Northern Technical University
Technical College / Kirkuk
Electronic and Control Techniques Eng. Dept.
Third Year

COMPUTER ARCHITECTURE

(MICROCONTROLLERS)

Fifth Edition 2019/2020

Prepared by
Prof. Abdulrahman Ikram Siddiq

Contents

References:

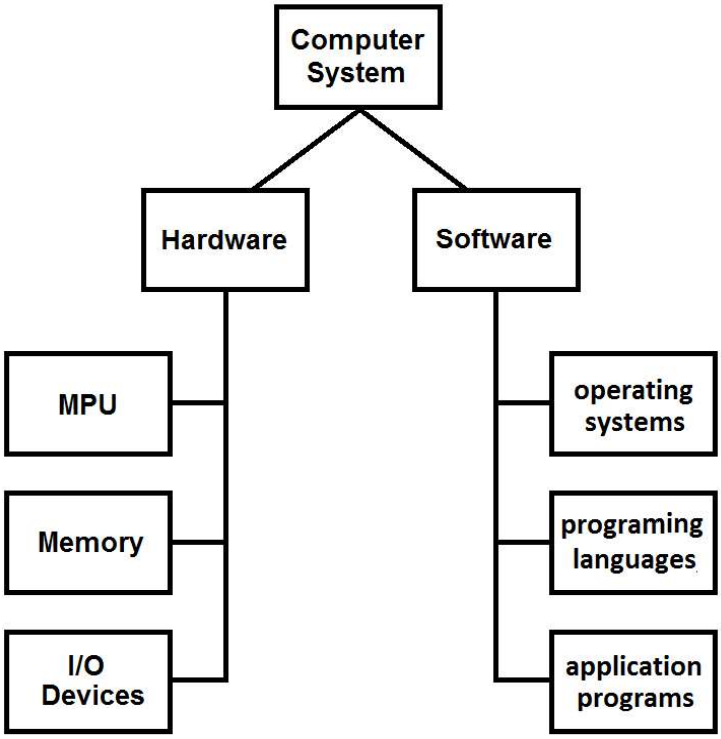
[1] DATASHEET of ATMEL 8-BIT Microcontroller with 4/8/16/32kbytes in-system programmable flash.

[2] Muhammad Ali Mazidi, The AVR Microcontroller and Embedded System using assembly and C, Prentice Hall, 2011.

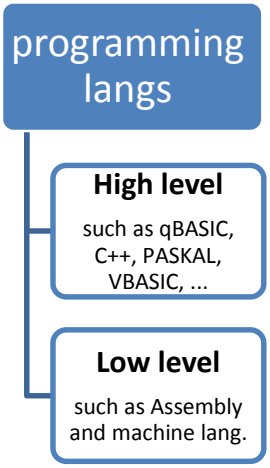
[3] A lot of similar references in the Internet.

Background

- **A computer system consists of hardware and software components.**
- **Hardware includes microprocessors, memories & input/output devices.**
- **Software includes the operating systems, programming languages and application programs.**



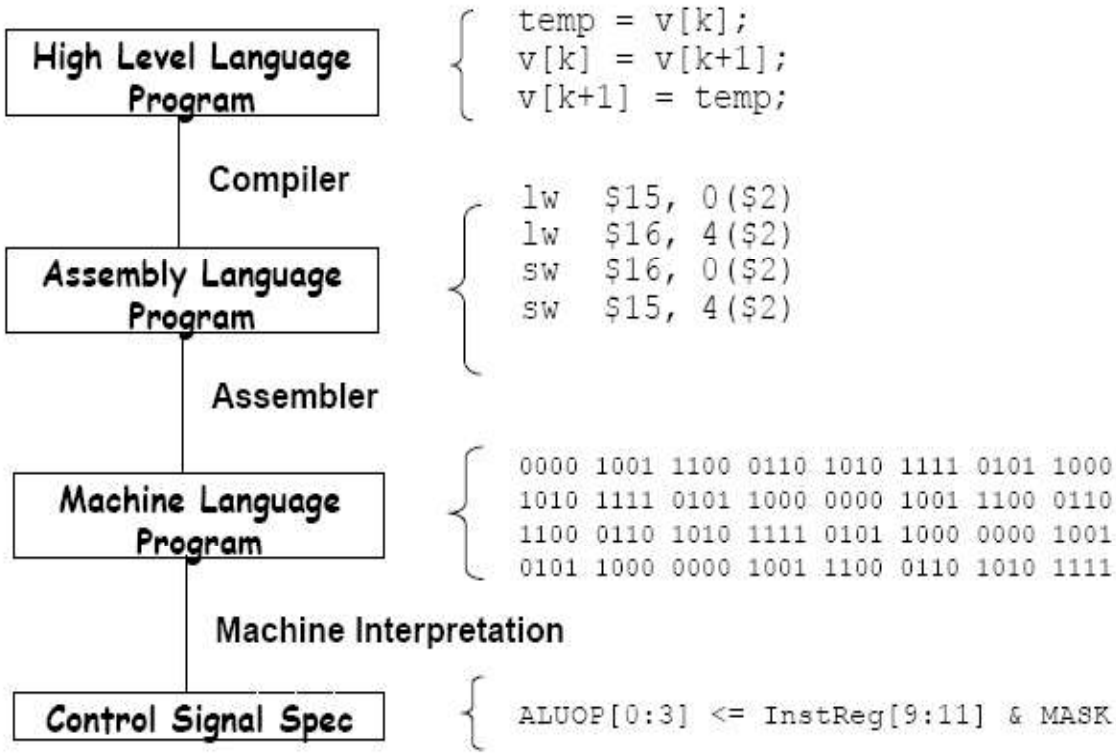
- **Programming languages may be classified as follows:**



