sub>2</sub>** in hexadecimal:
- ❑ Group the binary number by 4-bits starting from the right. Thus, **960E**

Decimal	Hexadecimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

# Hexadecimal Number System

- Hexadecimal is a weighted number system. The column weights are powers of 16, which increase from right to left.

Column weights  $\begin{cases} 16^3 & 16^2 & 16^1 & 16^0 \\ 4096 & 256 & 16 & 1 \end{cases}$

- Express  $1A2F_{16}$  in decimal:
  - Start by writing the column weights:

4096   256   16   1  
1   A   2    $F_{16}$

$$1(4096) + 10(256) + 2(16) + 15(1) = 6703_{10}$$

Decimal	Hexadecimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

## ❑ Arduino Nano 3.0

- Small, complete, and breadboard-friendly board
- Microcontroller Atmel ATmega328P
- Clock Speed 16 MHz, operating Voltage (logic level) 5 V
- 1KB EEPROM, 32KB Flash Memory, 2KB SRAM
- 14 digital pins can be used as an input or output
- instruction set [manual](#)



# Atmel Registers

---

- ❑ 32 General purpose registers, 8-bit, **R0 .. R31**
- ❑ 3 index registers, 16-bit, **X, Y, Z**  
but using R26..R31
- ❑ Status register **SREG**
- ❑ 64 Special registers for control and I/O, **GPIO0, GPIO1, SMCR, SP, ...**
- ❑ Further extended I/O space

# Atmel General Purpose Registers

	Addr.		
	7	0	
General Purpose Working Registers	R0	0x00	
	R1	0x01	
	R2	0x02	
	...		
	R13	0x0D	
	R14	0x0E	
	R15	0x0F	
	R16	0x10	
	R17	0x11	
	...		
	R26	0x1A	X-register Low Byte
	R27	0x1B	X-register High Byte
	R28	0x1C	Y-register Low Byte
	R29	0x1D	Y-register High Byte
	R30	0x1E	Z-register Low Byte
	R31	0x1F	Z-register High Byte

15	XH		XL	0
7		0	7	0
R27 (0x1B)			R26 (0x1A)	

15	YH		YL	0
7		0	7	0
R29 (0x1D)			R28 (0x1C)	

15	ZH		ZL	0
7		0	7	0
R31 (0x1F)			R30 (0x1E)	

**Note:** Not all instructions work with all registers ....

Label <i>(opt.)</i>	Command	Operands	Comment <i>(opt.)</i>
main:	ADD	R17, R18	; start

**Label:** Needed for branching / jumps

**Command:** Op-code for processor

**Operands:** The arguments the command operates on  
(e.g. constant, register, memory)

**Comments:** Should be meaningful and describe what the process  
accomplishes, rather than restating the command

## ❑ Arithmetic

ADD, SUB, MUL

*no division*

## ❑ Data Movement

LD, ST, MOV  
LDS, STS, IN, OUT

*move data between registers and memory*

## ❑ Logic and Shift

AND, OR, COM, EOR  
LSL, LSR, ROL, ROR

*and, or, not, xor*

*shift left/right, rotate left/right*

## ❑ Compare and Branch

CP, TST  
JMP, BREQ, BRLT

*compare*

*jump always, branch if-equal/if-less-than*

# Atmel Operands

---

## ❑ Each operand can be:

### ● **Constants** (immediate)

Constants are operands whose value does not change during program execution.

10                      Decimal constant

0xD7                  Hexadecimal constant

0b0110              Binary constant

'Z'                  Character const.

```
LDI R16, 0x0F ; R16:=15
```

### ● **Registers**

R0..R31 general purpose

```
MOV R1, R2 ; R1:=R2
```

### ● **Memory** values (*variables*)

usually specified by labels

```
LDS R5, name ; R5:=name
```



# Atmel Assembly Commands (1)

**Note:** With 2 operands, *result* goes to *first* operand

## ❑ Arithmetic

ADD	add registers
ADIW	add imm. word
SUB	subtr. registers
SUBI	subtr. immediate
MUL	multiply unsigned
MULS	multiply signed
--	<i>no division</i>
NEG	negate value
CLR	clear (set to 0)
INC	increment
DEC	decrement

## ❑ Logic

AND	and registers
ANDI	and immediate
OR	or registers
ORI	or immediate
EOR	exclusive-or reg.
COM	not (complement)
LSL	logical shift left
LSR	logical shift right
ROL	rotate left (carry)
ROR	rotate right (carry)

## ❑ Subroutine

CALL	call subroutine
RET	return

# Atmel Assembly Commands

- Please see instruction set [manual](#)

## ADD – Add without Carry

### Description:

Adds two registers without the C Flag and places the result in the destination register Rd.

