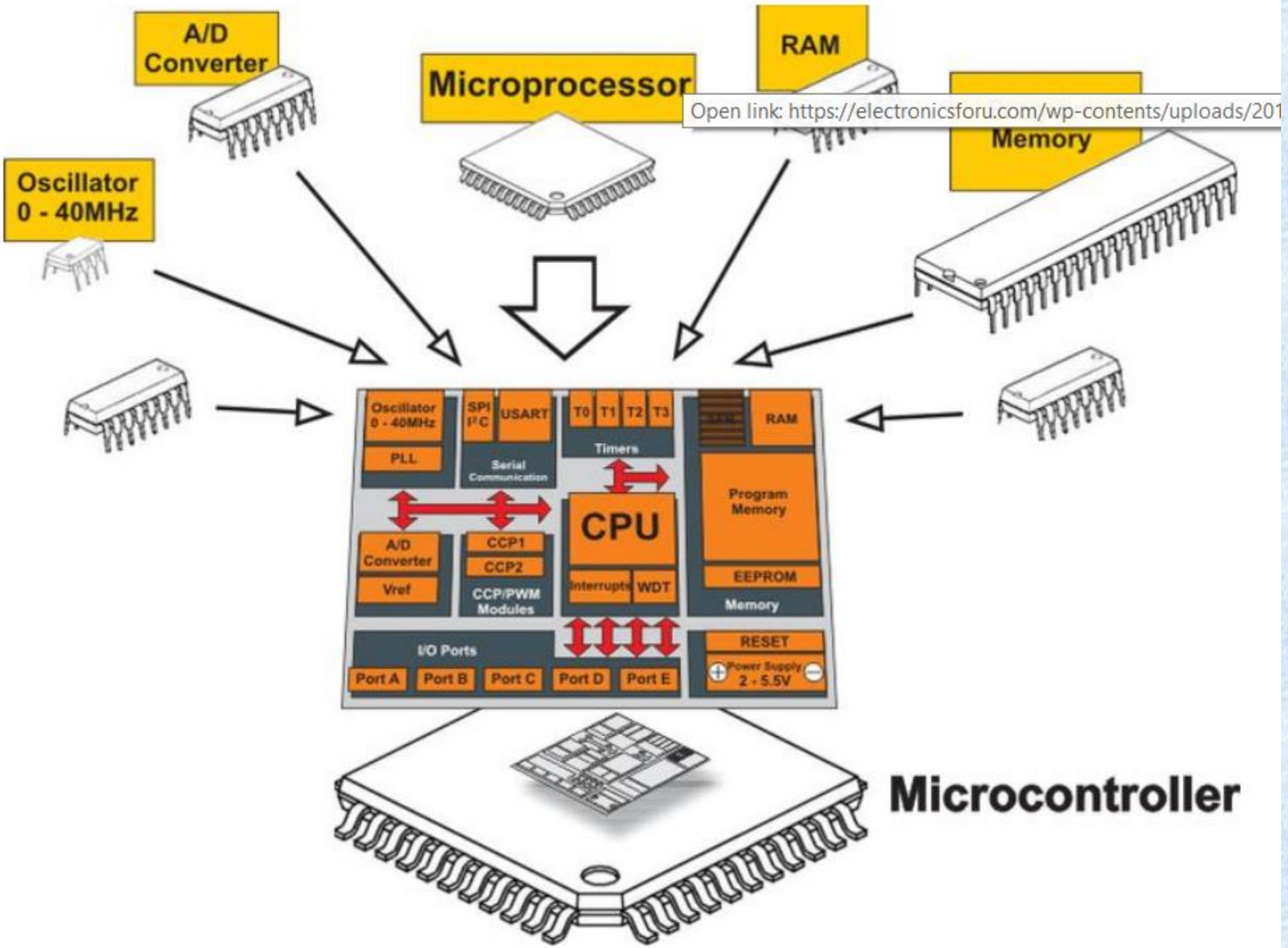
Microcontroller

Fourth Grade

Dr. Mohannad K. Sabir

2022-2023

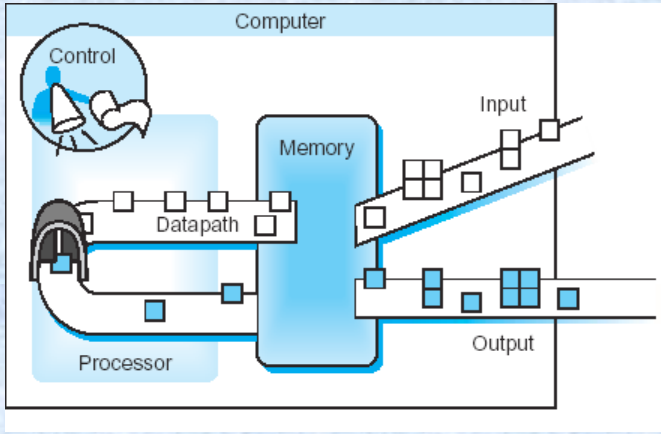

A/D Converter
Microprocessor
RAM
Memory
Oscillator 0 - 40MHz
Open link: https://electronicsforu.com/wp-contents/uploads/201
Oscillator 0 - 40MHz
PLL
SPI I²C
USART
Serial Communication
T0 T1 T2 T3
Timers
RAM
Program Memory
CPU
A/D Converter
Vref
CCP1
CCP2
CCP/PWM Modules
Interrupts
WDT
EEPROM Memory
I/O Ports
Port A
Port B
Port C
Port D
Port E
RESET
Power Supply 2 - 5.5V
Microcontroller

Basic Concepts

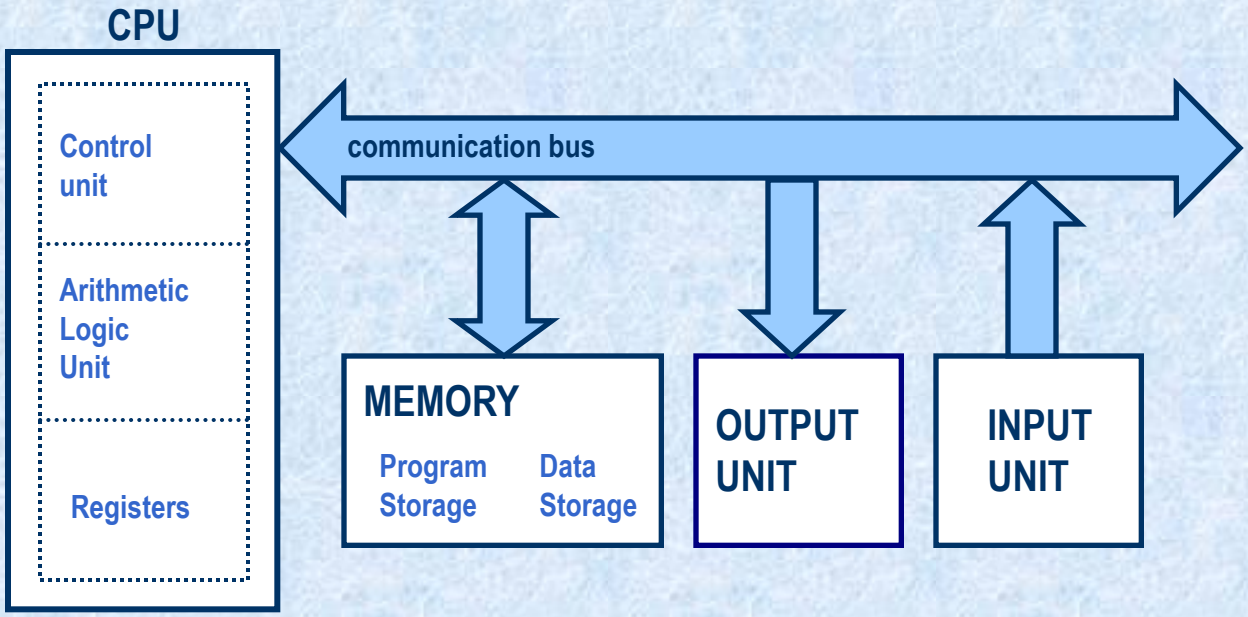
Introduction to Microcontrollers

Organization of Microcontrollers

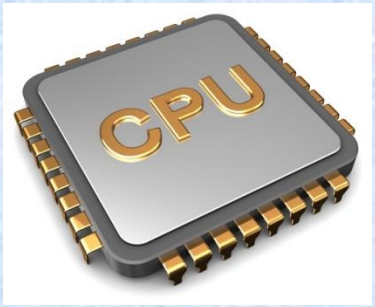
What is a Computing System?

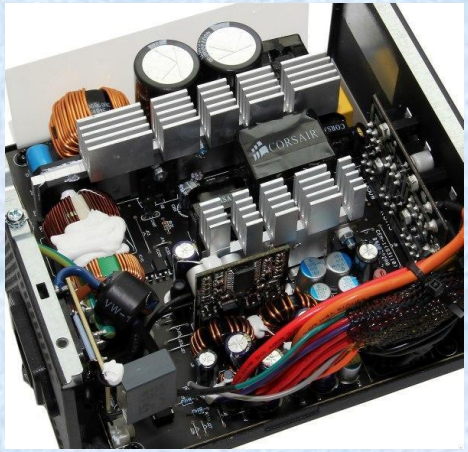


- Hardware & Software

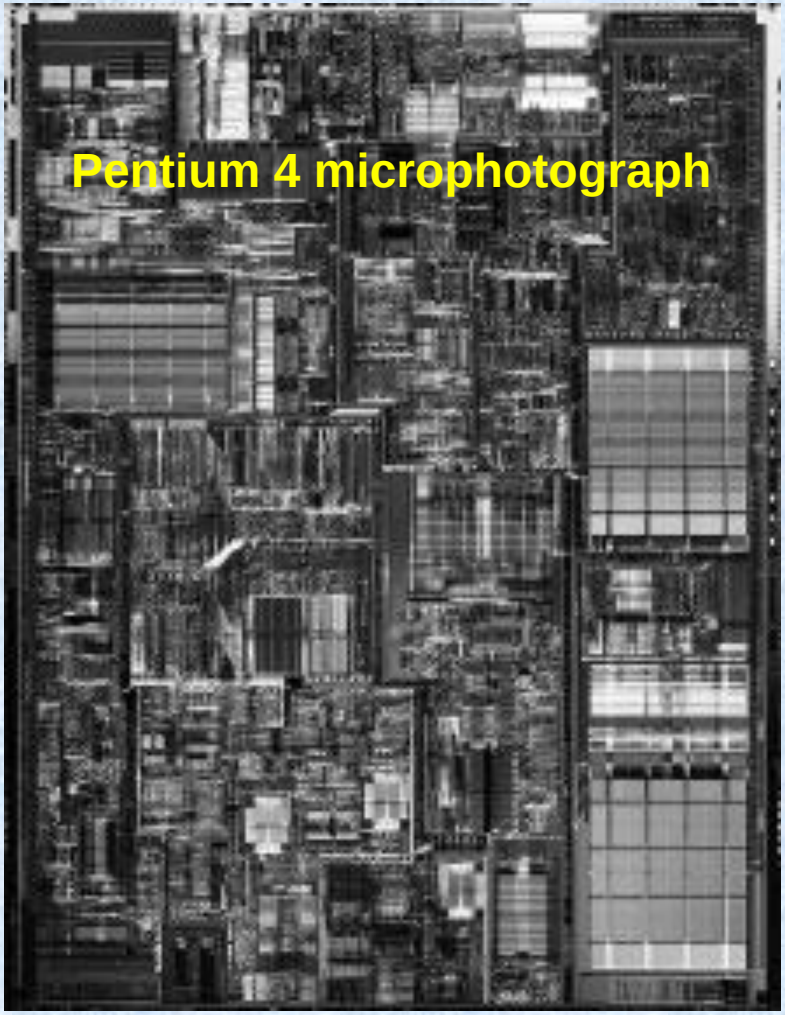


Components of Computer Systems

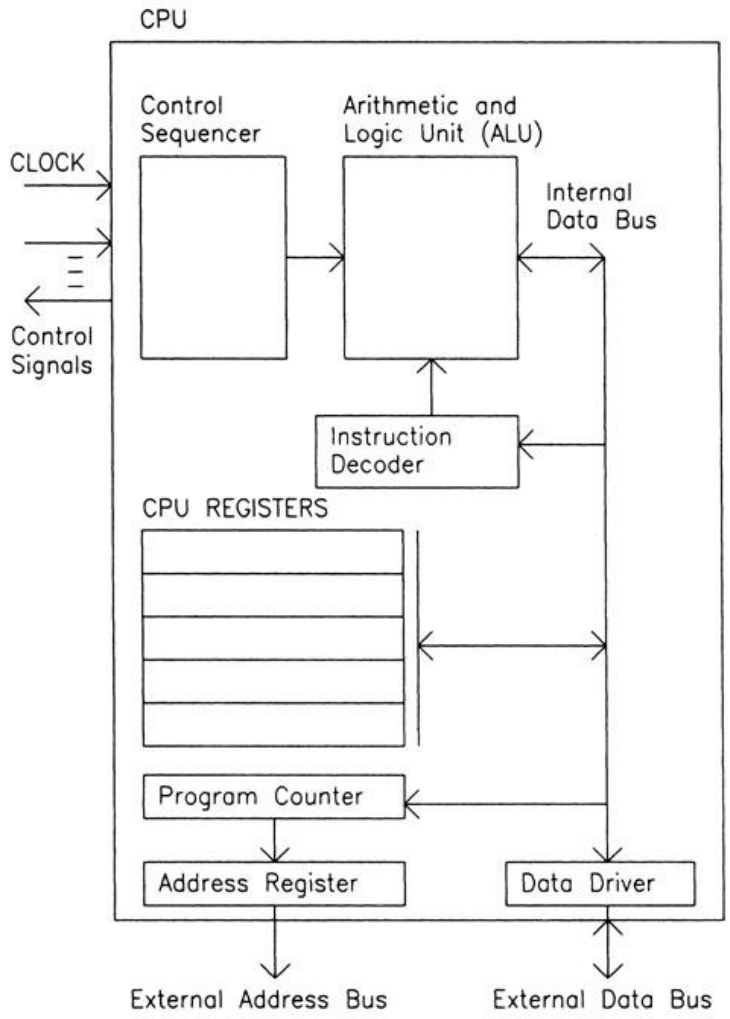




What is a Microprocessor ?

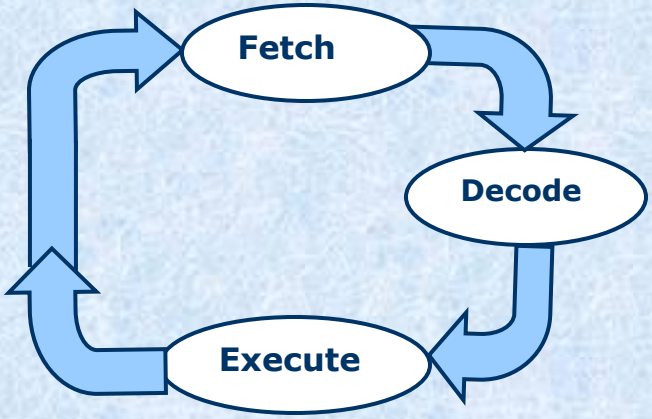


Microprocessor Structure



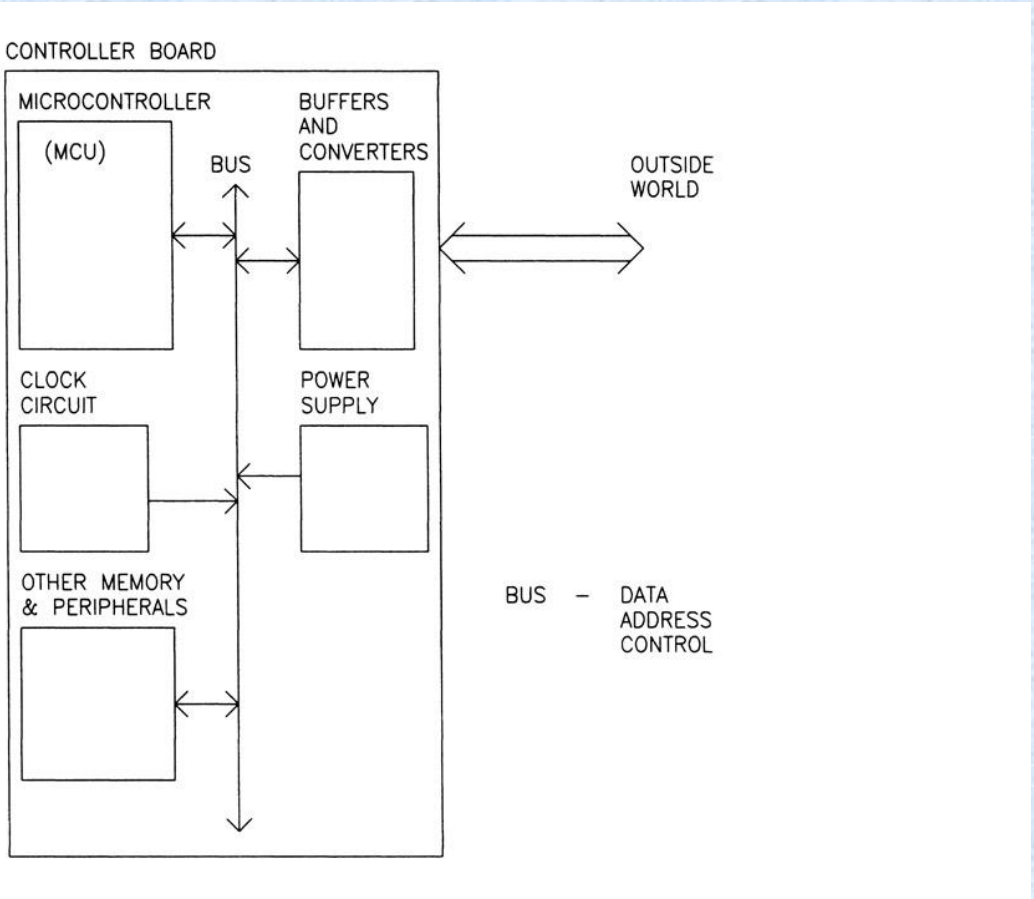
Microprocessor Basic Operation

- Program (instructions) and Data are stored in Memory
- Each instruction is read (fetched) from memory, interpreted (decoded), and executed
  - Arithmetic Logic Unit (ALU) performs operations on data
  - Data is transferred (register, memory, I/O)
- Program Counter (PC) indicates current location of program in Memory and is automatically incremented after each instruction
- Each instruction can take several clock cycles



What is a Microcomputer System?