A *High level language* is a set of instructions that let the programmer perform arithmetic, logical, data transfer, conditional branching, and I/O operations.

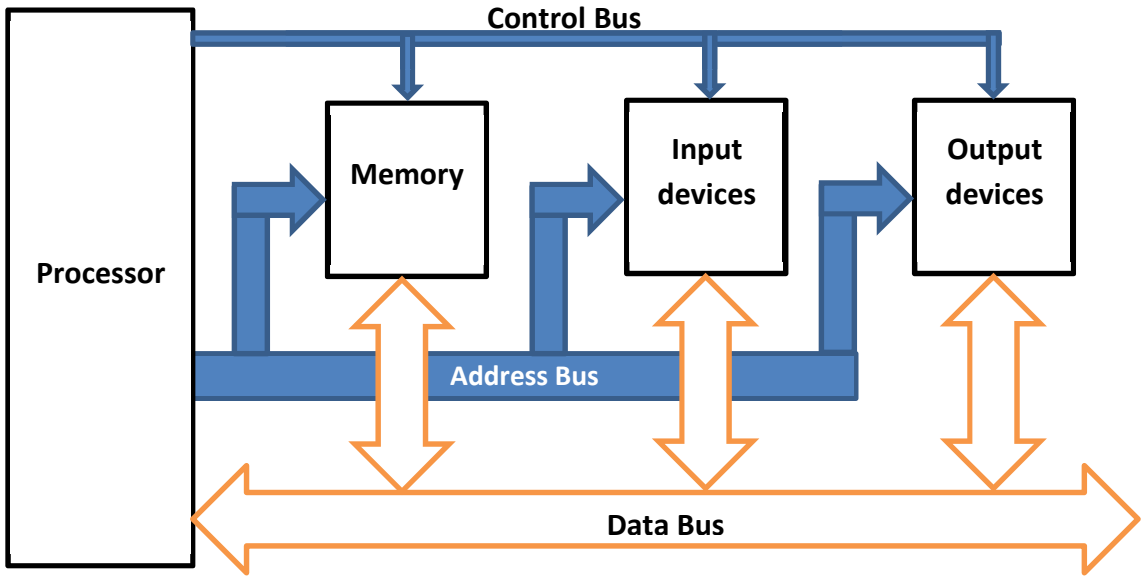
- **It is not necessary that the programmer has strict information about the internal structure of the computer system.**
- **High level programs are translated to the machine language of the specific processor using a software called "compiler", to be executed.**

- **A *Low level language* is a set of instructions that deal with the internal structure of the processor and the computer system, to perform arithmetic, logical, data transfer, conditional branching, and I/O operations.**



HARDWARE COMPONENTS

- **The function of the CPU is to execute (process) information stored in memory.**
- **The function of I/O devices, such as the keyboard and video monitor is to provide a means of communication between the user and the CPU.**
- **The CPU is connected with the memory and I/O devices through strips of wires called *busses*.**
- **The bus inside a computer allows carrying information from place to place, just as a street allows cars to carry people from place to place.**
- **In every computer system there are three types of buses: address bus, data bus and control bus.**