### Operation:

(i)  $Rd \leftarrow Rd + Rr$

### Syntax:

(i) ADD Rd,Rr

### Operands:

$0 \leq d \leq 31, 0 \leq r \leq 31$

### Program Counter:

$PC \leftarrow PC + 1$

### 16-bit Opcode:

0000	11rd	dddd	rrrr
------	------	------	------

### Status Register (SREG) and Boolean Formula:

I	T	H	S	V	N	Z	C
–	–	$\Leftrightarrow$	$\Leftrightarrow$	$\Leftrightarrow$	$\Leftrightarrow$	$\Leftrightarrow$	$\Leftrightarrow$

# Atmel Assembly Examples

Operation	Example	Result
load immed.	LDI R16, 100	R16 := 100 *
add	ADD R1, R2	R1 := R1 + R2
sub	SUB R1, R2	R1 := R1 - R2
sub immed.	SUBI R16, 4	R16 := R16 - 4 *
negation	NEG R0	R0 := -R0
increment	INC R0	R0 := R0 + 1
decrement	DEC R0	R0 := R0 - 1
and	AND R5, R6	R5 := R5 AND R6
or	ORI R16, 7	R16 := R16 OR 7 *
not	COM R4	R4 := NOT R4
move	MOV R5, R1	R5 := R1
exclusive-or	EOR R5, R6	R5 := R5 XOR R6
log. shift left	LSL R0	1111 1010 becomes: 1111 0100
log. shift right	LSR R0	1111 1010 becomes: 0111 1101

*\* Immediate commands only for R16..R31*

# Logical Instructions

---

- ❑ The instruction set includes a number of instructions that perform bitwise logical operations such as:
  - **AND Masking:** selectively **clear specific bits** of a data word. This is done by ANDing the data word with another word, called a **mask, which contains 0s in the bit positions to be cleared** and 1s everywhere else.
  - **OR Masking:** selectively **set specific bits** of a data word. This is done by ORing the data word with another word, called a **mask, which contains 1s in the bit positions to be cleared** and 0s everywhere else.
  - **EOR Masking:** selectively **invert specific bits** of a data word. This is done by **ExclusiveORing** the data word with another word, called a **mask, which contains 1s at the positions of the bits to be inverted**. The complement instruction can also be used to achieve this.

# AND masking

---

- ❑ Mask out bits 7, 6, 5 and 4, leave other bits unaffected.

D <sub>7</sub>	D <sub>6</sub>	D <sub>5</sub>	D <sub>4</sub>	D <sub>3</sub>	D <sub>2</sub>	D <sub>1</sub>	D <sub>0</sub>
----------------	----------------	----------------	----------------	----------------	----------------	----------------	----------------