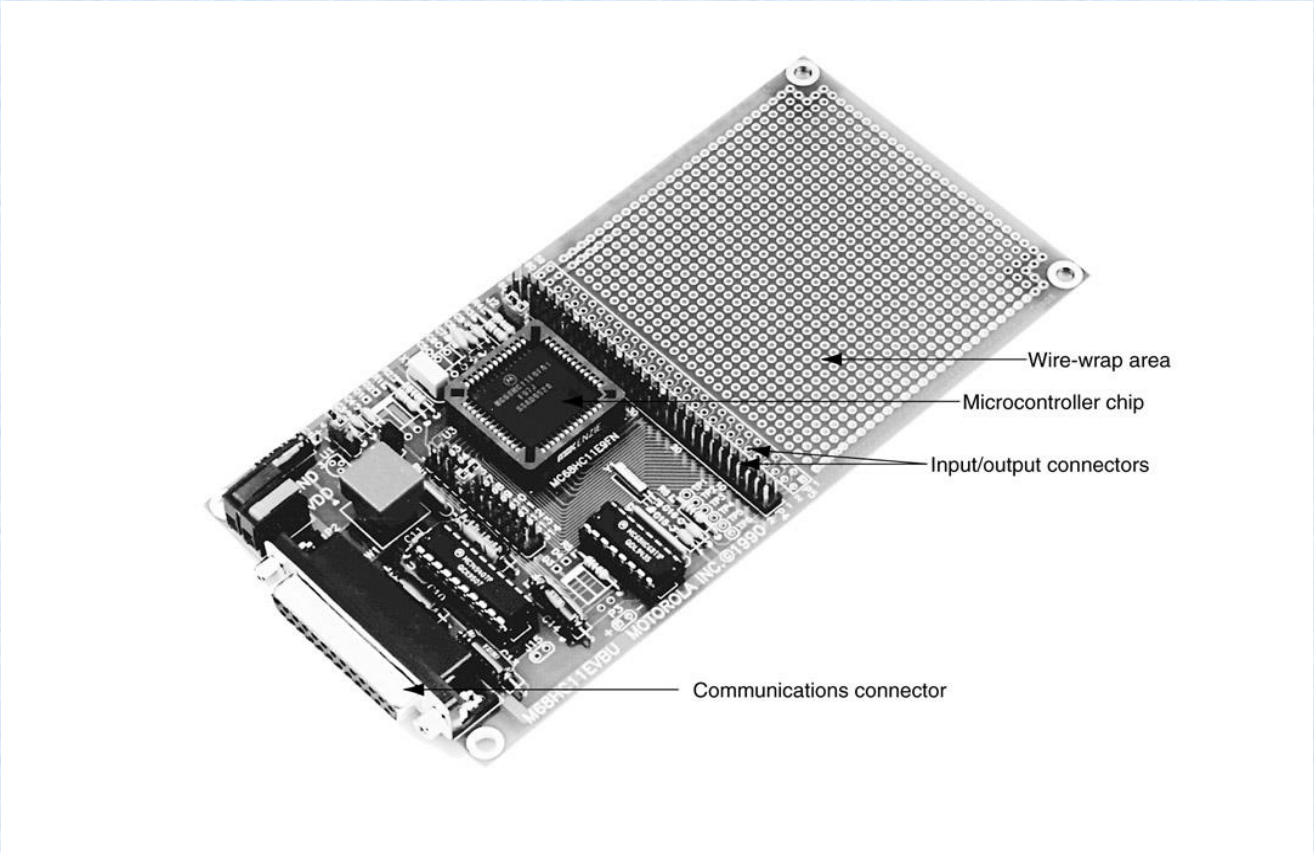
- It is a computing system based on microcontroller.



Microcontroller System

- The ***buffers*** **and** ***converters*** **condition I/O signal** levels if necessary
- The ***bus*** is a group of signals (data, address, control signal) with a common purpose.
- The ***clock*** *circuit* generates a fixed-frequency, **timing signal** for the entire system.
- The ***power supply*** converts a raw power source into the DC voltage (**nominally 5V**) required by the system.

An example of microcontroller system (MC68HC11EVBU evaluation board)



Microcontroller

- Integrated system designed to operate as an embedded computing system (a computer which is part of a larger system)
- A microcontroller is a small, low-cost and self contained computer-on-a-chip that can be used as an embedded system.
- It is composed by:
  - microprocessor (CPU),
  - ROM (for the program),
  - RAM (for the data)
  - I/O ports (to communicate/interface with external resources),
  - Peripheral devices (to make easier the interfacing and implementation of the desired functionalities),

Common Applications

- Consumer:
  - Washing machine,
  - Remote controls
  - Clocks and watches
  - Games and Toys
  - Audio/video

- Communication:
  - Telephone systems,
  - Answering machines
  - Cell phones and pagers
  - Networking (ATM, credit cards, Ethernet)

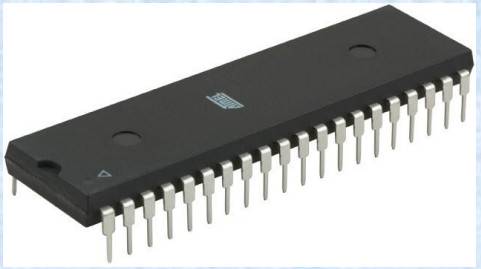
- Automotive:
  - Safety devices (Automatic Braking System, Airbag)
  - Motor control (ignition, exhausts)
  - Power windows and seats
  - Instrumentation

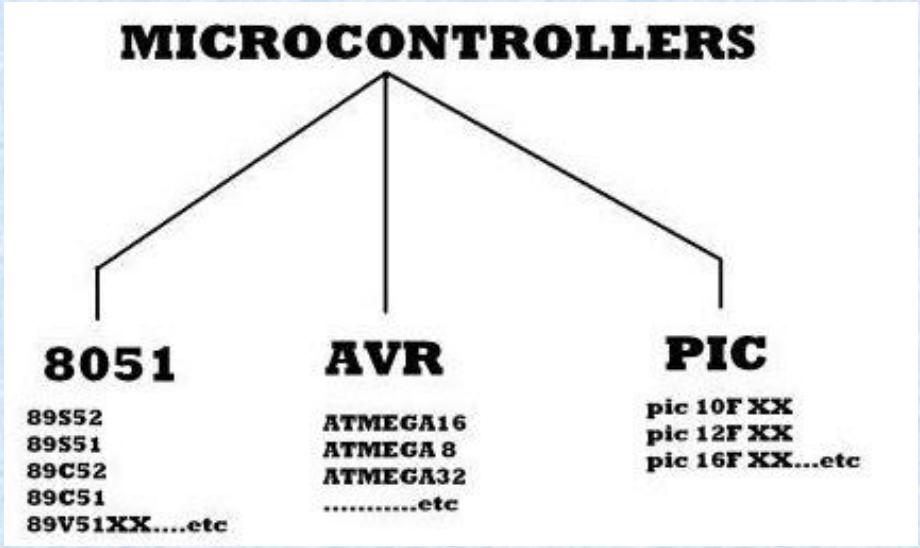
Common Applications

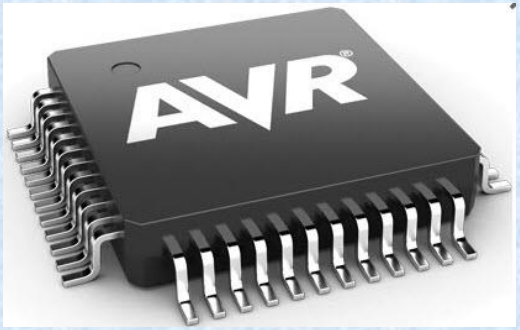
- Military:
  - Guidance systems
  - Global positioning systems
  - Target recognition systems
- Industrial:
  - Traffic control
  - Robotics
  - Production plants
  - Inventory and stock management
- Medical:
  - Cardiac monitors
  - Renal Monitors
  - Pacemakers
  - Dialysis machines

Classification of Microcontrollers

- The microcontrollers are characterized by their:
  - bits,
  - bus-width,
  - instruction set,
  - and memory structure.







Classification According to Number of Bits

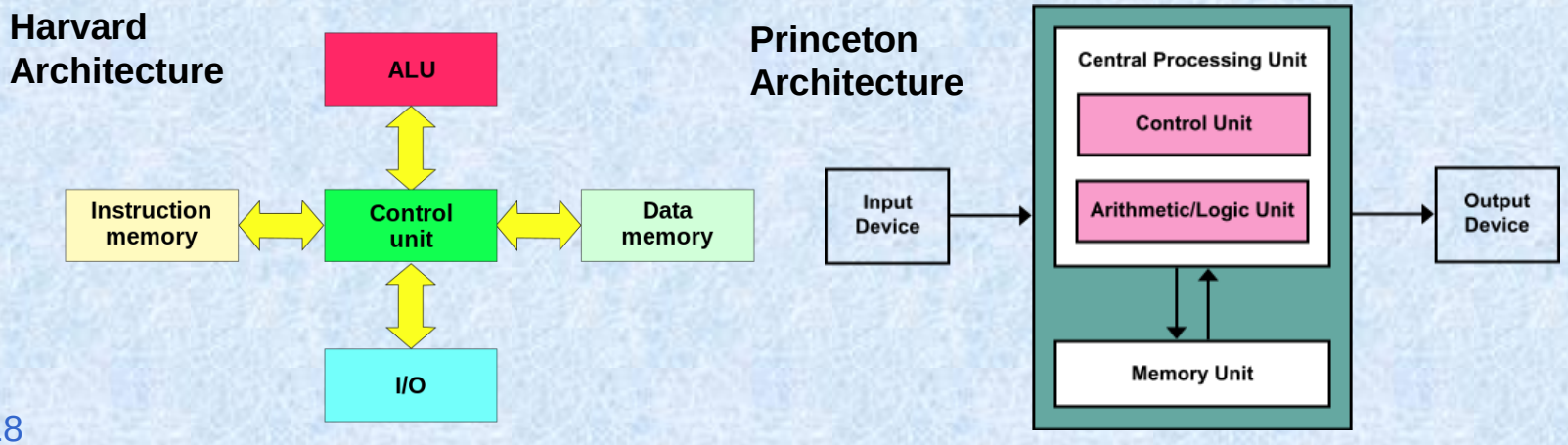
- In **8-bit** microcontroller, the point when the internal bus is 8-bit then the ALU is performs the arithmetic and logic operations.
- The **16-bit** microcontroller performs greater precision and performance as compared to 8-bit. For example 8 bit microcontrollers can only use 8 bits, resulting in a final range of 0×00 – 0xFF (0-255) for every cycle. In contrast, 16 bit microcontrollers with its 16 bit data width has a range of 0×0000 – 0xFFFF (0-65535) for every cycle.
- The **32-bit** microcontroller uses the 32-bit instructions to perform the arithmetic and logic operations. These are used in automatically controlled devices including implantable medical devices, engine control systems, office machines, appliances and other types of embedded systems.

Classification According to Memory Devices

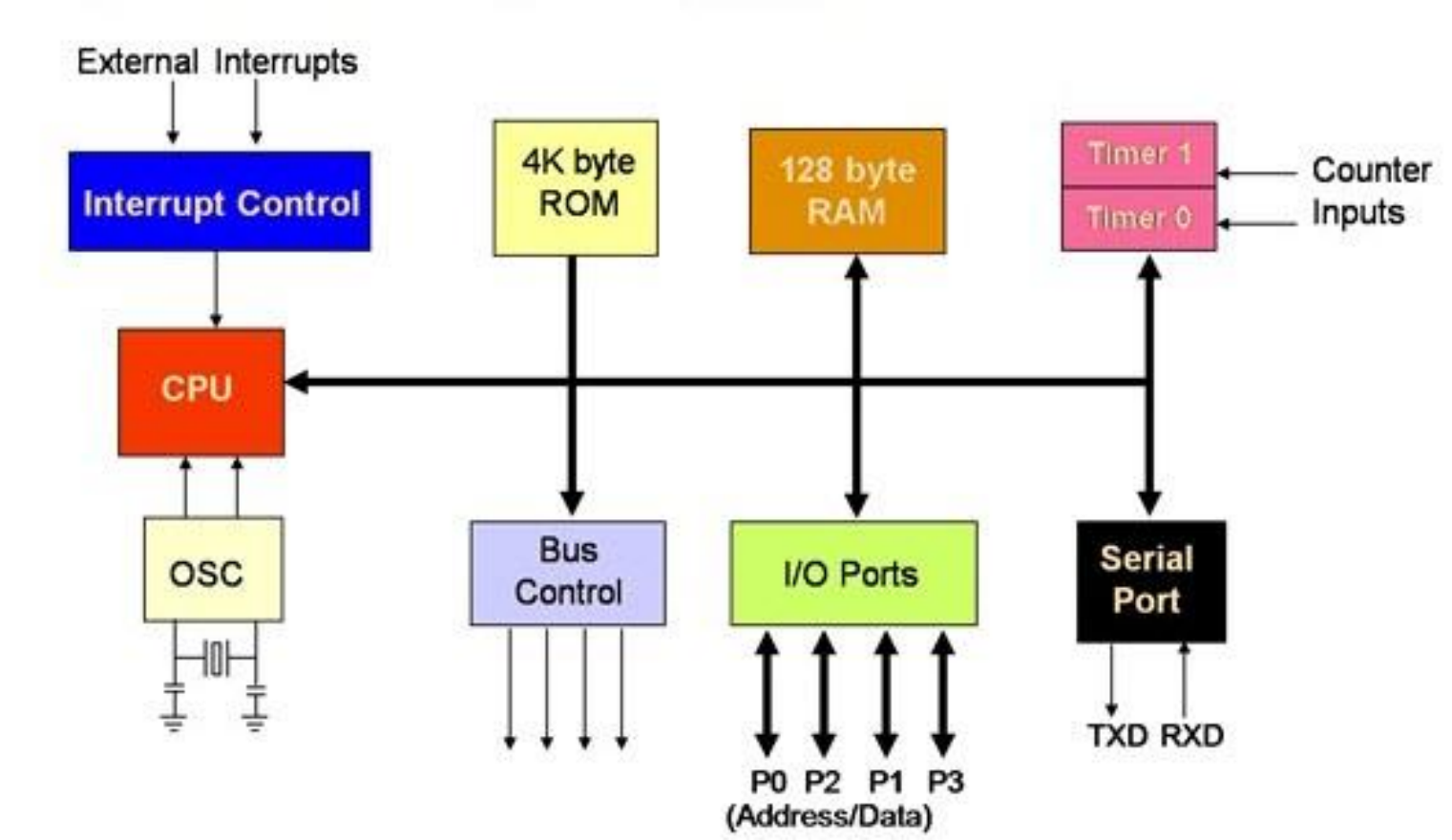
- **Embedded memory microcontroller**: When an embedded system has a microcontroller unit that has *all the functional blocks available on a chip* is called an embedded microcontroller. For example, 8051 having program & data memory, I/O ports, serial communication, counters and timers and interrupts on the chip is an embedded microcontroller.
- **External Memory Microcontroller**: When an embedded system has a microcontroller unit that has *not all the functional blocks available on a chip* is called an external memory microcontroller. For example, 8031 has no program memory on the chip is an external memory microcontroller.