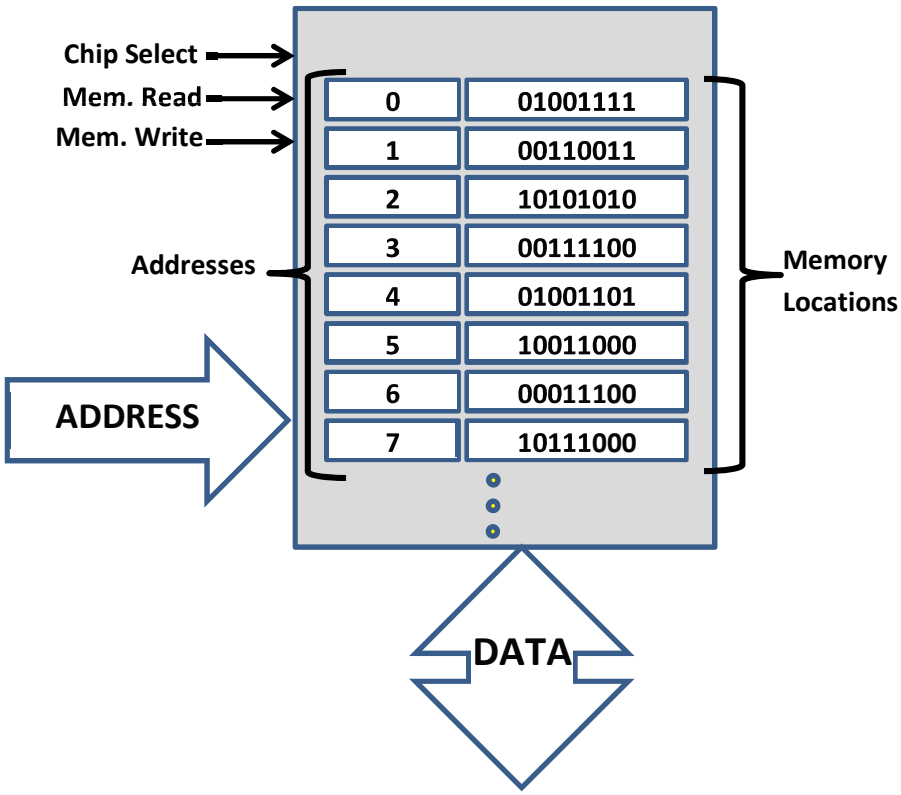
- ***Data bus:*** **is a set of bi-directional lines used to transfer data between different hardware components. The number of lines of a data bus depends on the CPU data word size (8 bits, 16 bits, …).**

- ***Address bus:*** **is a set of uni-directional lines used by the processor to transfer the address information to determine a memory location or an I/O device. The number of address lines depends on the capability of the processor. An n-bit address bus ⇒ 2^n different addresses. For 16-bit address bus, the number of addressable locations is $2^{16} = 2^{10}2^6 = 64$ Kbytes.**

- ***Control bus:*** **a number of wires used by the processor to control the operation of the other components of the system, such as memory read, memory write, I/O read, I/O write, … .**

MEMORY

A memory chip consists of a large number of memory locations. Each location stores binary data. The size of a memory location is typically 8 bits (1 Byte). The processor identifies a location by its address.



Memory Organization

- **Memory chips are organized into a number of locations within the IC.**
- **Each location can hold 1, 4, 8 or 16 bits depending on how the memory is internally designed and it is equal to the number of data pins on the chip.**
- **The number of storage locations = $2^{\text{number of address pins}}$.**
- **As an example, if memory chip has x data pins and y address pins then, there are 2^y storage location and each location in this memory stores x bits. Moreover, the total number of bits that can be stored in this memory is equal to ($2^y \times x$) bits.**

Memory Speed

- **One of the most important characteristics of a memory chip is the speed at which its data can be accessed.**
- **To access data, the address is presented to the address pins, the READ control signal is activated, and after a certain amount of time, the data appear at the data pins.**
- **The speed of the memory is usually referred to as the access time.**
- **The shorter is the access time, the better (and more expensive) is the memory chip.**
- **The access time of memory chips varies from few nanoseconds to hundreds nanoseconds, depending on the IC manufacturing technology.**

Example 0-12

A given memory chip has 12 address pins and 4 data pins. Find:
(a) the organization, and (b) the capacity.

Solution:

(a) This memory chip has 4,096 locations ($2^{12} = 4{,}096$), and each location can hold 4 bits of data. This gives an organization of 4,096 × 4, often represented as 4K × 4.
(b) The capacity is equal to 16K bits since there is a total of 4K locations and each location can hold 4 bits of data.