Data byte

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

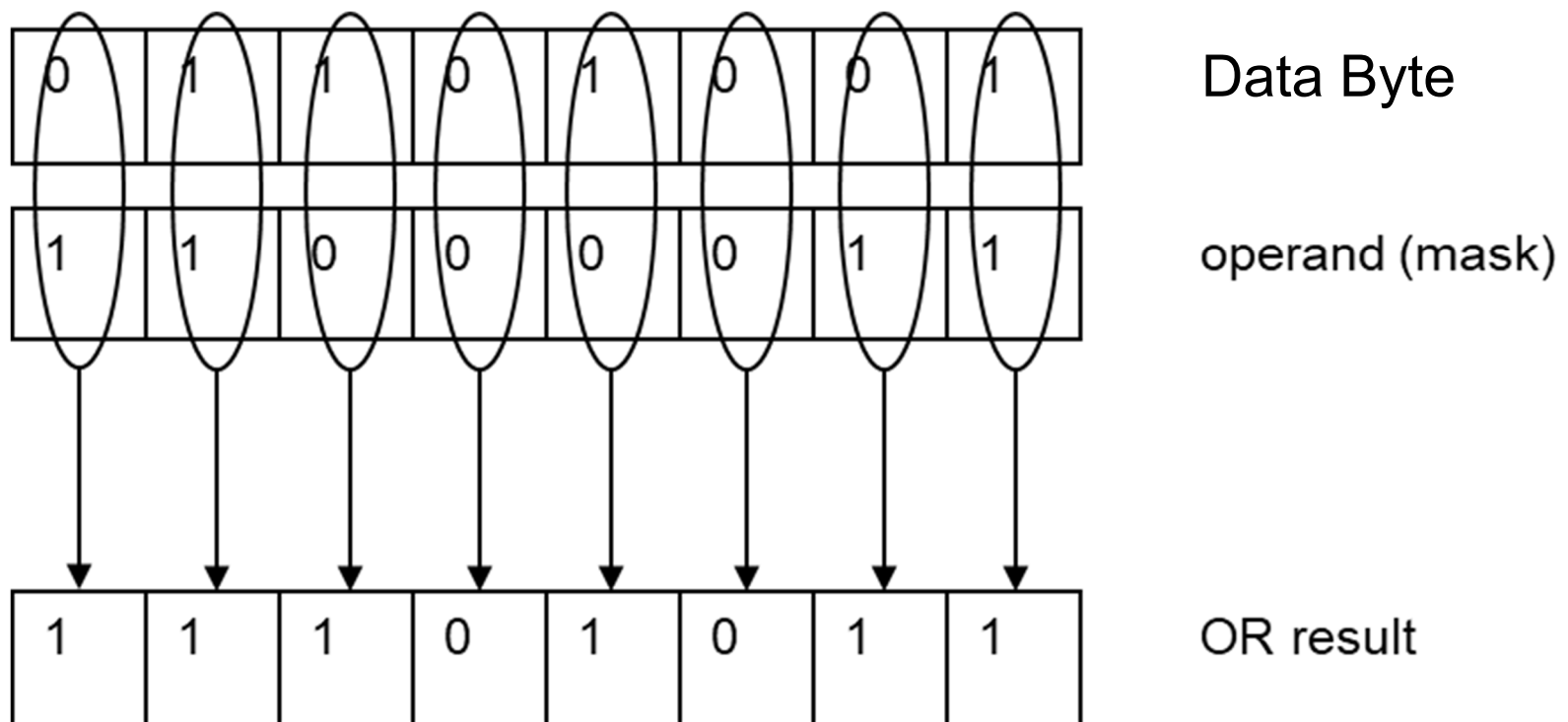
Mask (\$0F)

0	0	0	0	D <sub>3</sub>	D <sub>2</sub>	D <sub>1</sub>	D <sub>0</sub>
---	---	---	---	----------------	----------------	----------------	----------------

AND result

# OR masking

- Mask out bits 7, 6, 1 and 0, leave other bits unaffected.



# Atmel Jump and Branch

- ❑ After arithmetic or logic commands the **program counter** (PC) is always incremented to the next instruction
- ❑ Jump and branching instructions can be used to change control flow


## Unconditional jump

Change program counter to specified address

```
JMP label
[PC := label]
```

### Example:

```
JMP next ; always jump
...
next: ADD .. ; destinat.
```




## Conditional branch

Program counter is only changed if a condition is true

```
BREQ label
[if equal then PC := label]
```

### Example:

```
CP R1,R2 ; compare reg.
BREQ next ; branch if equal
CLR R7 ; else do this
...
next: ADD ... ; destination
```



# Atmel Conditional Branching

## Compare – Flags – Branching

R1 = 7    R2 = 5

1. Compare

**CP R1, R2**

condition flags in Status R. are **set** →

0	<b>0</b>	0	0	...
N	Z	C	V	...

2. Branch if condition is true

**BREQ** ~~calc~~ ←

calc: ...