Classification According to Memory Architecture

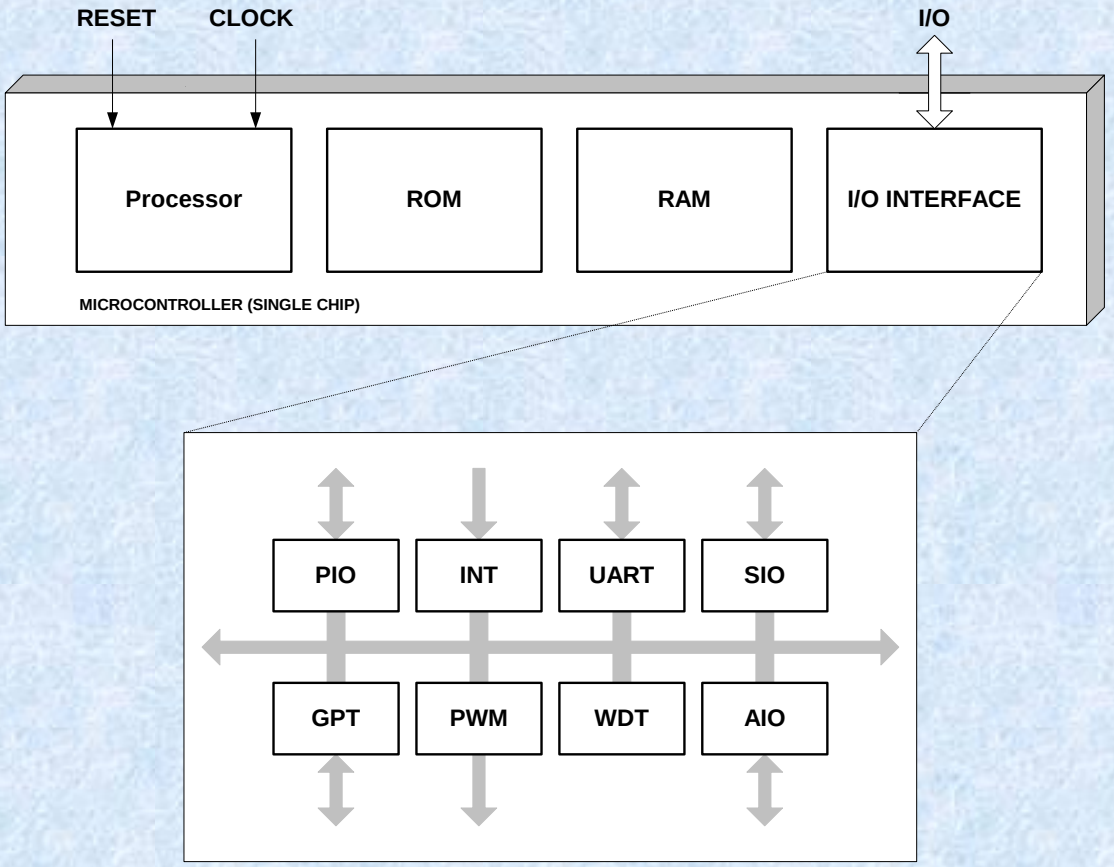
- **Harvard Memory Architecture Microcontroller**: The point when a microcontroller unit has a dissimilar memory address space for the program and data memory, the microcontroller has Harvard memory architecture in the processor.
- **Princeton Memory Architecture Microcontroller**: The point when a microcontroller has a common memory address for the program memory and data memory, the microcontroller has Princeton memory architecture in the processor.



Microcontroller Architecture



Microcontroller Organization



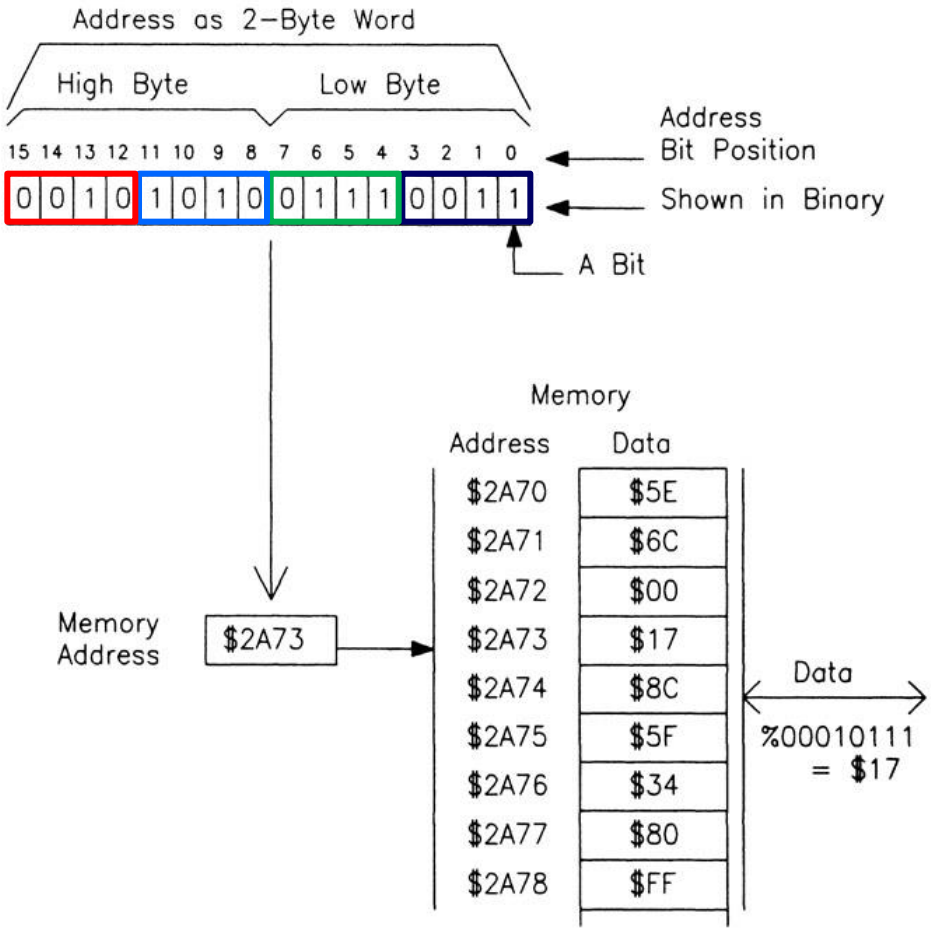
Microcontroller Functional Units

- **CPU**: Central Processing Unit
  (4,8,16,32 bit data bus)
- **ROM**: Read Only Memory (Firmware)
- **RAM**: Random Access Memory
  (Register File, Processor Stack, Temporary data)
- **PIO**: Parallel I/O (relays, sensors)
- **INT**: Interrupt Inputs
  (external/internal sources)
- **UART**: Universal Asynchronous Receiver Transmitter (e.g. RS232)
- **GPT**: General Purpose Timer
  (optional event counter)
- **PWM**: Pulse Width Modulator (motor controller)
- **WDT**: Watch Dog Timer (automatic reset)
- **AIO**: Analog I/O (ADC & DAC)

Memory basic concepts

- Digital data is stored in the form of **binary numbers**, however it is often represented using the hexadecimal numbering system.
- The ***bit*** **is the smallest digital unit**, and is either 1 or 0.
- A ***byte*** is defined to be 8 bits.
- A ***word*** varies from processor to processor and can be 8, 16, 32 or more bits.
- Normally, **the byte is the smallest addressable unit**; however, it is possible to **address individual bits in I/O registers.**
- **Motorola convention**: binary number are prefixed by **%** and hexadecimal numbers by **$**

Memory basic concepts



Memory Types

The main types of semiconductor memory are:

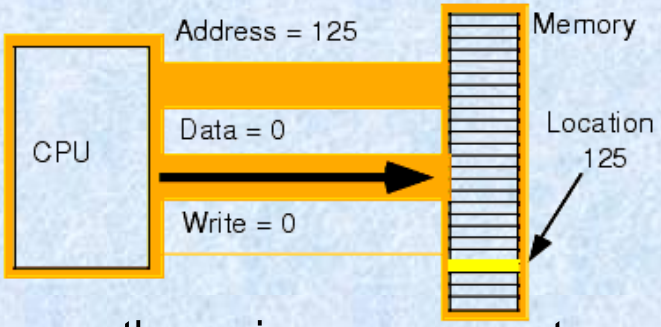
ROM – Read Only Memory

RAM – Random Access Memory

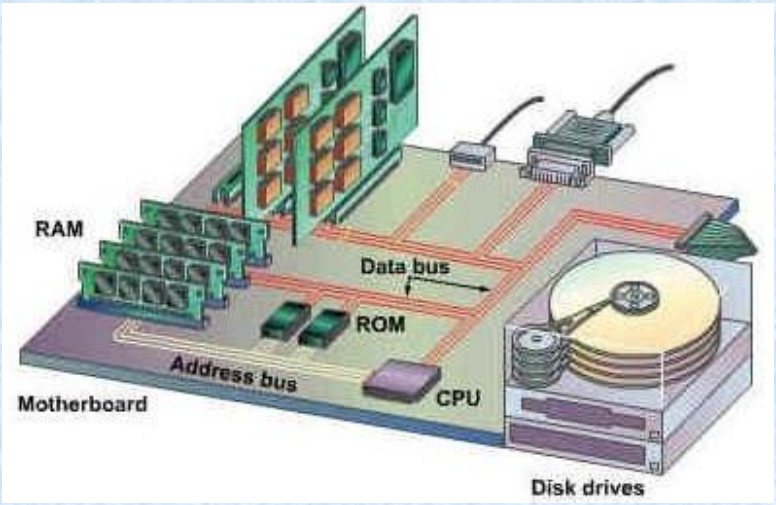
EPROM – Erasable Programmable Read Only Memory

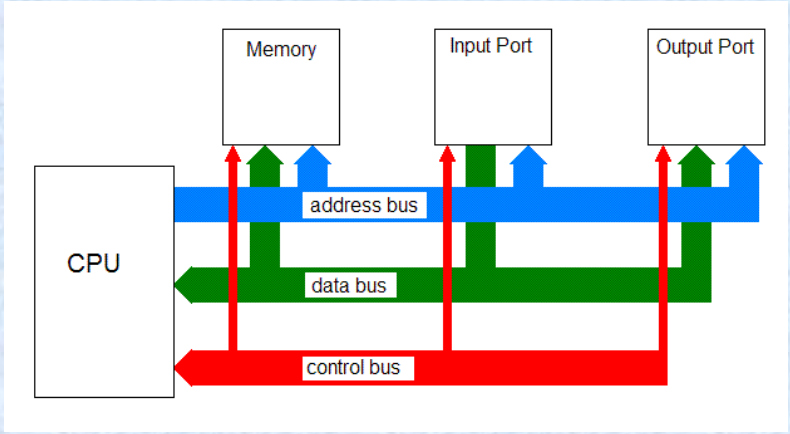
EEPROM – Electrically Erasable Programmable Read Only Memory

The Bus



- The bus provides the communication infrastructure among the various components of the system
- **Data bus** carries the information being transmitted/received.
- **Address bus** tells where the information is being transferred to/from.
- **Control bus** specifies when the information transfer take place by coordinating the access to the data bus and the address bus, and directs the data from/to the specific components.



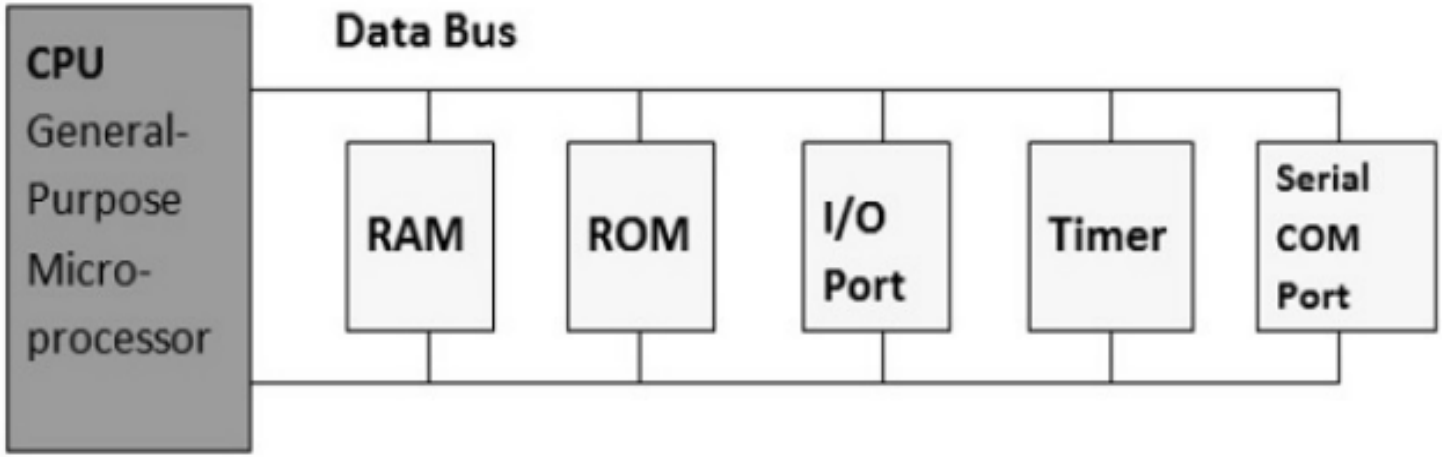


Microcontroller

Microprocessor

A microprocessor is a single VLSI chip having a CPU. In addition, it may also have other units such as coaches, floating point processing arithmetic unit, and pipelining units that help in faster processing of instructions.

Earlier generation microprocessors' fetch-and-execute cycle was guided by a clock frequency of order of ~1 MHz. Processors now operate at a clock frequency of 2GHz



A SIMPLE BLOCK DIAGRAM OF A MICROPROCESSOR

Microcontroller

Microcontroller

A microcontroller is a single-chip VLSI unit (also called **microcomputer**) which, although having limited computational capabilities, possesses enhanced input/output capability and a number of on-chip functional units.

CPU	RAM	ROM
I/O Port	Timer	Serial COM Port

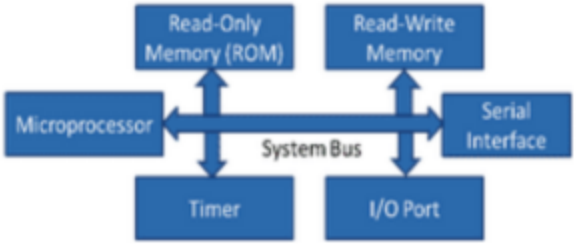
Microcontrollers are particularly used in embedded systems for real-time control applications with on-chip program memory and devices.

Microcontroller

Microprocessor vs Microcontroller

Let us now take a look at the most notable differences between a microprocessor and a microcontroller.

Microprocessor	Microcontroller
Microprocessors are multitasking in nature. Can perform multiple tasks at a time. For example, on computer we can play music while writing text in text editor.	Single task oriented. For example, a washing machine is designed for washing clothes only.
RAM, ROM, I/O Ports, and Timers can be added externally and can vary in numbers.	RAM, ROM, I/O Ports, and Timers cannot be added externally. These components are to be embedded together on a chip and are fixed in numbers.
Designers can decide the number of memory or I/O ports needed.	Fixed number for memory or I/O makes a microcontroller ideal for a limited but specific task.
External support of external memory and I/O ports makes a microprocessor-based system heavier and costlier.	Microcontrollers are lightweight and cheaper than a microprocessor.
External devices require more space and their power consumption is higher.	A microcontroller-based system consumes less power and takes less space.

Microcontroller	Microprocessor
	
The microcontroller is the heart of an embedded system.	The microprocessor is the heart of a Computer system.
The microcontroller has an external processor along with internal memory and i/O components	It is just a processor. Memory and I/O components have to be connected externally
Since memory and I/0 are present internally, the circuit is small.	Since memory and I/O have to be connected externally, the circuit becomes large.
Can be used in compact systems and hence it is an efficient technique	Cannot be used in compact systems and hence inefficient
The cost of the entire system is low	Cost of the entire system increases
Since external components are low, total	Due to external components, the entire

Microcontroller

Since external components are low, total power consumption is less and can be used with devices running on stored power like batteries.	Due to external components, the entire power consumption is high. Hence it is not suitable to used with devices running on stored power like batteries.
Most of the microcontrollers have power-saving modes like idle mode and power-saving mode. This helps to reduce power consumption even further.	Most microprocessors do not have power-saving features.
Since components are internal, most of the operations are internal instruction, hence speed is fast.	Since memory and I/O components are all external, each instruction will need an external operation, hence it is relatively slower.
Microcontrollers have more number of registers, hence the programs are easier to write.	Microprocessors have less number of registers, hence more operations are memory based
Microcontrollers are based on Harvard architecture where program memory and Data memory are separate	Microprocessors are based on the von Neumann model/architecture where programs and data are stored in the same memory module
Used mainly in washing machines, MP3 players	Mainly used in personal computers

Microcontroller

Microprocessor

- It has only the CPU inside; ie the processing powers such as Intel's Pentium 1,2,3,4 core 2 duos, i3, i5 etc.
- Don't have RAM, ROM and other peripheral on the chip. The system designer has to add them externally to make them functional.
- Application includes desktop PCs, laptops, notepads etc.
- Applications are where tasks are unspecific like developing software, games, websites, photo editing, creating documents etc.
- Since microprocessors cannot be used stand alone as it needs RAM, ROM and other peripherals the system that uses microprocessors is costlier than a microcontroller.
- The clock speed of the microprocessor is quite high as compared to the microcontroller. This can operate above 1 GHz as they perform complex tasks.