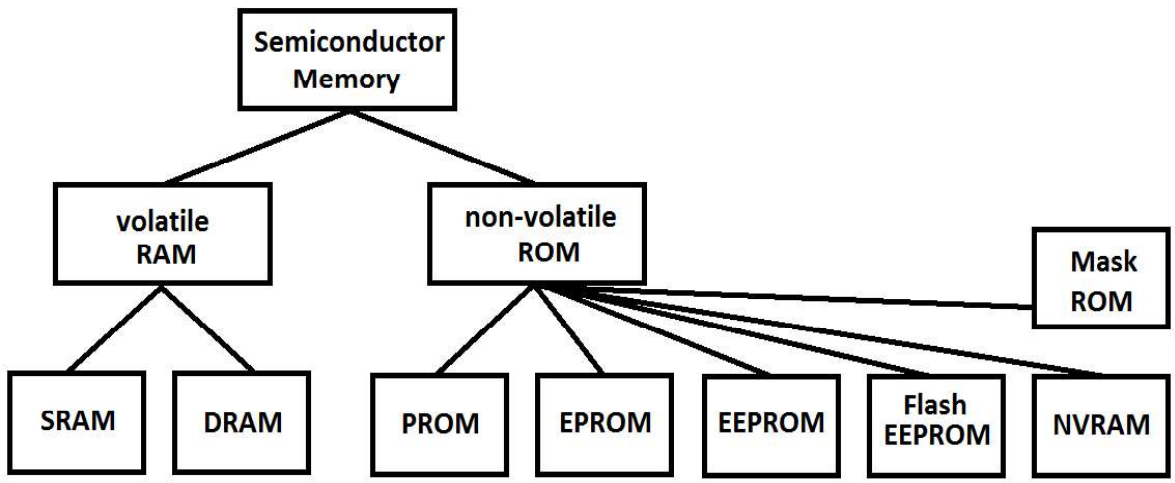
Example 0-13

A 512K memory chip has 8 pins for data. Find:
(a) the organization, and (b) the number of address pins for this memory chip.

Solution:

(a) A memory chip with 8 data pins means that each location within the chip can hold 8 bits of data. To find the number of locations within this memory chip, divide the capacity by the number of data pins. 512K/8 = 64K; therefore, the organization for this memory chip is 64K × 8.
(b) The chip has 16 address lines since $2^{16} = 64K$.

Memory Classification



ROM (Read Only Memory)

- ROM is a type of memory that does not lose its contents when the power is turned off.
- For this reason, ROM is also known as non-volatile memory.
- There are many types of ROM, such as, PROM, EPROM, EEPROM, Flash EPROM, and Mask ROM.

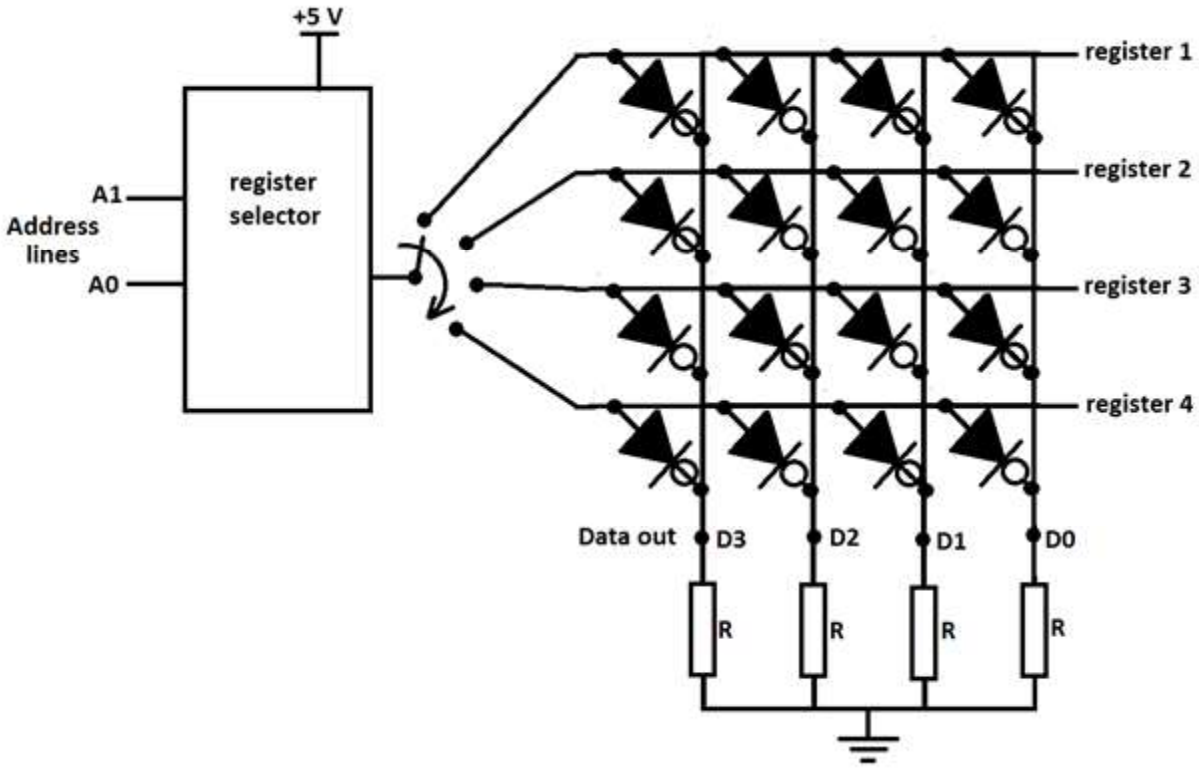
PROM (Programmable ROM) or OTP

- The user can burn information in to this type of ROM.
- For every bit of the PROM, there exists a fuse. If the fuse is burned, then a 0 is stored. If the fuse is left as it is, then a 1 is stored.
- If the information burned in to a PROM is wrong, then the PROM must be discarded. Therefore, it is also called One Time Programmable (OTP) ROM.
- Programming PROM (burning PROM) requires an equipment called ROM programmer or ROM burner.