flag is **zero**,  
**no branching**

*This is how CPU memorizes the result form the preceding operation*



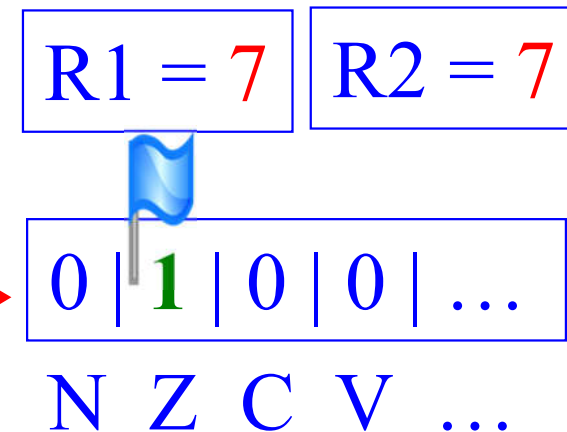
# Atmel Conditional Branching

## Compare – Flags – Branching

1. Compare

**CP R1, R2**

condition flags in Status R. are **set** →



2. Branch if condition is true

**BREQ calc**

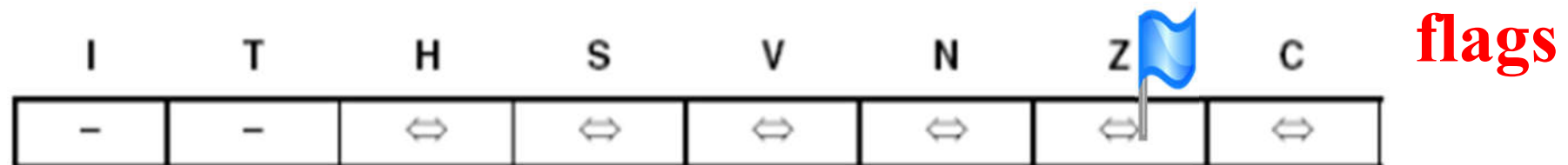
...  
calc: ...

flag is **one**,  
yes, do branching

*This is how CPU memorizes the result form the preceding operation*

# Atmel Conditional Branching

## Status Register



Each assembly command:

- calculates its result (e.g. in a register)
- sets the flags in the status register accordingly (e.g. **Z** if result was zero)

# Atmel Branch Commands

- ❑ Branching can be made on several different **conditions**
- ❑ **Conditions** are the result of the previous arithmetic or logic operation and are stored in the condition code register
- ❑ Most important conditions:
  - Z     zero
  - N     negative
  - C     carry (e.g. in addition)
  - V     overflow (e.g. in addition)

- ❑ Most important branches:
  - BREQ   branch on equal (=0)
  - BRNE   branch on not equal
  - BRGE   branch on gr. or equal (sig.)
  - BRLT   branch on less than (sign.)
  - BRPL   branch on plus
  - BRMI   branch on minus
  - BRCS   branch on carry set
  - BRCC   branch on carry clear
  - BRVS   branch on overflow set
  - BRVC   branch on overflow clear
- ❑ if condition then PC:= label  
else PC := PC + inc. (next instr.)

# Atmel Assembly Commands

**Note:** With 2 operands, result goes to **first** operand

## ❑ Compare

CP	compare registers
CPI	compare with imme.
TST	test single register

## ❑ Branching

JMP	jump (always, abs.)
BREQ	branch if equal (=0)
BRNE	branch if not eq.
BRGE	branch if gr.or eq.
BRLT	branch if less th.
BRPL	branch if plus
BRMI	branch if minus
BRCS	branch if carry set
BRCC	branch if carry clear

## ❑ Move Data

LDI	load register immed.
LD	load indirect
LDS	load memory
ST	store indirect
STS	store memory
MOV	move betw. registers
IN	read input data
OUT	write output data

## ❑ Bit Operations

SBR	set bit in register
CBR	clear bit in register
SBI	set bit in I/O register
CBI	clear bit in I/O reg.

# Atmel Conditional Execution

---

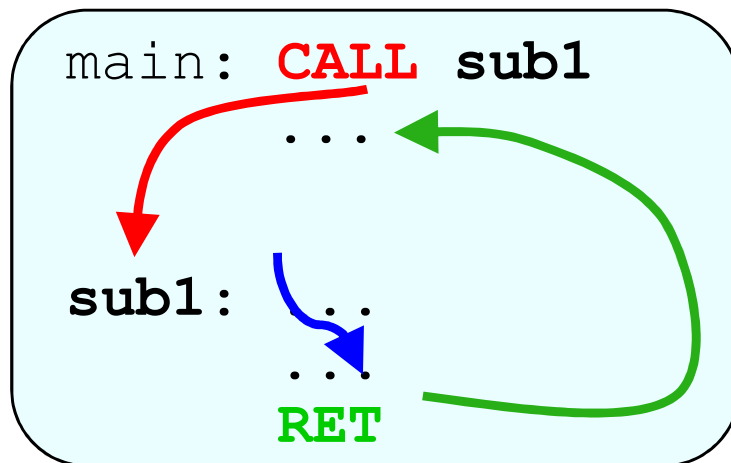
Example: **if** (R16==10) **then** R16:= R16+R1  
**else** R16:= R16–R2  
**end**