Microcontroller

Microcontroller

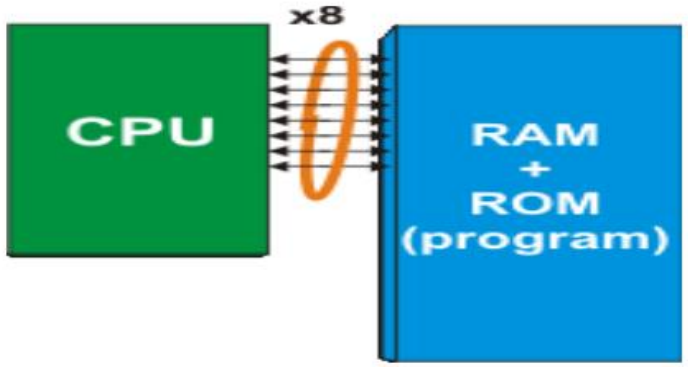
- In a microcontroller CPU, RAM, ROM, and other peripherals are embedded on a single chip.
- At times it is termed a mini computer or a computer on a single chip.
- Some giants in the manufacturing business of microcontrollers are ATMEL, microchip, TI, Freescale, Philips, Motorola etc.
- Designed to perform specific tasks. ie, the relationship between the input and output is defined.
- Since the applications are very specific, they need small resources like RAM, ROM, I/O ports and hence can be embedded on a single chip.
- The clock speed of a microcontroller varies from a few MHz to 30-50 MHz.

Microcontroller

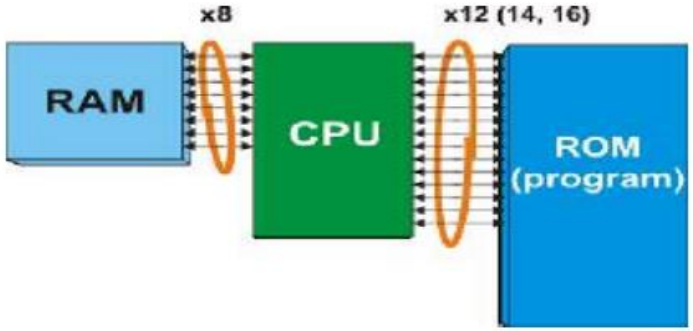
INTERNAL ARCHITECTURE

- All MCs use one of two basic design models:

  Harvard Architecture and v*on-Neumann* architecture.
- They represent two different ways of exchanging data between CPU and memory.
- **VON-NEUMANN ARCHITECTURE:**



- **HARVARD ARCHITECTURE:**



Microcontroller

CISC and RISC

- MCs with Harvard architecture are called "RISC MCs". MCs with von- Neumann's architecture are called 'CISC microcontrollers'.
- Harvard architecture is a newer concept than von-Neumann's.
- In Harvard architecture, data bus and address bus are separate. Thus a greater flow of data is possible through the CPU, and of course, a greater speed of work.
- It is also typical for Harvard architecture to have fewer instructions than von-Neumann's, and to have instructions usually executed in one cycle.

Microcontroller

CISC and RISC

CISC is a Complex Instruction Set Computer. It is a computer that can address a large number of instructions.

In the early 1980s, computer designers recommended that computers should use fewer instructions with simple constructs so that they can be executed much faster within the CPU without having to use memory. Such computers are classified as Reduced Instruction Set Computer or RISC.

CISC vs RISC

The following points differentiate a CISC from a RISC –

CISC	RISC
Larger set of instructions. Easy to program	Smaller set of Instructions. Difficult to program.
Simpler design of compiler, considering larger set of instructions.	Complex design of compiler.
Many addressing modes causing complex instruction formats.	Few addressing modes, fix instruction format.
Instruction length is variable.	Instruction length varies.
Higher clock cycles per second.	Low clock cycle per second.
Emphasis is on hardware.	Emphasis is on software.
Control unit implements large instruction set using micro-program unit.	Each instruction is to be executed by hardware.
Slower execution, as instructions are to be read from memory and decoded by the decoder unit.	Faster execution, as each instruction is to be executed by hardware.
Pipelining is not possible.	Pipelining of instructions is possible, considering single clock cycle.

Microcontroller

Compiler

The name "compiler" is primarily used for programs that translate the source code from a high level programming language to a low-level language (e.g., assembly language or machine code).

Microcontroller

Cross-Compiler

If the compiled program can run on a computer having different CPU or operating system than the computer on which the compiler compiled the program, then that compiler is known as a cross-compiler.

Decompiler

A program that can translate a program from a low-level language to a high-level language is called a decompiler.

Language Converter

A program that translates programs written in different high-level languages is normally called a language translator, source to source translator, or language converter.

A compiler is likely to perform the following operations –

- Preprocessing
- Parsing
- Semantic Analysis (Syntax-directed translation)
- Code generation
- Code optimization

Assemblers

An assembler is a program that takes basic computer instructions (called as assembly language) and converts them into a pattern of bits that the computer's processor can use to perform its basic operations. An assembler creates object code by translating assembly instruction mnemonics into opcodes, resolving symbolic names to memory locations. Assembly language uses a mnemonic to represent each low-level machine operation (opcode).

Microcontroller

Debugging Tools in an Embedded System

Debugging is a methodical process to find and reduce the number of bugs in a computer program or a piece of electronic hardware, so that it works as expected. Debugging is difficult when subsystems are tightly coupled, because a small change in one subsystem can create bugs in another. The debugging tools used in embedded systems differ greatly in terms of their development time and debugging features. We will discuss here the following debugging tools –

- Simulators
- Microcontroller starter kits
- Emulator

Simulators

Code is tested for the MCU / system by simulating it on the host computer used for code development. Simulators try to model the behavior of the complete microcontroller in software.

Functions of Simulators

A simulator performs the following functions –

- Defines the processor or processing device family as well as its various versions for the target system.
- Monitors the detailed information of a source code part with labels and symbolic arguments as the execution goes on for each single step.
- Provides the status of RAM and simulated ports of the target system for each single step execution.
- Monitors system response and determines throughput.
- Provides trace of the output of contents of program counter versus the processor registers.
- Provides the detailed meaning of the present command.
- Monitors the detailed information of the simulator commands as these are entered from the keyboard or selected from the menu.
- Supports the conditions (up to 8 or 16 or 32 conditions) and unconditional breakpoints.
- Provides breakpoints and the trace which are together the important testing and debugging tool.
- Facilitates synchronizing the internal peripherals and delays.

Microcontroller

Microcontroller Starter Kit

A microcontroller starter kit consists of −

- Hardware board (Evaluation board)
- In-system programmer
- Some software tools like compiler, assembler, linker, etc.
- Sometimes, an IDE and code size limited evaluation version of a compiler.

A big advantage of these kits over simulators is that they work in real-time and thus allow for easy input/output functionality verification. Starter kits, however, are completely sufficient and the cheapest option to develop simple microcontroller projects.

Emulators

An emulator is a hardware kit or a software program or can be both which emulates the functions of one computer system (the guest) in another computer system (the host), different from the first one, so that the emulated behavior closely resembles the behavior of the real system (the guest).

Emulation refers to the ability of a computer program in an electronic device to emulate (imitate) another program or device. Emulation focuses on recreating an original computer environment. Emulators have the ability to maintain a closer connection to the authenticity of the digital object. An emulator helps the user to work on any kind of application or operating system on a platform in a similar way as the software runs as in its original environment.

Microcontroller

Peripheral Devices in Embedded Systems

Embedded systems communicate with the outside world via their peripherals, such as following &mins;

- Serial Communication Interfaces (SCI) like RS-232, RS-422, RS-485, etc.
- Synchronous Serial Communication Interface like I2C, SPI, SSC, and ESSI
- Universal Serial Bus (USB)
- Multi Media Cards (SD Cards, Compact Flash, etc.)
- Networks like Ethernet, LonWorks, etc.
- Fieldbuses like CAN-Bus, LIN-Bus, PROFIBUS, etc.
- imers like PLL(s), Capture/Compare and Time Processing Units.
- Discrete IO aka General Purpose Input/Output (GPIO)
- Analog to Digital/Digital to Analog (ADC/DAC)
- Debugging like JTAG, ISP, ICSP, BDM Port, BITP, and DP9 ports

Criteria for Choosing Microcontroller

While choosing a microcontroller, make sure it meets the task at hand and that it is cost effective. We must see whether an 8-bit, 16-bit or 32-bit microcontroller can best handle the computing needs of a task. In addition, the following points should be kept in mind while choosing a microcontroller –

- **Speed** – What is the highest speed the microcontroller can support?
- **Packaging** – Is it 40-pin DIP (Dual-inline-package) or QFP (Quad flat package)? This is important in terms of space, assembling, and prototyping the end-product.
- **Power Consumption** – This is an important criteria for battery-powered products.
- **Amount of RAM and ROM** on the chip.
- **Count of I/O pins and Timers** on the chip.
- **Cost per Unit** – This is important in terms of final cost of the product in which the microcontroller is to be used.

Further, make sure you have tools such as compilers, debuggers, and assemblers, available with the microcontroller. The most important of all, you should purchase a microcontroller from a reliable source.

Microcontroller

1. Introduction to Microcontroller

- Figure 1.1 shows the block diagram of a typical microcontroller. All components are connected via an internal bus and are all integrated on one chip. The modules are connected to the outside world via I/O pins.

Microcontroller

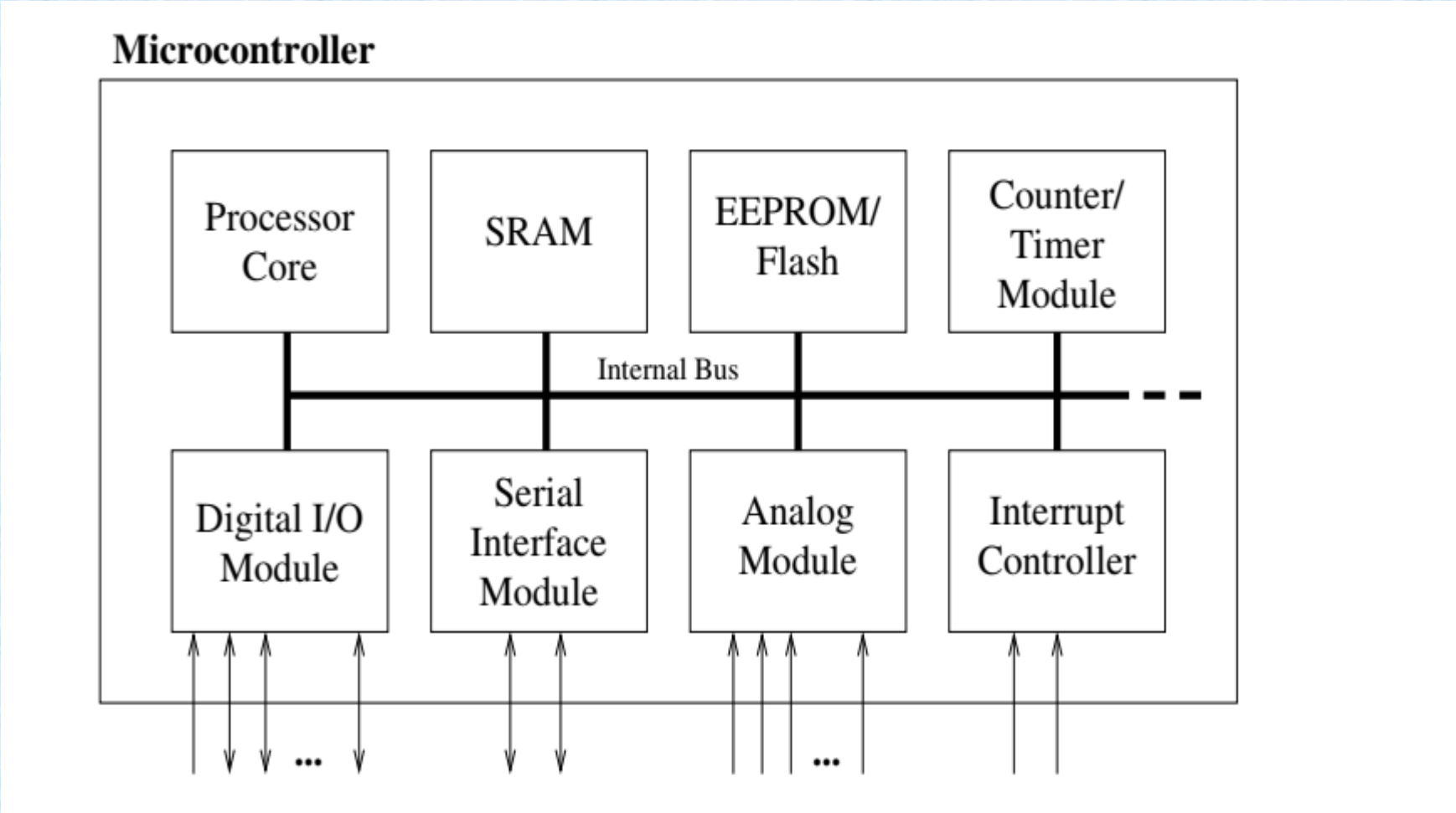


Figure 1.1 Basic layout of a microcontroller.

Microcontroller

- The following list contains the modules typically found in a microcontroller.

1-Processor Core: The CPU of the controller. It contains the arithmetic logic unit, the control unit, and the registers (stack pointer, program counter, accumulator register, register file, . . .).

2-Memory: The memory is sometimes split into program memory and data memory. In larger controllers, a DMA controller handles data transfers between peripheral components and the memory.

Microcontroller

3- **Interrupt Controller:** Interrupts are useful for interrupting the normal program flow in case of (important) external or internal events. In conjunction with sleep modes, they help to conserve power.

4- **Timer/Counter:** Most controllers have at least one and more likely 2-3 Timer/Counters, which can be used to timestamp events, measure intervals, or count events.

Microcontroller

Many controllers also contain PWM (pulse width modulation) outputs, which can be used to drive motors or for safe breaking (antilock brake system, ABS). Furthermore the PWM output can, in conjunction with an external filter, be used to realize a cheap digital/analog converter.

5-**Digital I/O:** Parallel digital I/O ports are one of the main features of microcontrollers. The number of I/O pins varies from 3-4 to over 90, depending on the controller family and the controller type.

Microcontroller

6- **Analog I/O:** Apart from a few small controllers, most microcontrollers have integrated analog/digital converters, which differ in the number of channels (2-16) and their resolution (8-12 bits). The analog module also generally features an analog comparator. In some cases, the microcontroller includes digital/analog converters.

7- **Interfaces:** Controllers generally have at least one serial interface which can be used to download the program and for communication with the development PC in general. Since serial interfaces can also be used to communicate with external peripheral devices, most controllers offer several and varied interfaces like SPI and SCI.

Microcontroller

Many microcontrollers also contain integrated bus controllers for the most common (field)busses. IIC and CAN controllers lead the field here. Larger microcontrollers may also contain PCI, USB, or Ethernet interfaces.

8- **Watchdog Timer:** Since safety-critical systems form a major application area of microcontrollers, it is important to guard against errors in the program and/or the hardware. The watchdog timer is used to reset the controller in case of software "crashes".

Microcontroller

9- **Debugging Unit:** Some controllers are equipped with additional hardware to allow remote debugging of the chip from the PC. So there is no need to download special debugging software, which has the distinct advantage that erroneous application code cannot overwrite the debugger.

To summarize, a microcontroller is a (stripped-down) processor which is equipped with memory, timers, (parallel) I/O pins and other on-chip peripherals. The driving element behind all this is cost: Integrating all elements on one chip saves space and leads to both lower manufacturing costs and shorter development times.

Microcontroller

- As a result, microcontrollers are generally tailored for specific applications, and there is a wide variety of microcontrollers to choose from.
- The first choice a designer has to make is the controller family – it defines the controller's architecture. All controllers of a family contain the same processor core and hence are code-compatible, but they differ in the additional components like the number of timers or the amount of memory.

Microcontroller

- You will find that there are many different controller families like 8051, PIC, HC, ARM to name just a few, and that even within a single controller family you may again have a choice of many different controllers.

Microcontroller

Controller	Flash (KB)	SRAM (Byte)	EEPROM (Byte)	I/O-Pins	A/D (Channels)	Interfaces
AT90C8534	8	288	512	7	8	
AT90LS2323	2	128	128	3		
AT90LS2343	2	160	128	5		
AT90LS8535	8	512	512	32	8	UART, SPI
AT90S1200	1	64		15		
AT90S2313	2	160	128	15		
ATmega128	128	4096	4096	53	8	JTAG, SPI, IIC
ATmega162	16	1024	512	35		JTAG, SPI
ATmega169	16	1024	512	53	8	JTAG, SPI, IIC
ATmega16	16	1024	512	32	8	JTAG, SPI, IIC
ATtiny11	1		64	5+1 In		
ATtiny12	1		64	6		SPI
ATtiny15L	1		64	6	4	SPI
ATtiny26	2	128	128		16	SPI
ATtiny28L	2	128		11+8 In		

Table 1.1: Comparison of AVR 8-bit controllers (AVR, ATmega, ATtiny).