Example: Draw the (4x4) diode matrix of a PROM. The contents are according to the given table.

register	A1	A0	Data
1	0	0	1011
2	0	1	1000
3	1	0	0101
4	1	1	1010

The PROM looks like this:



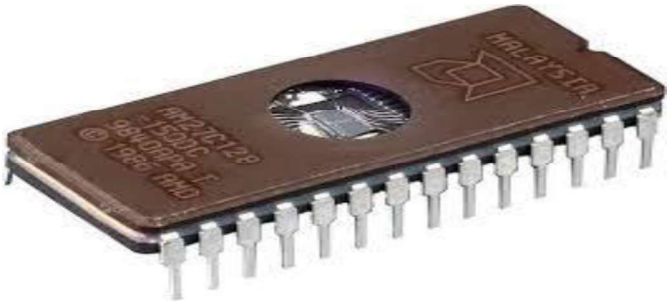
Homework: Repeat for (4x6) and (4x8) PROMs.

EPROM (Erasable Programmable ROM) or UV-EPROM

- EPROM was invented to allow changes in the contents of PROM after it is programmed.
- EPROM can be programmed and erased thousands of times.
- A widely used EPROM is called UV-EPROM where UV stands for ultraviolet.
- A UV-EPROM can be eased by shining UV radiation through a glass window for about 20 minutes.
- The program/erase cycle of UV-EPROM is 1000 times.

To program an EPROM, the following steps must be taken:

1. Erase the memory chip by removing it from its socket on the system board and placing it in EPROM eraser equipment to expose it to UV radiation for 15-20 minutes.
2. Place the memory chip in to another equipment called EPROM programmer to load the program by applying a 12.5V or higher dc voltage on the V_{pp} pin.
3. Place the memory chip into its socket on the system board.



Some UV-EPROM Chips

Part #	Capacity	Org.	Access	Pins	V_{PP}
2716	16K	2K × 8	450 ns	24	25 V
2732	32K	4K × 8	450 ns	24	25 V
2732A-20	32K	4K × 8	200 ns	24	21 V
27C32-1	32K	4K × 8	450 ns	24	12.5 V CMOS
2764-20	64K	8K × 8	200 ns	28	21 V
2764A-20	64K	8K × 8	200 ns	28	12.5 V
27C64-12	64K	8K × 8	120 ns	28	12.5 V CMOS
27128-25	128K	16K × 8	250 ns	28	21 V
27C128-12	128K	16K × 8	120 ns	28	12.5 V CMOS
27256-25	256K	32K × 8	250 ns	28	12.5 V
27C256-15	256K	32K × 8	150 ns	28	12.5 V CMOS
27512-25	512K	64K × 8	250 ns	28	12.5 V
27C512-15	512K	64K × 8	150 ns	28	12.5 V CMOS
27C010-15	1024K	128K × 8	150 ns	32	12.5 V CMOS
27C020-15	2048K	256K × 8	150 ns	32	12.5 V CMOS
27C040-15	4096K	512K × 8	150 ns	32	12.5 V CMOS

Example:

For the ROM chip 27128, find the number of data and address pins.

Solution:

The 27128 has a capacity of 128K bits.

It has (16K ×8) organization (all ROMs have 8 data pins).

The number of data pins of 27128 is 8.

The number of address pins is 14 since $16K=2^{14}$.

EEPROM (Electrically Erasable Programmable ROM) or E^2PROM

EEPROM has several differences with EPROM, such as:

- The erasure method is electrical.
- In EPROM, one can select which byte to erase, in contrast to UV-EPROM in which the entire contents are erased.
- EEPROM can be erased and programmed while it is still in the system board.
- The cost per bit of EEPROM is much higher than UV-EPROM.
- The program/erase cycle of EEPROM is 100000 times.

Flash EEPROM

- Since the early 1990s of the 20^{th} century, Flash EEPROM has become a popular user-programmable memory chip because

its **entire contents** can be electrically erased in less than a second (flash).

- In recent Flash memories, the contents are divided into blocks, and the erasure can be done block by block.
- Flash memory can be erased and programmed while it is in its socket on the system board.
- Flash memory is widely used as the BIOS ROM of the PC, and as a hard disk mass storage medium (SSD = Solid State Drive)
- The program/erase cycle of Flash EEPROM is 100000 times.