Standard way: branching to implement an if-then-else

```
        CPI    R16, 10
        BREQ   then
else:    SUB    R16, R2
        JMP    next
then:    ADD    R16, R1
next:    ...
```

# Atmel Subroutines and Stack

- ❑ Command **CALL** (call subroutine) is similar to **JMP**, but preserves current program counter on stack, so execution can return later using **RET** (return from subroutine)



- ❑ Most subroutines have to pass parameters e.g. like math function:  $z := f(x,y)$
- ❑ Passing parameters in registers is one method

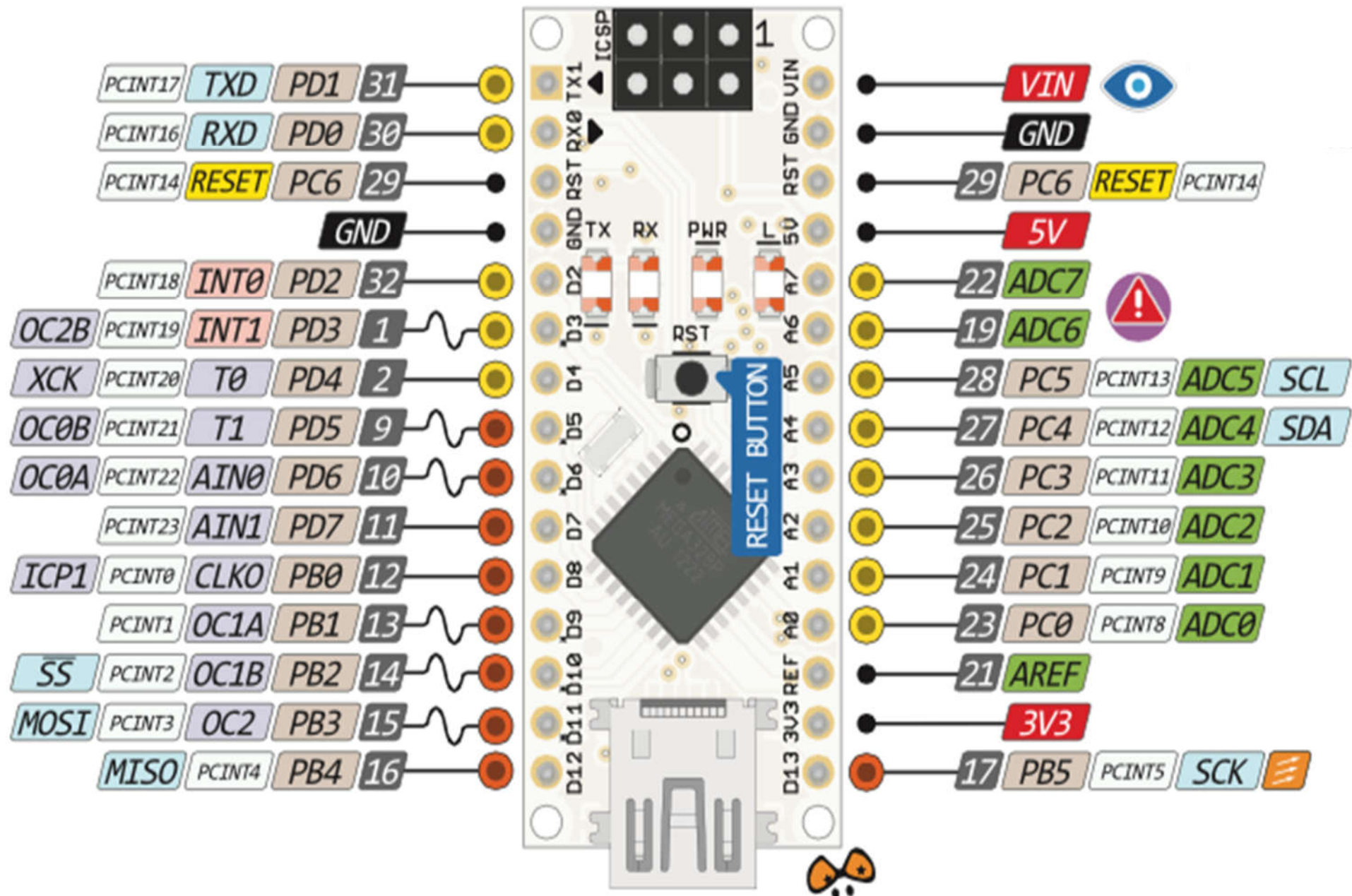
# Atmel Parameter Passing: Registers

---

```
main: LDI R16, 21
      LDI R17, 47
      CALL add
      ; result is now in R0
      ...
```

```
add:  MOV R0, R16
      ADD R0, R17
      RET
```

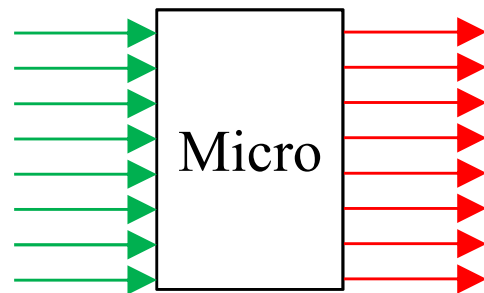
# Atmel Arduino Nano Input/Output



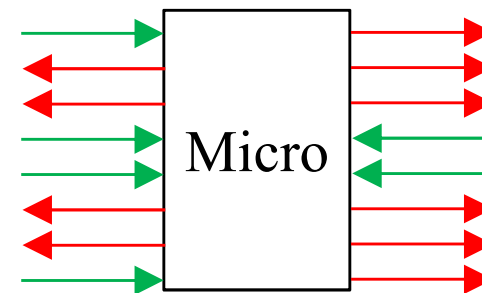


# Ports

- ❑ The term port is used to describe the mechanism to get information from the output world into or out of the computer.
- ❑ Ports can be input, output, or bidirectional. I/O ports can be designed to pass information in a serial or parallel format.
- ❑ Traditional processors
  - Fixed number of inputs and fixed number of outputs
- ❑ Modern processors
  - Each I/O can be configured as input or output line
    - Flexibility
    - Configure before use!



**Traditional processors**



**Modern processors**

# Atmel Ports

---

- ❑ Atmel processor family has I/O lines grouped in **ports** of **8 pins** each
- ❑ The **Nano** uses the **Atmel 328P** chip with accessible **ports B, C and D**