Microcontroller

1.2 Frequently Used Terms

1- **Microprocessor:** This is a normal CPU (Central Processing Unit) as you can find in a PC. Communication with external devices is achieved via a data bus, hence the chip mainly features data and address pins as well as a couple of control pins. All peripheral devices (memory, floppy controller, USB controller, timer, . . .) are connected to the bus. A microprocessor cannot be operated stand-alone, at the very least it requires some memory and an output device to be useful.

Microcontroller

2- **Microcontroller:** A microcontroller already contains all components which allow it to operate standalone, and it has been designed in particular for monitoring and/or control tasks. In consequence, in addition to the processor it includes memory, various interface controllers, one or more timers, an interrupt controller, and last but definitely not least general purpose I/O pins which allow it to directly interface to its environment. Microcontrollers also include bit operations which allow you to change one bit within a byte without touching the other bits.

Microcontroller

3- **Mixed-Signal Controller:** This is a microcontroller which can process both digital and analog signals.

4- **Embedded System:** A major application area for microcontrollers are embedded systems. In embedded systems, the control unit is integrated into the systems. As an example, think of a cell phone, where the controller is included in the device. This is easily recognizable as an embedded system. On the other hand, if you use a normal PC in a factory to control an assembly line, this also meets many of the definitions of an embedded system. The same PC, however, equipped with a normal operating system and used by the night guard to kill time is certainly no embedded system.

Microcontroller

5- **Real-Time System:** Controllers are frequently used in real-time systems, where the reaction to an event has to occur within a specified time. This is true for many applications in aerospace, railroad, or automotive areas, e.g., for brake-by-wire in cars.

Microcontroller

1.3 Notation

- When we talk about the values of digital lines, we generally mean their logical values, 0 or 1. We indicate the complement of a logical value X with $\bar{X}$, so $\bar{1}$ = 0 and 0 = 1.
- Hexadecimal values are denoted by a preceding $ or 0x. Binary values are either given like decimal values if it is obvious that the value is binary, or they are marked with $(\cdot)_2$.
- The notation M[X] is used to indicate a memory access at address X.
- In our assembler examples, we tend to use general-purpose registers, which are labeled with R and a number, e.g., R0.
- The $\propto$ sign means "proportional to".
- In a few cases, we will need intervals. We use the standard interval notations, which are [.,.] for a closed interval, [.,.) and (.,.] for half-open intervals, and (.,.) for an open interval. Variables denoting intervals will be overlined, e.g. d_{latch} = (0, 1]. The notation d_{latch} + 2 adds the constant to the interval, resulting in (0, 1] + 2 = (2, 3].
- We use *k* as a generic variable, so do not be surprised if *k* means different things in different sections or even in different paragraphs within a section.

Microcontroller

Furthermore, you should be familiar with the following *power prefixes*[4]:

Name	Prefix	Power	Name	Prefix	Power
kilo	k	10^{3}	milli	m	10^{-3}
mega	M	10^{6}	micro	μ, u	10^{-6}
giga	G	10^{9}	nano	n	10^{-9}
tera	T	10^{12}	pico	p	10^{-12}
peta	P	10^{15}	femto	f	10^{-15}
exa	E	10^{18}	atto	a	10^{-18}
zetta	Z	10^{21}	zepto	z	10^{-21}
yotta	Y	10^{24}	yocto	y	10^{-24}

Table 1.2: Power Prefixes

Microcontroller

2. Microcontroller Components

2.1 Processor Core

- A basic CPU architecture is depicted in Figure 2.1. It consists of the data path, which executes instructions, and of the control unit, which basically tells the data path what to do.

Microcontroller

2.1.1 Architecture

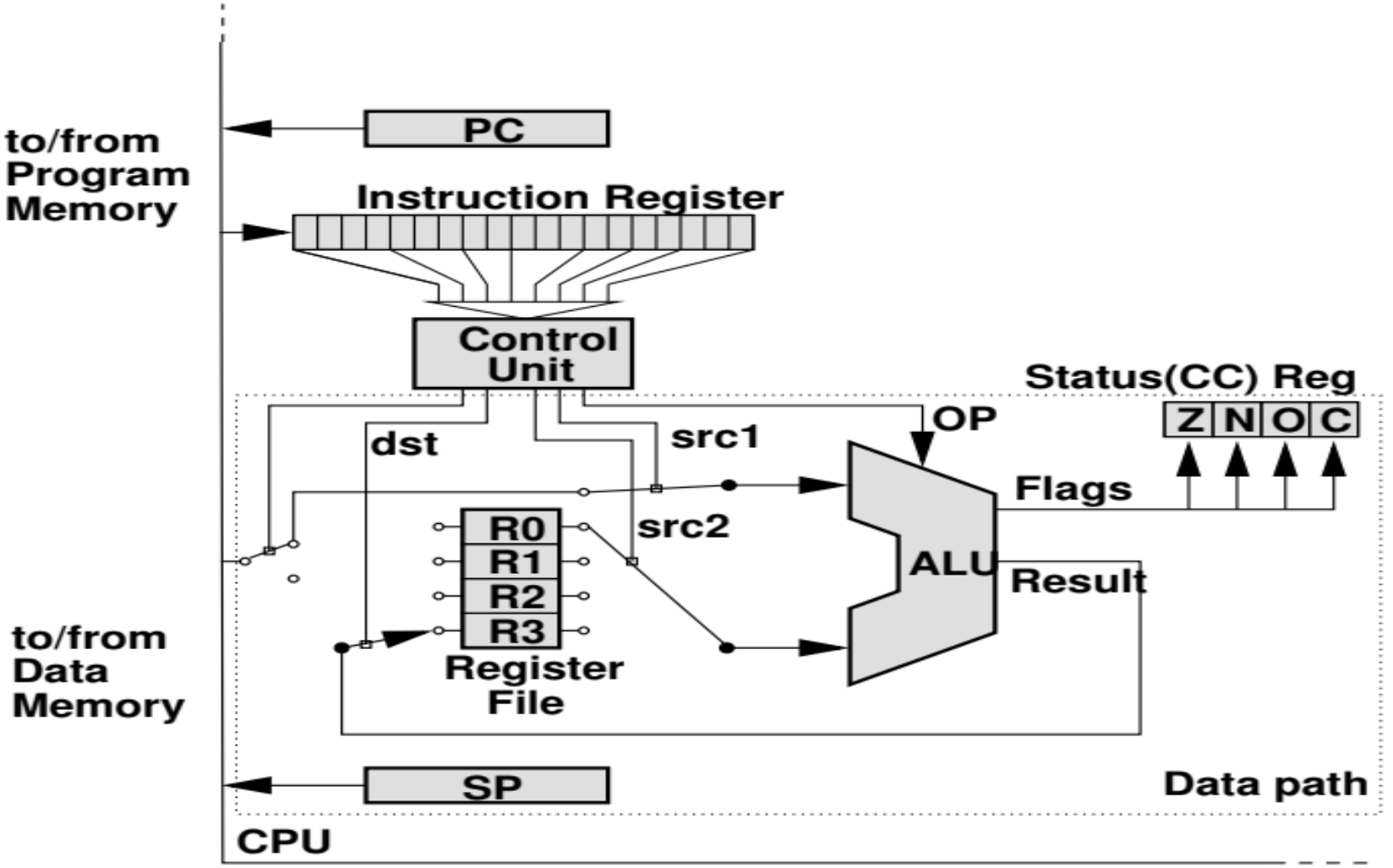


Figure 2.1: Basic CPU architecture.

Microcontroller

1- **Arithmetic Logic Unit**

At the core of the CPU is the arithmetic logic unit (ALU), which is used to perform computations (AND, ADD, INC, . . .). Several control lines select which operation the ALU should perform on the input data. The ALU takes two inputs and returns the result of the operation as its output. Source and destination are taken from registers or from memory. In addition, the ALU stores some information about the nature of the result in the status register (also called condition code register):

Microcontroller

- **Z (Zero):** The result of the operation is zero.
- **N (Negative):** The result of the operation is negative, that is, the most significant bit (msb) of the result is set (1).
- **O (Overflow):** The operation produced an overflow, that is, there was a change of sign in a two's- complement operation.
- **C (Carry):** The operation produced a carry.

Microcontroller

2- **Register File**

The register file contains the working registers of the CPU. It may either consist of a set of general purpose registers (generally 16–32, but there can also be more), each of which can be the source or destination of an operation, or it consists of some dedicated registers. Dedicated registers are e.g. an accumulator, which is used for arithmetic/logic operations, or an index register, which is used for some addressing modes.

In any case, the CPU can take the operands for the ALU from the file, and it can store the operation's result back to the register file. Alternatively, operands/result can come from/be stored to the memory. However, memory access is much slower than access to the register file, so it is usually wise to use the register file if possible.

Microcontroller

Example: Use of Status Register

The status register is very useful for a number of things, e.g., for adding or subtracting numbers that exceed the CPU word length. The CPU offers operations which make use of the carry flag, like ADDC[a] (add with carry). Consider for example the operation 0x01f0 + 0x0220 on an 8-bit CPU[b] [c]:

```
CLC                 ; clear carry flag
LD R0,    #0xf0     ; load first low byte into register R0
ADDC R0,  #0x20     ; add 2nd low byte with carry (carry <- 1)
LD R1,    #0x01     ; load first high byte into R0
ADDC R1,  #0x02     ; add 2nd high byte, carry from
                    ; previous ADC is added
```

The first ADDC stores 0x10 into R0, but sets the carry bit to indicate that there was an overflow. The second ADDC simply adds the carry to the result. Since there is no overflow in this second operation, the carry is cleared. R1 and R0 contain the 16 bit result 0x0410. The same code, but with a normal ADD (which does not use the carry flag), would have resulted in 0x0310.

---

[a] We will sometimes use assembler code to illustrate points. We do not use any specific assembly language or instruction set here, but strive for easily understood pseudo-code.

[b] A # before a number denotes a constant.

[c] We will denote hexadecimal values with a leading `$` (as is generally done in Assembly language) or a leading `0x` (as is done in C).

Microcontroller

3- Stack Pointer

The stack is a portion of consecutive memory in the data space which is used by the CPU to store return addresses and possibly register contents during subroutine and interrupt service routine calls. It is accessed with the commands PUSH (put something on the stack) and POP (remove something from the stack). To store the current fill level of the stack, the CPU contains a special register called the stack pointer (SP), which points to the top of the stack. Stacks typically grow “down”, that is, from the higher memory addresses to the lower addresses. So the SP generally starts at the end of the data memory and is decremented with every push and incremented with every pop. The reason for placing the stack pointer at the end of the data memory is that your variables are generally at the start of the data memory, so by putting the stack at the end of the memory it takes longest for the two to collide.

Microcontroller

- Unfortunately, there are two ways to interpret the memory location to which the SP points: It can either be seen as the first free address, so a PUSH should store data there and then decrement the stack pointer as depicted in Figure 2.2 (the Atmel AVR controllers use the SP that way), or it can be seen as the last used address, so a PUSH first decrements the SP and then stores the data at the new address (this interpretation is adopted for example in Motorola's HCS12). Since the SP must be initialized by the programmer, you must look up how your controller handles the stack and either initialize the SP to the last address in memory (if a push stores first and decrements afterwards) or to the last address + 1 (if the push decrements first).

Microcontroller

- As we have mentioned, the controller uses the stack during subroutine calls and interrupts, that is, whenever the normal program flow is interrupted and should resume later on. Since the return address is a pre-requisite for resuming program execution after the point of interruption, every controller pushes at least the return address onto the stack. Some controllers even save register contents on the stack to ensure that they do not get overwritten by the interrupting code. This is mainly done by controllers which only have a small set of dedicated registers.

Microcontroller

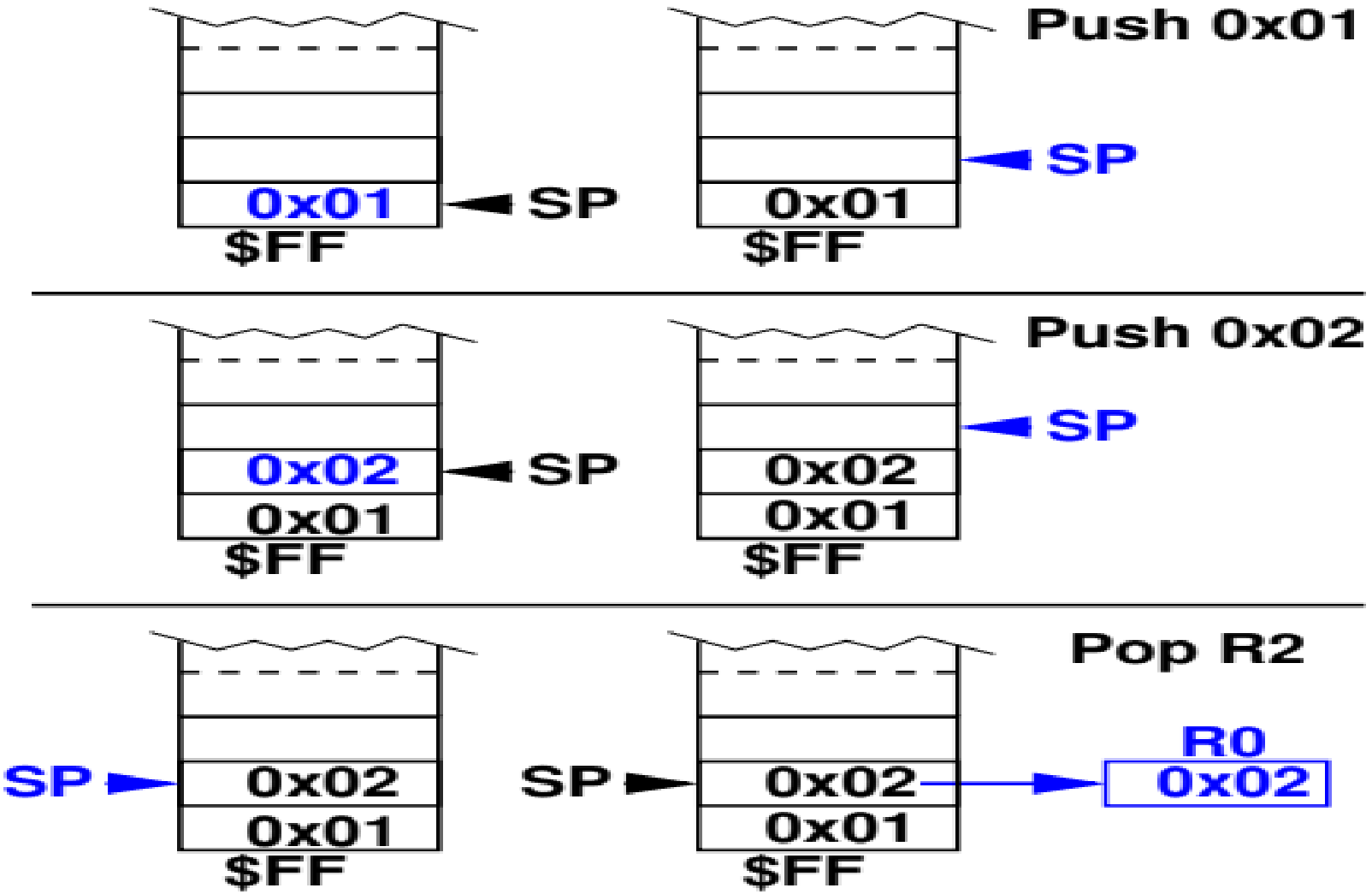


Figure 2.2: Stack operation (write first).

Microcontroller

Control Unit

- It is the task of the control unit to determine which operation should be executed next and to configure the data path accordingly.
- To do so, another special register, the ***program counter (PC)***, is used to store the address of the next program instruction.
- The control unit loads this instruction into the ***instruction register (IR)***, decodes the instruction, and sets up the data path to execute it.

Microcontroller

- Data path configuration includes providing the appropriate inputs for the ALU (from registers or memory), selecting the right ALU operation, and making sure that the result is written to the correct destination (register or memory).
- The PC is either incremented to point to the next instruction in the sequence, or is loaded with a new address in the case of a jump or subroutine call.
- After a reset, the PC is typically initialized to $0000.

Microcontroller

- Traditionally, the control unit was hard-wired, that is, it basically contained a look-up table which held the values of the control lines necessary to perform the instruction, plus a rather complex decoding logic.
- This meant that it was difficult to change or extend the instruction set of the CPU. To ease the design of the control unit, ***Maurice Wilkes*** reflected that the control unit is actually a small CPU by itself and could benefit from its own set of ***microinstructions***.
- In his subsequent control unit design, program instructions were broken down into microinstructions, each of which did some small part of the whole instruction (like providing the correct register for the ALU).

Microcontroller

- This essentially made control design a programming task: Adding a new instruction to the instruction set boiled down to programming the instruction in microcode.
- As a consequence, it suddenly became comparatively easy to add new and complex instructions, and instruction sets grew rather large and powerful as a result.
- This earned the architecture the name ***Complex Instruction Set Computer (CISC)***.

Microcontroller

- Of course, the powerful instruction set has its price, and this price is speed: Microcoded instructions execute slower than hard-wired ones.
- Furthermore, studies revealed that only 20% of the instructions of a CISC machine are responsible for 80% of the code (80/20 rule).
- This and the fact that these complex instructions can be implemented by a combination of simple ones gave rise to a movement back towards simple hard-wired architectures, which were correspondingly called ***Reduced Instruction Set Computer (RISC)***.