Some EEPROM and Flash Chips

EEPROMs

Part No.	Capacity	Org.	Speed	Pins	V_{PP}
2816A-25	16K	2K × 8	250 ns	24	5 V
2864A	64K	8K × 8	250 ns	28	5 V
28C64A-25	64K	8K × 8	250 ns	28	5 V CMOS
28C256-15	256K	32K × 8	150 ns	28	5 V
28C256-25	256K	32K × 8	250 ns	28	5 V CMOS

Flash

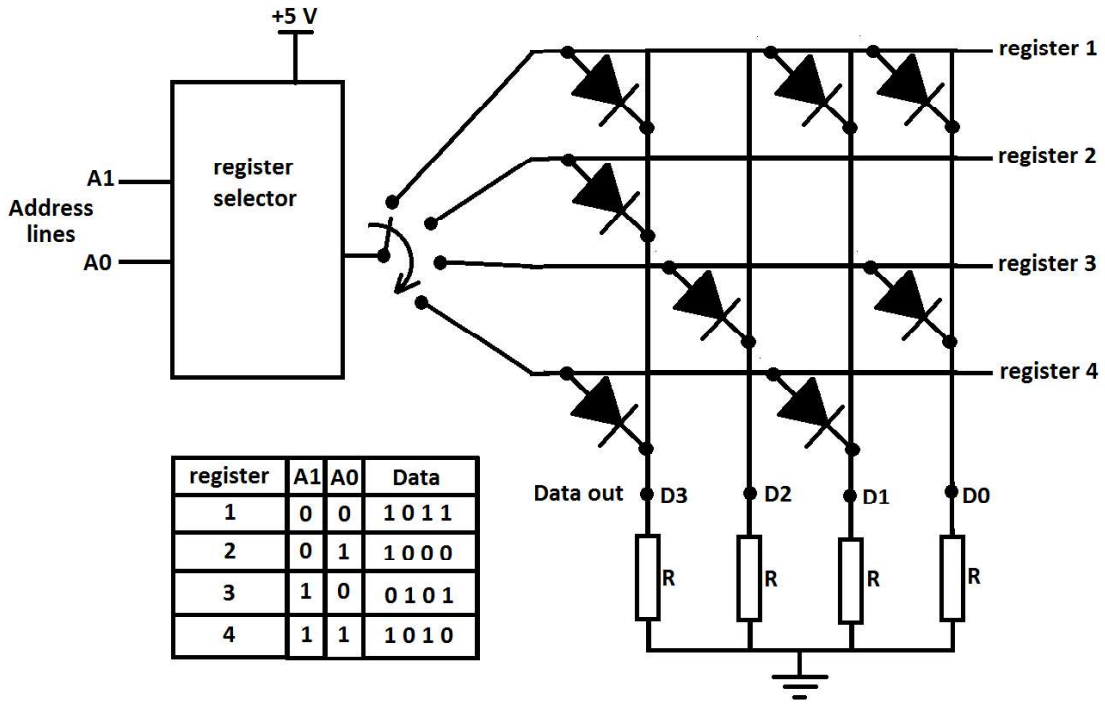
Part No.	Capacity	Org.	Speed	Pins	V_{PP}
28F256-20	256K	32K × 8	200 ns	32	12 V CMOS
28F010-15	1024K	128K × 8	150 ns	32	12 V CMOS
28F020-15	2048K	256K × 8	150 ns	32	12 V CMOS

Mask ROM

It is the kind of ROM that its contents are programmed by the IC manufacturer. It is not a user programmable memory.

The main advantage of Mask ROM is its low cost.

Example: The diode matrix of a (4x4) Mask ROM is shown below.



register	A1	A0	Data
1	0	0	1 0 1 1
2	0	1	1 0 0 0
3	1	0	0 1 0 1
4	1	1	1 0 1 0

Question: What is the difference between PROM and Mask ROM?

Homework: Repeat the above example for (4x6) and (4x8) MROMs.

RAM (Random Access Memory)

- RAM memory is called volatile since cutting off the power to the memory IC results in the loss of data.
- Sometimes, it is referred to RAM as R/WM (Read/Write Memory).
- There are three types of RAM:
    - Static RAM (SRAM)
    - Dynamic RAM (DRAM)
    - Non-vilatile RAM (NV-RAM)

SRAM

The storage cells are made of flip-flops, and each flip-flop requires at least 4 transistors to build. A storage cell holds a single bit.