- ❑ All ports pins are **bi-directional**,  
→ specify I/O direction before use through the data-direction register (**DDR**)
- ❑ Read port input using the **PIN** register
- ❑ Set output using the **PORT** register
- ❑ Each port has its own **DDR**, **PIN** and **PORT** register, i.e.:  
DDRB, PINB, PORTB, DDRC, PINC, PORTC, DDRD, PIND, PORTD

# Atmel Ports

## PORTA – Port A Data Register

Bit	7	6	5	4	3	2	1	0	
0x02 (0x22)	PORTA7	PORTA6	PORTA5	PORTA4	PORTA3	PORTA2	PORTA1	PORTA0	PORTA
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	

## DDRA – Port A Data Direction Register

Bit	7	6	5	4	3	2	1	0	
0x01 (0x21)	DDA7	DDA6	DDA5	DDA4	DDA3	DDA2	DDA1	DDA0	DDRA
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	

## PINA – Port A Input Pins Address

Bit	7	6	5	4	3	2	1	0	
0x00 (0x20)	PINA7	PINA6	PINA5	PINA4	PINA3	PINA2	PINA1	PINA0	PINA
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	

# Atmel Ports for Input/Output

---

- ❑ We are using **ports B and D**
- ❑ All ports bits are bi-directional, so before using them the I/O direction has to be specified

```
LDI R16, 0xFF ; make all port D pins output
```

```
OUT DDRD, R16 ; Data Direction Register D
```

```
LDI R16, 0x00 ; make all port B pins input
```

```
OUT DDRB, R16 ; Data Direction Register B
```

# Atmel Ports for Input/Output

---

## ❑ Reading from port B:

```
IN R16, PINB ; read 8 values
```

## ❑ Writing to port D:

```
LDI R16, 0x55
```

```
OUT PORTD, R16 ; write 8 values 0x01010101
```

# Acknowledgments

---

- ❑ Credit is acknowledged where credit is due.  
Please refer to the full list of references.