Microcontroller

RISC: The RISC architecture has simple, hard-wired instructions which often take only one or a few clock cycles to execute. RISC machines feature a small and fixed code size with comparatively few instructions and few addressing modes. As a result, execution of instructions is very fast, but the instruction set is rather simple.

Microcontroller

CISC: The CISC architecture is characterized by its complex microcoded instructions which take many clock cycles to execute. The architecture often has a large and variable code size and offers many powerful instructions and addressing modes. In comparison to RISC, CISC takes longer to execute its instructions, but the instruction set is more powerful.

Microcontroller

- Of course, when you have two architectures, the question arises which one is better. In the case of **RISC** vs. **CISC**, the answer depends on what you need.
  - If your solution frequently employs a **powerful instruction** or **addressing mode** of a given **CISC** architecture, you probably will be better off using **CISC**.
  - If you mainly need **simple instructions** and addressing modes, you are most likely better off using **RISC**.

Microcontroller

- Of course, this choice also depends on other factors like the clocking frequencies of the processors in question.

Microcontroller

Example: CISC vs. RISC

Let us compare a complex CISC addressing mode with its implementation in a RISC architecture. The 68030 CPU from Motorola offers the addressing mode "memory indirect preindexed, scaled":

```
MOVE D1, ([24,A0,4*D0])
```

This operation stores the contents of register D1 into the memory address

$$24 + [A0] + 4 * [D0]$$

where square brackets designate "contents of" the register or memory address.

To simulate this addressing mode on an Atmel-like RISC CPU, we need something like the following:

```
LD   R1, X       ; load data indirect (from [X] into R1)
LSL  R1          ; shift left -> multiply with 2
LSL  R1          ; 4*[D0] completed
MOV  X, R0       ; set pointer (load A0)
LD   R0, X       ; load indirect ([A0] completed)
ADD  R0, R1      ; add obtained pointers ([A0]+4*[D0])
LDI  R1, $24     ; load constant ($ = hex)
ADD  R0, R1      ; and add (24+[A0]+4*[D0])
MOV  X, R0       ; set up pointer for store operation
ST   X, R2       ; write value ([24+[A0]+4*[D0]] <- R2)
```

In this code, we assume that R0 takes the place of A0, X replaces D0, and R2 contains the value of D1.

Although the RISC architecture requires 10 instructions to do what the 68030 does in one, it is actually not slower: The 68030 instruction takes 14 cycles to complete, the corresponding RISC code requires 13 cycles, assuming that all instructions take one clock cycle, except memory load/store, which take two.

Microcontroller

2.1.2 Instruction Set

- The instruction set is an important characteristic of any CPU. It influences the code size, that is, how much memory space your program takes. Hence, you should choose the controller whose instruction set best fits your specific needs.
- The metrics of the instruction set that are important for a design decision are:

Microcontroller

- Instruction Size
- Execution Speed
- Available Instructions
- Addressing Modes

1- **Instruction Size**

An instruction contains in its **opcode** information about both the **operation** that should be executed and its **operands.**

Obviously, a machine with many different instructions and addressing modes requires longer opcodes than a machine with only a few instructions and addressing modes, so ***CISC*** machines tend to have longer opcodes than ***RISC*** machines.

Microcontroller

- Note that longer opcodes do not necessarily imply that your program will take up more space than on a machine with short opcodes. As we pointed out in our CISC vs. RISC example, it depends on what you need. For instance, the 10 lines of ATmega16 RISC code require 20 byte of code (each instruction is encoded in 16 bits), whereas the 68030 instruction fits into 4 bytes. So here, the 68030 clearly wins. If, however, you only need instructions already provided by an architecture with short opcodes, it will most likely beat a machine with longer opcodes.

Microcontroller

- Obviously, a lot of space in the opcode is taken up by the operands. So one way of reducing the instruction size is to cut back on the number of operands that are explicitly encoded in the opcode. In consequence, we can distinguish four different architectures, depending on how many explicit operands a binary operation like ADD requires:

Microcontroller

Stack Architecture: This architecture, also called ***0-address format architecture***, does not have any explicit operands. Instead, the operands are organized as a stack: An instruction like ADD takes the top-most two values from the stack, adds them, and puts the result on the stack.

Microcontroller

Accumulator Architecture: This architecture, also called ***1-address format architecture***, has an accumulator which is always used as one of the operands and as the destination register. The second operand is specified explicitly.

Microcontroller

2-address Format Architecture: Here, both operands are specified, but one of them is also used as the destination to store the result. Which register is used for this purpose depends on the processor in question, for example, the ATmega16 controller uses the first register as implicit destination, whereas the 68000 processor uses the second register.

Microcontroller

3-address Format Architecture: In this architecture, both source operands and the destination are explicitly specified. This architecture is the most flexible, but of course it also has the longest instruction size.

Microcontroller

- Table 2.1 shows the differences between the architectures when computing (A+B)*C. We assume that in the cases of the 2- and 3-address format, the result is stored in the first register. We also assume that the 2- and 3-address format architectures are load/store architectures, where arithmetic instructions only operate on registers. The last line in the table indicates where the result is stored.

Microcontroller

stack	accumulator	2-address format	3-address format
PUSH A	LOAD A	LOAD R1, A	LOAD R1, A
PUSH B	ADD B	LOAD R2, B	LOAD R2, B
ADD	MUL C	ADD R1, R2	ADD R1, R1, R2
PUSH C		LOAD R2, C	LOAD R2, C
MUL		MUL R1, R2	MUL R1, R1, R2
stack	accumulator	R1	R1

Table 2.1: Comparison between architectures.

Microcontroller

2. Execution Speed

The execution speed of an instruction depends on several factors. It is mostly influenced by the complexity of the architecture, so you can generally expect a CISC machine to require more cycles to execute an instruction than a RISC machine.

It also depends on the word size of the machine, since a machine that can fetch a 32 bit instruction in one go is faster than an 8-bit machine that takes 4 cycles to fetch such a long instruction.

Microcontroller

Finally, the oscillator frequency defines the absolute speed of the execution, since a CPU that can be operated at 20 MHz can afford to take twice as many cycles and will still be faster than a CPU with a maximum operating frequency of 8 MHz.

Microcontroller

3. Available Instructions

Of course, the nature of available instructions is an important criterion for selecting a controller. Instructions are typically parted into several classes:

Arithmetic-Logic Instructions: This class contains all operations which compute something, e.g., ADD, SUB, MUL, . . . , and logic operations like AND, OR, XOR, ... It may also contain bit operations like BSET (set a bit), BCLR (clear a bit), and BTST (test whether a bit is set). Bit operations are an important feature of the microcontroller, since it allows to access single bits without changing the other bits in the byte.

Microcontroller

- Shift operations, which move the contents of a register one bit to the left or to the right, are typically provided both as logical and as arithmetical operations.
- The difference lies in their treatment of the most significant bit when shifting to the right (which corresponds to a division by 2). Seen arithmetically, the msb is the sign bit and should be kept when shifting to the right. So if the msb is set, then an arithmetic right-shift will keep the msb set. Seen logically, however, the msb is like any other bit, so here a right-shift will clear the msb.

Microcontroller

- Note that there is no need to keep the msb when shifting to the left (which corresponds to a multiplication by 2). Here, a simple logical shift will keep the msb set anyway as long as there is no overflow. If an overflow occurs, then by not keeping the msb we simply allow the result to wrap, and the status register will indicate that the result has overflowed. Hence, an arithmetic shift to the left is the same as a logical shift.

Microcontroller

Example: Arithmetic shift

To illustrate what happens in an arithmetic shift to the left, consider a 4-bit machine. Negative numbers are represented in two's complement, so for example -7 is represented as binary 1001. If we simply shift to the left, we obtain 0010 = 2, which is the same as -14 modulo 16. If we had kept the msb, the result would have been 1010 = -6, which is simply wrong.

Shifting to the right can be interpreted as a division by two. If we arithmetically right-shift -4 = 1100, we obtain 1110 = -2 since the msb remains set. In a logical shift to the right, the result would have been 0110 = 6.

Microcontroller

Data Transfer: These operations transfer data between two registers, between registers and memory, or between memory locations. They contain the normal memory access instructions like LD (load) and ST (store), but also the stack access operations PUSH and POP.

Microcontroller

Program Flow: Here you will find all instructions which influence the program flow. These include jump instructions which set the program counter to a new address, conditional branches like BNE (branch if the result of the prior instruction was not zero), subroutine calls, and calls that return from subroutines like RET or RETI (return from interrupt service routine).

Microcontroller

Control Instructions: This class contains all instructions which influence the operation of the controller. The simplest such instruction is NOP, which tells the CPU to do nothing. All other special instructions, like power-management, reset, debug mode control, . . . also fall into this class.

Microcontroller

4. Addressing Modes

When using an arithmetic instruction, the application programmer must be able to specify the instruction's explicit operands.

Operands may be constants, the contents of registers, or the contents of memory locations. Hence, the processor has to provide means to specify the type of the operand.

Microcontroller

Hence, the processor has to provide means to specify the type of the operand. While every processor allows you to specify the above-mentioned types, access to memory locations can be done in many different ways depending on what is required.

So the number and types of addressing modes provided is another important characteristic of any processor. There are numerous addressing modes, but we will restrict ourselves to the most common ones.

Microcontroller

immediate/literal: Here, the operand is a constant. From the application programmer's point of view, processors may either provide a distinct instruction for constants (like the LDI —load immediate— instruction of the ATmega16), or require the programmer to flag constants in the assembler code with some prefix like #.

Microcontroller

register: Here, the operand is the register that contains the value or that should be used to store the result.

direct/absolute: The operand is a memory location.

register indirect: Here, a register is specified, but it only contains the memory address of the actual source or destination. The actual access is to this memory location.

Microcontroller

autoincrement: This is a variant of indirect addressing where the contents of the specified register is incremented either before (pre-increment) or after (post-increment) the access to the memory location. The post-increment variant is very useful for iterating through an array, since you can store the base address of the array as an index into the array and then simply access each element in one instruction, while the index gets incremented automatically.

Microcontroller

autodecrement: This is the counter-part to the autoincrement mode, the register value gets decremented either before or after the access to the memory location. Again nice to have when iterating through arrays.

displacement/based: In this mode, the programmer specifies a constant and a register. The contents of the register is added to the constant to get the final memory location. This can again be used for arrays if the constant is interpreted as the base address and the register as the index within the array.

Microcontroller

indexed: Here, two registers are specified, and their contents are added to form the memory address. The mode is similar to the displacement mode and can again be used for arrays by storing the base address in one register and the index in the other. Some controllers use a special register as the index register. In this case, it does not have to be specified explicitly.

Microcontroller

memory indirect: The programmer again specifies a register, but the corresponding memory location is interpreted as a pointer, i.e., it contains the final memory location. This mode is quite useful, for example for jump tables.

Table 2.2 shows the addressing modes in action. In the table, M[x] is an access to the memory address x, d is the data size, and #n indicates a constant.

Microcontroller

- As we have already mentioned, CISC processors feature more addressing modes than RISC processors, so CISC processors must construct more complex addressing modes with several instructions. Hence, if you often need a complex addressing mode, a CISC machine providing this mode may be the wiser choice.

Microcontroller

- Before we close this section, we would like to introduce you to a few terms you will often encounter:

1. An instruction set is called *orthogonal* if you can use every instruction with every addressing mode.
2. If it is only possible to address memory with special memory access instructions (LOAD, STORE), and all other instructions like arithmetic instructions only operate on registers, the architecture is called a *load/store architecture*.
3. If all registers have the same function (apart from a couple of system registers like the PC or the SP), then these registers are called *general-purpose registers*.

Microcontroller

addressing mode	example	result
immediate	ADD R1, #5	R1 ← R1 + 5
register	ADD R1, R2	R1 ← R1 + R2
direct	ADD R1, 100	R1 ← R1 + M[100]
register indirect	ADD R1, (R2)	R1 ← R1 + M[R2]
post-increment	ADD R1, (R2)+	R1 ← R1 + M[R2]
		R2 ← R2 + d
pre-decrement	ADD R1, −(R2)	R2 ← R2 − d
		R1 ← R1 + M[R2]
displacement	ADD R1, 100(R2)	R1 ← R1 + M[100 + R2]
indexed	ADD R1, (R2+R3)	R1 ← R1 + M[R2+R3]
memory indirect	ADD R1, @(R2)	R1 ← R1 + M[M[R2]]

Table 2.2: Comparison of addressing modes.

Microcontroller

3. **Memory**

- The register file is, of course, just a small memory embedded in the CPU.
- Also, we briefly mentioned data being transferred between registers and the *data memory*, and instructions being fetched from the *instruction memory*.
- Therefore, an obvious distinction of memory types can be made according to their function:

Microcontroller

- **Register File**: A (usually) relatively small memory embedded on the CPU. It is used as a scratchpad for temporary storage of values the CPU is working with - you could call it the CPU's short term memory.
- **Data Memory**: For longer term storage, generic CPUs usually employ an external memory which is much larger than the register file. Data that is stored there may be short-lived, but may also be valid for as long as the CPU is running. Of course, attaching external memory to a CPU requires some hardware effort and thus incurs some cost. For that reason, microcontrollers usually sport on-chip data memory.

Microcontroller

- **Instruction Memory**: Like the data memory, the instruction memory is usually a relatively large external memory (at least with general CPUs). Actually, with von-Neumann-architectures, it may even be the same physical memory as the data memory. With microcontrollers, the instruction memory, too, is usually integrated right into the MCU.

Microcontroller

- These are the most prominent uses of memory in or around a CPU. However, there is more memory in a CPU than is immediately obvious. Depending on the type of CPU, there can be:
- pipeline
- registers,
- caches,
- various buffers, and so on.

Microcontroller

- About memory embedded in an MCU: Naturally, the size of such on-chip memory is limited. Even worse, it is often not possible to expand the memory externally (in order to keep the design simple).
- However, since MCUs most often are used for relatively simple tasks and hence do not need excessive amounts of memory, it is prudent to include a small amount of data and instruction memory on the chip.
- Different members in a MCU family usually provide different amounts of memory, so you can choose a particular MCU which offers the appropriate memory space.

Microcontroller

- Now, the **functional distinction** of memory types made above is based on the way the memory is used. From a **programmer's perspective**, that makes sense.
- However, **hardware or chip designers** usually view memory rather differently: They prefer to distinguish according to the **physical properties** of the **electronic parts** the memory is made of.

Microcontroller

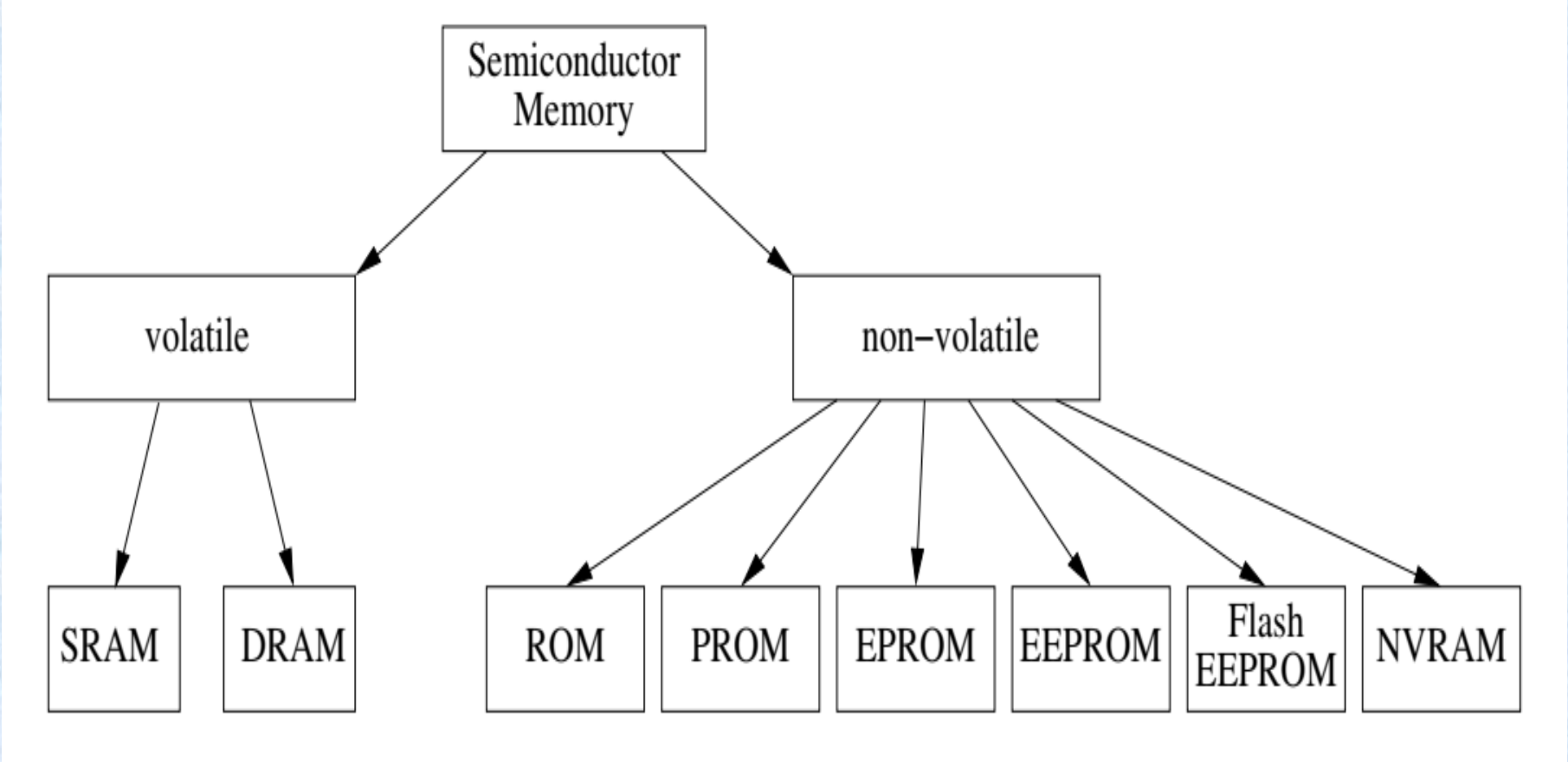


Fig 3.1 Types of Semiconductor Memory

Microcontroller

3.1 Volatile Memory

- As mentioned above, volatile memory retains its contents only so long as the system is powered on.
- Then why should you use volatile memory at all, when non-volatile memory is readily available?
- The problem here is that non-volatile memory is usually a **lot slower**, more involved to work with, and much more **expensive**.