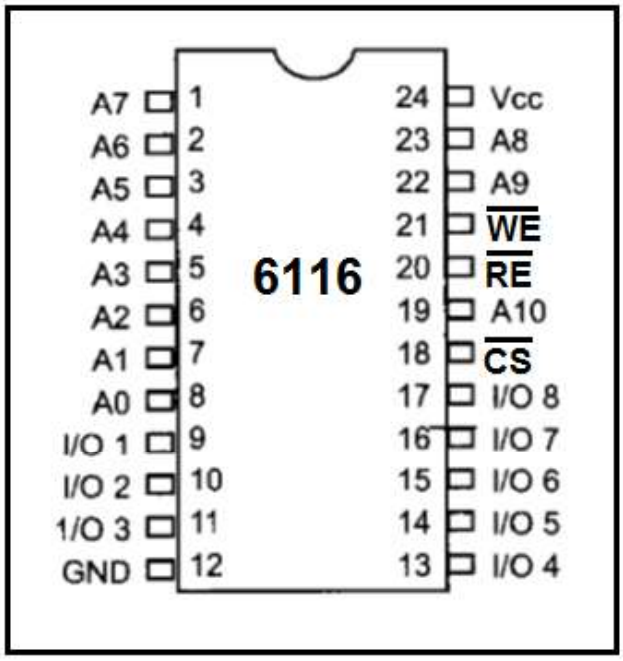
Some SRAM and NV-RAM Chips

SRAM

Part No.	Capacity	Org.	Speed	Pins	V_{PP}
6116P-1	16K	2K × 8	100 ns	24	CMOS
6116P-2	16K	2K × 8	120 ns	24	CMOS
6116P-3	16K	2K × 8	150 ns	24	CMOS
6116LP-1	16K	2K × 8	100 ns	24	Low-power CMOS
6116LP-2	16K	2K × 8	120 ns	24	Low-power CMOS
6116LP-3	16K	2K × 8	150 ns	24	Low-power CMOS
6264P-10	64K	8K × 8	100 ns	28	CMOS
6264LP-70	64K	8K × 8	70 ns	28	Low-power CMOS
6264LP-12	64K	8K × 8	120 ns	28	Low-power CMOS
62256LP-10	256K	32K × 8	100 ns	28	Low-power CMOS
62256LP-12	256K	32K × 8	120 ns	28	Low-power CMOS

Example:

For the RAM chip 6116, find the number of data and address pins.



Solution:

- The 6116 has a capacity of 16K bits.
- It has (2K ×8) organization.
- The 6116 has 8 I/O data pins.
- The number of address pins is 11 since $2K=2^{11}$.
- Also, the SRAM has three pins for the following control signals:

Chip Select (-CS) to activate or deactivate the memory IC.

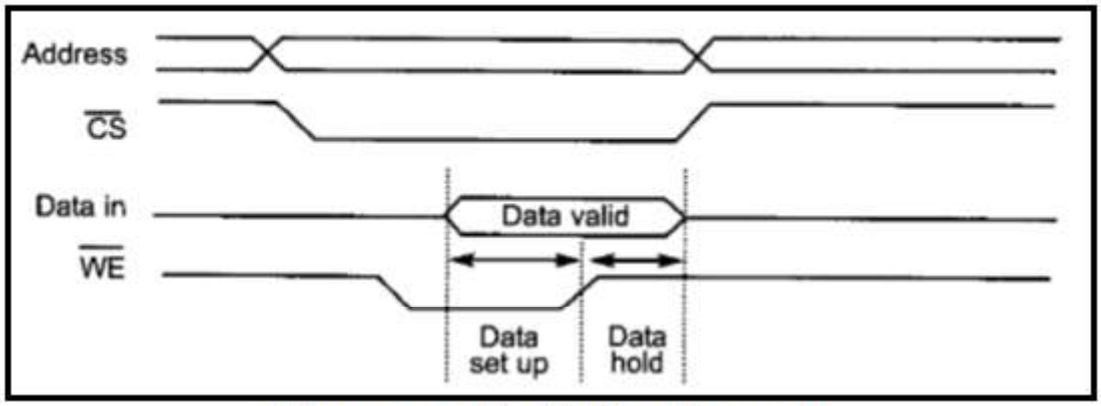
Write Enable (-WE) to enable writing (input) data into the RAM.

Read Enable (-RE) to enable reading (output) data from the RAM.

Note that all of these control signals are active low digital signals.