Microcontroller

Static RAM

- Disregarding the era of computers before the use of integrated circuits, Static Random Access Memory (SRAM) was the first type of volatile memory to be widely used.
- An SRAM chip consists of an array of cells, each capable of storing one bit of information.
- To store a bit of information, a so-called **flip-flop** is used, which basically consists of **six transistors**.

Microcontroller

- Looking at Figure 3.2, you see that one SRAM cell has the following inputs and outputs:
- **Data In** D_{in} On this input, the cell accepts the one bit of data to be stored.
- **Data Out** D_{out} As the name implies, this output reflects the bit that is stored in the cell.

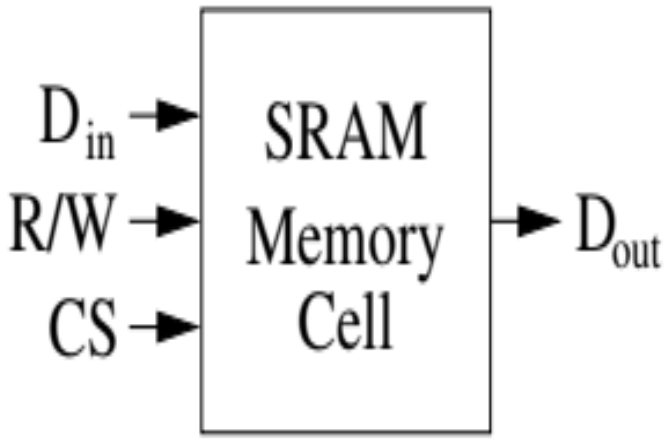


Fig 3.2 An SRAM cell as a Black box.

Microcontroller

- **Read/Write** $R/\overline{W}$ Via the logical value at this input, the type of access is specified: 0 means the cell is to be written to, i.e., the current state of D_{in} should be stored in the cell. 1 means that the cell is to be read, so it should set D_{out} to the stored value.
- **Cell Select** *CS* As long as this input is logical 0, the cell does not accept any data present at D_{in} and keeps its output D_{out} in a so-called high resistance state, which effectively disconnects it from the rest of the system. On a rising edge, the cell either accepts the state at D_{in} as the new bit to store, or it sets D_{out} to the currently stored value.

Microcontroller

- To get a useful memory, many such cells are arranged in a matrix as depicted in Figure 3.3.
- As you can see, all Dout lines are tied together. If all cells would drive their outputs despite not being addressed, a short between GND and VCC might occur, which would most likely destroy the chip.
- Therefore, the *CS* line is used to select one cell in the matrix and to put all other cells into their high resistance state.

Microcontroller

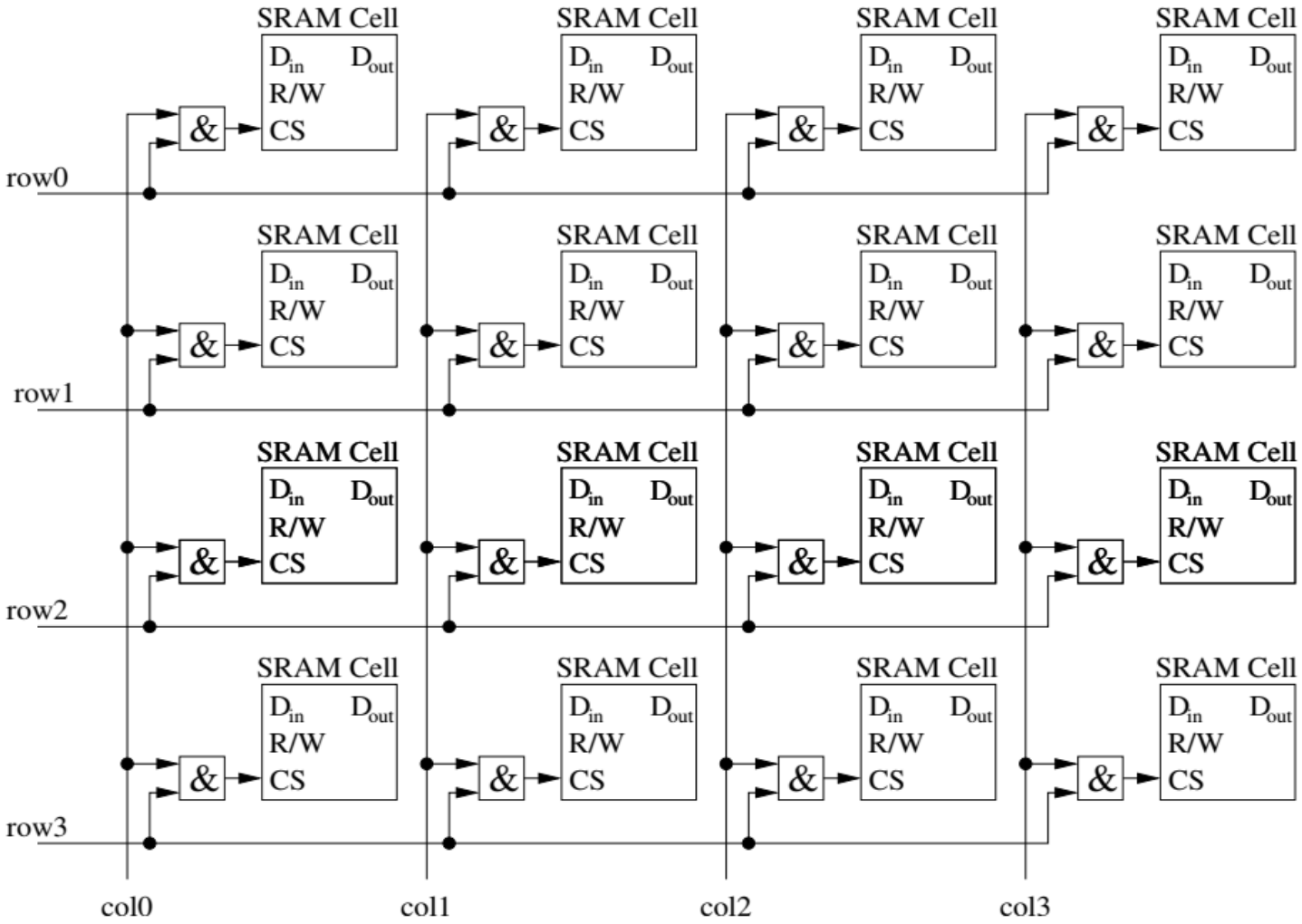


Fig. 3.3 A matrix of memory in an SRAM

Microcontroller

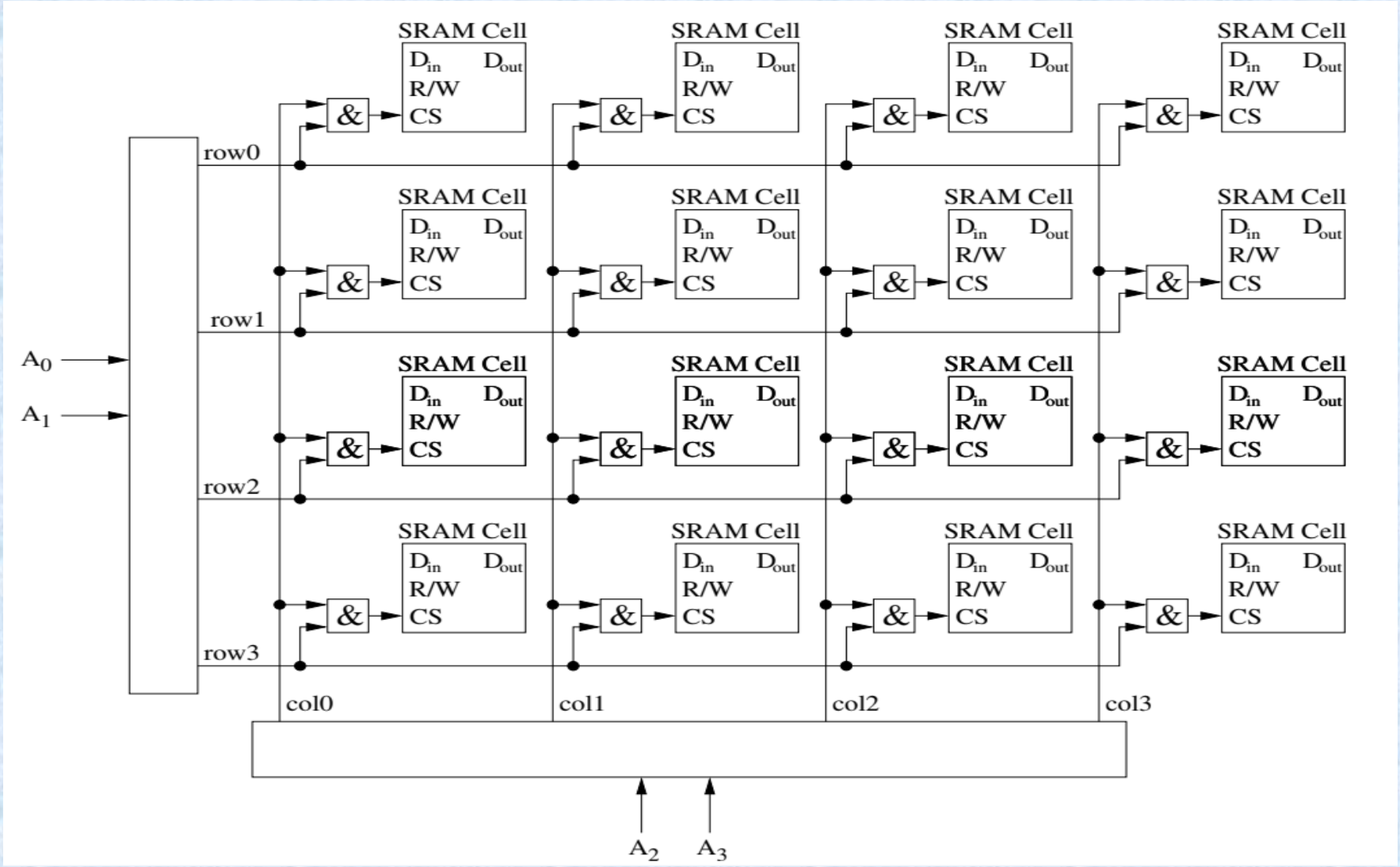


Fig. 3.4 Further reducing the number of external address pins.

Microcontroller

- So, instead of actually setting one of many rows, we just need the number of the row we wish to select, and the decoder produces the actual row lines. With that change, our 16Kx1 SRAM needs no more than 14 address lines.
- So much for the internals of a SRAM. Now, what do we actually see from the outside? Well, a SRAM usually has the following external connections (most of which you already know from the layout of one memory cell):

Microcontroller

- **Address Lines** $A0 \ldots An-1$ They are used to select one memory cell out of a total of 2^n cells.
- **Data In** (D_{in}) The function is basically the same as with one memory cell. For RAMs of width $n \geq 2$, this is actually a bus composed of n data lines.
- **Data Out** (D_{out}) Same function as in a single memory cell. Like D_{in}, for RAMs of width $n \geq 2$, this would be a bus.

Microcontroller

- **Chip Select** (*CS*) or **Chip Enable** (*CE*) This is what Cell Select was for the memory cell.
- **Read/Write** $R/\overline{W}$ Again, this works just like $R/\overline{W}$ in a memory cell.

Microcontroller

Dynamic RAM

- Obviously, we would like to get as much storage capacity as possible out of a memory chip of a certain size.
- Now, we already know that **SRAM** usually needs **six transistors** to store one single bit of information.
- If we could reduce the number of components needed – say, we only use half as much transistors –, then we would get about twice the storage capacity.

Microcontroller

- That is what was achieved with ***Dynamic Random Access Memory*** – DRAM: The number of **transistors** needed per bit of information was brought down to **one**.
- So at the same chip size, a DRAM has much larger storage capacity compared to an SRAM.
- Well, instead of using a lot of transistors to build flip-flops, one bit of **information** is **stored** in a **capacitor**.

Microcontroller

- They kind of work like little rechargeable batteries – you apply a voltage across them, and they store that voltage. Disconnect, and you have a loaded capacitor. Connect the pins of a loaded capacitor via a resistor, and an electrical current will flow, discharging the capacitor.

Microcontroller

- Well, the information is indeed stored in a capacitor, but in order to select it for reading or writing, a transistor is needed.
- If you want to store a **logical one**, you address the memory cell you want to access by **driving** the **transistor**.
- Then, you **apply a voltage**, which **charges** the **capacitor**.
- To store a **logical zero**, you **select** the **cell** and **discharge** the **capacitor**.

Microcontroller

- However, due to the flow of minimal currents through the non-perfect insulators on the chip (so-called *leakage currents*), the capacitor loses its charge, despite not being accessed.
- And since these capacitors are rather small, their capacity is accordingly small. This means that after loading the capacitor, the charge will unavoidably decrease. After some time (in the range of **10 to 100 ms**), the charge will be **lost**, and the information with it.

Microcontroller

- Well, they kind of handed the problem over to the users: By accessing DRAM, the information is refreshed (the capacitors are recharged). So DRAM has to be accessed every few milliseconds or so, else the information is lost.
- Often, the CPU does not need to access its RAM every cycle, but also has internal cycles to do its actual work. A DRAM refresh controller logic can use the cycles in between the CPUs accesses to do the refreshing.

Microcontroller

- **DRAM** has about four times larger storage capacity than **SRAM** at about the same cost and chip size. This means that **DRAM**s are available in larger capacities.
- Apart from the need for memory refresh, there is another severe disadvantage of **DRAM**: It is much slower than **SRAM**. However, due to the high cost of **SRAM**, it is just not an option for common desktop PCs. Therefore, numerous variants of **DRAM** access techniques have been devised, steadily increasing the speed of **DRAM** memory.
- In microcontrollers, you will usually find **SRAM**, as only moderate amounts of memory are needed, and the refresh logic required for **DRAM** would use up precious silicon area.

Microcontroller

3.2 Non-volatile Memory

- Contrary to SRAMs and DRAMs, non-volatile memories retain their content even when power is cut. But, as already mentioned, that advantage comes at a price: Writing non-volatile memory types is usually much slower and comparatively complicated, often downright annoying.

Microcontroller

1. ROM
2. PROM
3. EPROM
4. EEPROM: The EEPROM (Electrically Erasable and Programmable ROM) has all the advantages of an EPROM without the hassle. No special voltage is required for programming anymore, and – as the name implies – no more UV light source is needed for erasing. EEPROM works very similar to EPROM, except that the electrons can be removed from the floating gate by applying an elevated voltage.

- Of course, EEPROMs have their limitations, too: They endure a limited number of write/erase- cycles only (usually in the order of 100.000 cycles), and they do not retain their information indefinitely, either.
- EEPROMs are used quite regularly in microcontroller applications.

Microcontroller

5. Flash: Flash is a variant of EEPROM where erasing is not possible for each address, but only for larger blocks or even the entire memory (erased ‘in a flash’, so to speak).

- Therefore, Flash-EEPROMs often have a lower guaranteed write/erase cycle endurance compared to EEPROMs – about 1.000 to 10.000 cycles. This, too, makes Flash-EEPROMs cheaper.

Microcontroller

6. NVRAM: Finally, there is a type of memory that combines the advantages of volatile and non-volatile memories: Non-Volatile RAM (NVRAM). This can be achieved in different ways. One is to just add a small internal battery to an SRAM device, so that when external power is switched off, the SRAM still retains its content. Another variant is to combine a SRAM with an EEPROM in one package. Upon power-up, data is copied from the EEPROM to the SRAM. During operation, data is read from and written to the SRAM. When power is cut off, the data is copied to the EEPROM

Microcontroller

3.3 Accessing Memory

- Many microcontrollers come with on-chip program and data memory. Usually, the program memory will be of the Flash-EEPROM type, and the data memory will be composed of some SRAM and some EEPROM.
- How does a particular address translate in terms of the memory addressed? Basically, there are two methods:

Microcontroller

1. Each memory is addressed separately, see Figure 3.4 (e.g. ATmega16).

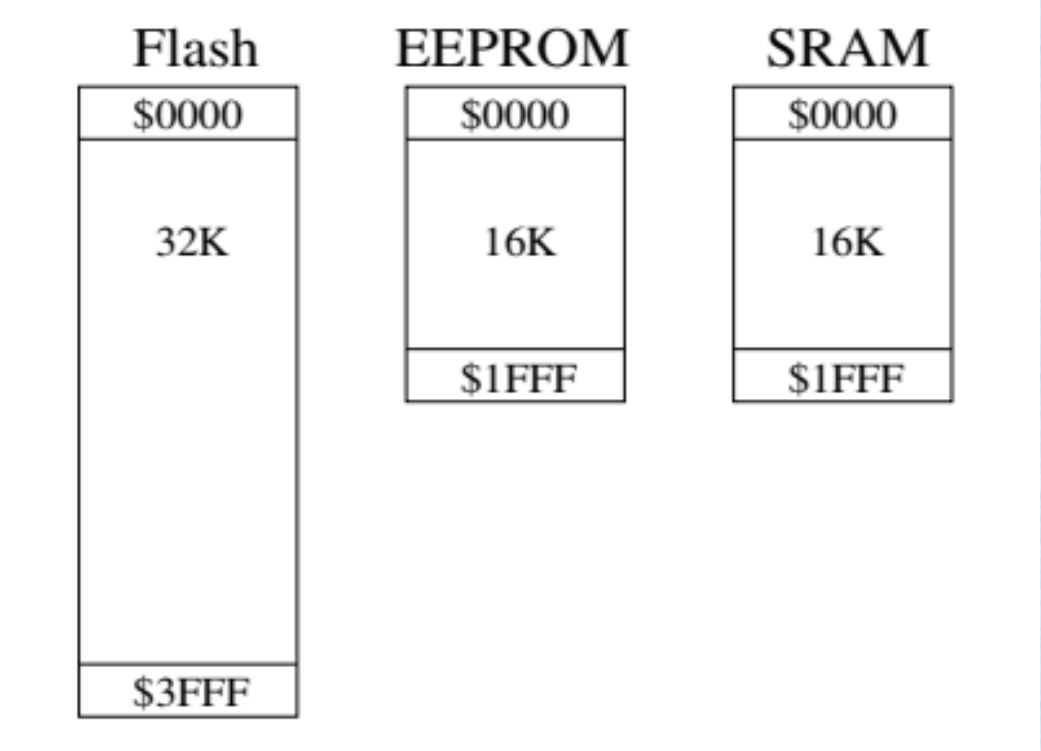


Fig. 3.4 Separate Memory Addressing.

Microcontroller

- The address ranges of the three different memory types can be the same. The programmer specifies which memory is to be accessed by using different access methods. E.g., to access EEPROM, a specific EEPROM-index register is used.

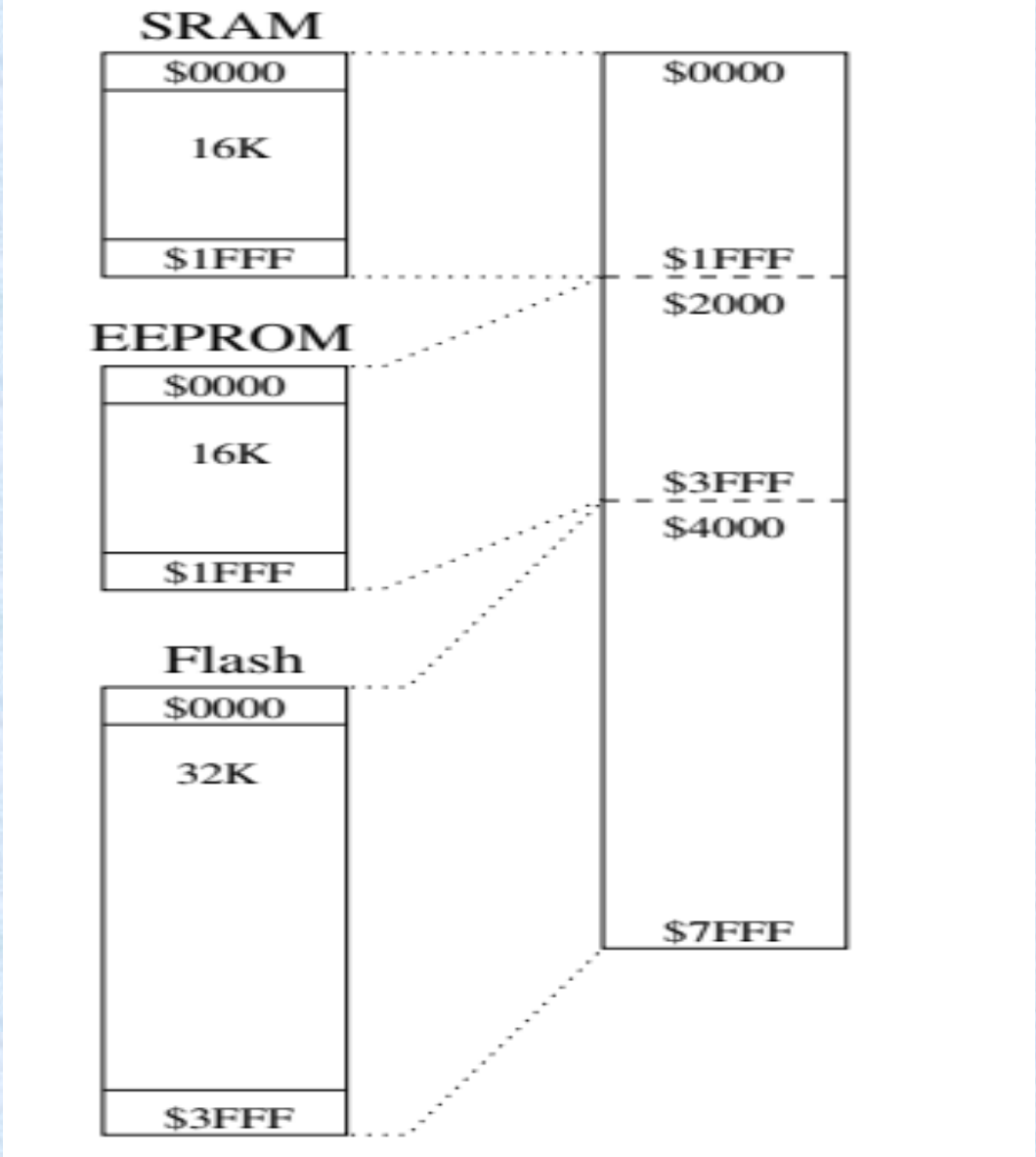
Microcontroller

2. All memory types share a common address range, see Figure 3.5 (e.g. HCS12).

- Here, the programmer accesses EEPROM in the same way as SRAM. The microcontroller uses the address to decide which memory the access goes to. For example, EEPROM could be assigned an address range of 0x1000 – 0x2000, while SRAM shows up in the range 0x2000 – 0x3000. Now, when the programmer accesses address 0x1800, the microcontroller knows that this is in the EEPROM range, and therefore it will access the EEPROM.

Microcontroller

Fig. 3.5 Different memory types mapped into one address range.



Microcontroller

- While this method is very straightforward, it is also inherently less safe: A wrong address can lead to the wrong type of memory being accessed. This would be especially dangerous if you were to inadvertently access the EEPROM instead of SRAM – with frequent access, the EEPROM could wear out in a matter of minutes.
- Separate memory addressing, on the other hand, comes with an implicit protection against access to the wrong type of memory.

Microcontroller

- When accessing byte-addressed memory word-wise, there is a special pitfall to be considered: Suppose a 16 bit controller writes a word (two bytes) into SRAM, say at address 0x0100. The word consists of a low and a high byte. Now, in what order are the bytes to be written? There are two variants: the low byte could go to 0x0100 and the high byte to the next address (0x0101), or the other way around. That is the problem of endianness:

Microcontroller

- **Big Endian**: Big Endian architectures store the high byte first. So, if you write the word 0x1234 to address 0x0100, the high byte 0x12 goes to address 0x0100, and the low byte 0x34 to address 0x0101. The name is derived from this order: The Big End of the word is stored first – therefore, it is called Big Endian.
- **Little Endian**: Little Endian architectures access memory the other way around (Little End of the word first). Here, the low byte is stored first. Writing 0x1234 at address 0x0100 on a little endian architecture writes 0x34 to address 0x0100 and 0x12 to address 0x0101.

Microcontroller

- Note carefully, however, that this difference in the ordering of high and low is only relevant on a byte level. The bits within a byte are numbered from right to left on both architectures. So, the least significant bit is always the rightmost one.

Microcontroller

4. **Interrupts**

- Microcontrollers tend to be deployed in systems that have to react to events. Events signify state changes in the controlled system and generally require some sort of reaction by the microcontroller.
- Reactions range from simple responses like incrementing a counter whenever a workpiece crosses a photoelectric barrier on the conveyor belt to time-critical measures like shutting down the system if someone reaches into the working area of a machine. Assuming that the controller can observe the event, that is, there is an input line that changes its state to indicate the event, there is still the question of how the controller should monitor the input line to ensure a proper and timely reaction.

Microcontroller

- It is of course possible to simply *poll* the input signal, that is, to periodically check for state changes.
- However, this polling has its drawbacks: Not only does it unnecessarily waste processor time if the event only occurs infrequently, it is also hard to modify or extend. After all, a microcontroller generally has a lot more to do than just wait for a single event, so the event gets polled periodically in such a way that the rest of the program can be executed as well.

Microcontroller

- Fortunately, the microcontroller itself offers a convenient way in the form of *interrupts*. Here, the microcontroller polls the signal and interrupts the main program only if a state change is detected.
- As long as there is no state change, the main program simply executes without any concerns about the event.
- As soon as the event occurs, the microcontroller calls an interrupt service routine (ISR) which handles the event.
- The ISR must be provided by the application programmer.

Microcontroller

4.1 Interrupt Control

- Two bits form the main interface to the interrupt logic of the microcontroller:

1. The *interrupt enable* (IE) bit is set by the application programmer to indicate that the controller should call an ISR in reaction to the event.
2. The *interrupt flag* (IF) bit is set by the microcontroller whenever the event occurs, and it is cleared either automatically upon entering the ISR or manually by the programmer.

Microcontroller

- Basically, the IF bit shows that the interrupt condition has occurred, whereas the IE bit allows the interrupt itself to occur.
- The IE and IF bits are generally provided for every interrupt source the controller possesses.
- Apart from the IE and IF bits, the controller will most likely offer additional control bits (interrupt mode) for some of its interrupt sources. They are used to select which particular signal changes should cause an interrupt (e.g., only a falling edge, any edge, . . .). It is sometimes even possible to react to the fact that an input signal has not changed. This is called a *level interrupt*.

Microcontroller

- Since it would be inconvenient and time-consuming to disable all currently enabled interrupts whenever the program code should not be interrupted by an ISR (atomic action), a microcontroller also offers one global interrupt enable bit which enables/disables all currently enabled interrupts.
- Hence, an ISR is only called if both the IE bit for the interrupt source and the global IE bit are enabled. Note that in the case of the global IE bit, “enabled” does not necessarily mean “set”, so always check whether the bit should be set or cleared to enable interrupts.

Microcontroller

- Disabling interrupts does not necessarily imply that you will miss events. The occurrence of an event is stored in its IF, regardless of whether the IE bit is set or not (this refers to both the global and local IE). So if an event occurs during a period when its interrupt was disabled, and this interrupt is later enabled again, then the corresponding ISR will be called, albeit somewhat belatedly. The only time you will miss events is when a second event occurs before the first one has been serviced.

Microcontroller

- Apart from the normal interrupts, which can be disabled, some controllers also offer a non-maskable interrupt (NMI), which cannot be disabled by the global IE bit. Such interrupts are useful for particularly important events, when the reaction to the event must not be delayed regardless of whether it affects the program or not.
- After a reset, interrupts are generally disabled both at the source and globally. However, the application programmer should be aware that the start-up code generated by the compiler may already have enabled the global IE bit before the application code begins its execution.

Microcontroller

Interrupt Vector Table

- Apart from enabling a given interrupt, the programmer must also have the means to tell the controller which particular interrupt service routine should be called. The mapping of interrupts to ISRs is achieved with the *interrupt vector table*, which contains an entry for each distinct *interrupt vector*.

Microcontroller

- An interrupt vector is simply a number associated with a certain interrupt. Each vector has its fixed address in the vector table, which in turn has a fixed base address in (program) memory. At the vector address, the application programmer has to enter either the starting address of the ISR (e.g., HCS12) or a jump instruction to the ISR (e.g., ATmega16).

Microcontroller

Example: ATmega16 Interrupt Vector Table

The following interrupt vector table has been taken from the Atmel ATmega16 manual, p. 43, but the program address has been changed to the byte address (the manual states the word address).

Vector No.	Prg. Addr.	Source	Interrupt Definition
1	\$000	RESET	External Pin, Power-on Reset, ...
2	\$004	INT0	External Interrupt Request 0
3	\$008	INT1	External Interrupt Request 1
4	\$00C	TIMER2 COMP	Timer/Counter 2 Compare Match
...	...	...	...

As you can see, the vector table starts at program address 0x0000 with the reset vector. Vector k has the address $4(k-1)$, and the controller expects a jump instructions to the appropriate ISR there. The ATmega16 has fixed interrupt priorities, which are determined by the vector number: The smaller the vector number, the higher the interrupt's priority.

Microcontroller

Interrupt Priorities

- Most controllers with many interrupts and a vector table use the interrupt vector as an indication to the priority. The ATmega16, for example, statically assigns the highest priority to the interrupt with the smallest interrupt vector. If the controller offers NMIs, they will most likely have the highest priority.
- Of course, a static assignment of priorities may not always reflect the requirements of the application program. In consequence, some controllers allow the user to dynamically assign priorities to at least some interrupts. Others enable the user to select within the ISR which interrupts should be allowed to interrupt the ISR.

Microcontroller

4.2 Interrupt Handling

- To summarize, from the detection of the event on, interrupt handling is executed in the following steps:

1. **Set interrupt flag**: The controller stores the occurrence of the interrupt condition in the IF.
2. **Finish current instruction**: Aborting half-completed instructions complicates the hardware, so it is generally easier to just finish the current instruction before reacting to the event. Of course, this prolongs the time until reaction to the event by one or more cycles.
3. Identify ISR: The occurrence of an event does not necessarily imply that an ISR should be called. If the corresponding IE bit is not set, then the user does not desire an interrupt. Furthermore, since the controller has several interrupt sources which may produce events simultaneously, more than one IF flag can be set. So the controller must find the interrupt source with the highest priority out of all sources with set IF and IE bits.

Microcontroller

4. **Call ISR**: After the starting address has been determined, the controller saves the PC etc. and finally executes the ISR.

- The whole chain of actions from the occurrence of the event until the execution of the first instruction in the ISR causes a delay in the reaction to the event, which is subsumed in the *interrupt latency*.
- The latency generally tends to be within 2-20 cycles, depending on what exactly the controller does before executing the ISR.

Microcontroller

4.3 Interrupt Service Routine

- The *interrupt service routine* contains the code necessary to react to the interrupt. This could include clearing the interrupt flag if it has not already been cleared, or disabling the interrupt if it is not required anymore.
- The ISR may also contain the code that reacts to the event that has triggered the interrupt. However, the decision what to do in the ISR and what to do in the main program is often not easy and depends on the situation, so a good design requires a lot of thought and experience.

Microcontroller

Interrupt or Polling

- First of all, you face the decision of whether you should poll or whether you should use an interrupt. This decision is influenced by several factors. Is this a large program, or just a small one? Are there other things to do in the main loop which are not related to the event you want to service? How fast do you have to react? Are you interested in the state or in the state change?

Microcontroller

- As an example, consider a button that is actuated by a human operator. As long as the button is pressed, a dc motor should be active.
- Checking every couple of milliseconds will easily be enough, and you can fit a lot of other code into a period of 1-10 ms.
- Assume that the impulses to advance the stepper motor come from a machine and are short. Here, interrupts are definitely the best solution, first because there is no bouncing involved, and mainly because polling may miss the short impulse, especially if there is other code that has to be executed as well.

Microcontroller

- To summarize, indications for using interrupts are
  - ❑ event occurs infrequently
  - ❑ long intervals between two events
  - ❑ the state change is important
  - ❑ short impulses, polling might miss them
  - ❑ event is generated by HW, no bouncing effects or spikes
  - ❑ nothing else to do in main, could enter sleep mode
- whereas polling may be the better choice if
  - ❑ the operator is human
  - ❑ no precise timing is necessary
  - ❑ the state is important
  - ❑ impulses are long
  - ❑ the signal is noisy
  - ❑ there is something else to do in main anyway, but not too much