<u>To write data into the 6116 SRAM:</u>

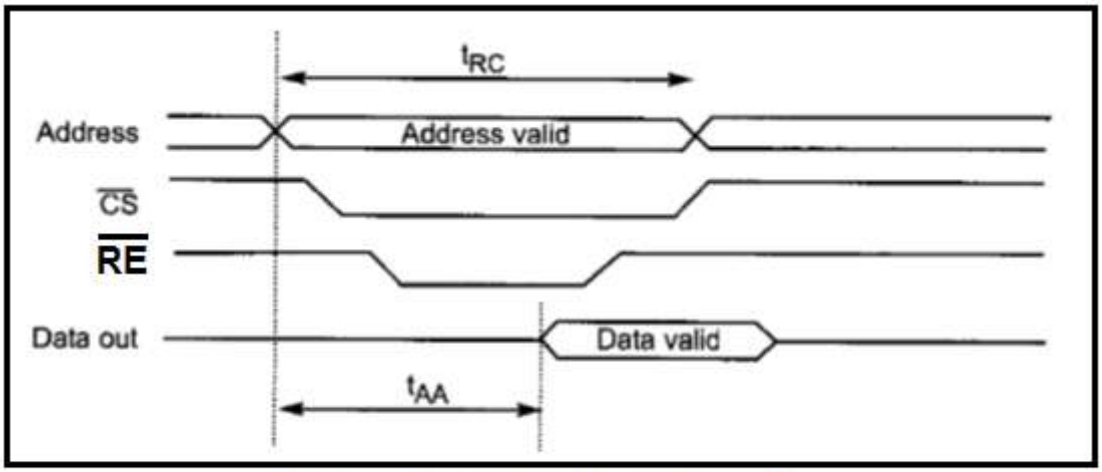
1. Provide the address to pins A_0 to A_{10}.
2. Activate the –CS pin.
3. Make –WE=0 and –RE=1.
4. Provide the data to pins I/O_0=I/O_7.
5. Make –WE=1, and the data will be written into SRAM on the positive edge of the –WE signal.



Memory write timing diagram

To read data from the 6116 SRAM:

1. Provide the address to pins A_0 to A_{10}.This is the start of the access time t_{AA}.
2. Activate the –CS pin.
3. Make –WE=1 and –RE=0. On the negative edge of –RE, the data will appear at the data pins D_0-D_7.



Memory read timing diagram

NV-RAM

- Like other RAMs, the NV-RAM allows the CPU to read and write to it, but when the power is turned off the contents are not lost.
- NV-RAM combines the best of RAM and ROM.
- The NV-RAM chip internally is made of the following components:
    1. Very low power consumption SRAM cells made of CMOS transistors.
    2. An internal lithium battery as a backup energy source.
    3. UPS control circuitry.

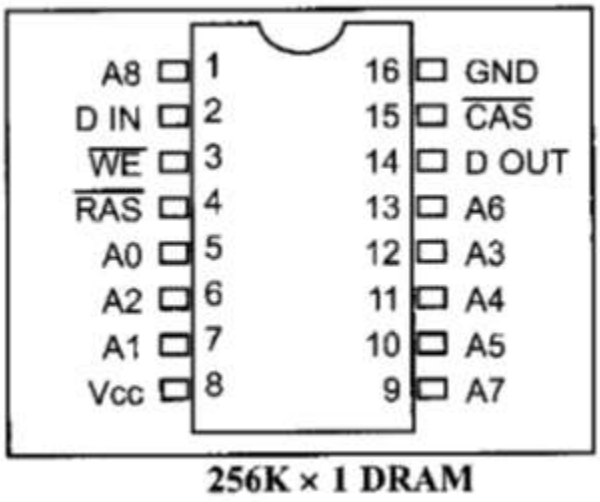
NV-RAM from Dallas Semiconductor

Part No.	Capacity	Org.	Speed	Pins	V_{PP}
DS1220Y-150	16K	2K × 8	150 ns	24	
DS1225AB-150	64K	8K × 8	150 ns	28	
DS1230Y-85	256K	32K × 8	85 ns	28	

DRAM

- In 1970, Intel Corporation introduced the first dynamic RAM. Its density (capacity) was 1024 bits.
- The memory cell in DRAM is a capacitor. A charged capacitor holds 1 and a discharged capacitor holds 0.
- To solve the charge leakage problem, a refreshment circuitry is used. The capacitors are refreshed periodically.
- The memory cell is smaller than SRAM.
- It has a higher density and smaller size than SRAM.
- It is cheaper and lower power consumption than SRAM.
- The data in DRAM cannot be accessed while it is being refreshed.

DRAM Organization



256K × 1 DRAM

- The DRAM can have ×1, ×4, ×8, or ×16 memory organization.
- In DRAM memory chips, the data pins are also called I/O. In some DRAMs there re separate D_{in} and D_{out} pins.
- The memory cells are arranged as a matrix. Therefore, half of the number of address lines is needed. The address pins are used to determine the row (when RAS is active) and the column (when CAS is active) of the intended memory cell. For example, the 256K×1 DRAM chip has nine address pins (A_0-A_8), with RAS and CAS.

Example: Find the number of address pins in (16K×4) SRAM and (16K×4) DRAM.

SRAM: 16K= 2^{14}, Address pins =14.

DRAM: 16K= 2^{14}, Address pins =$\frac{14}{2} = 7$ with RAS and CAS.

Some DRAMs

Part No.	Speed	Capacity	Org.	Pins
4164-15	150 ns	64K	64K × 1	16
41464-8	80 ns	256K	64K × 4	18
41256-15	150 ns	256K	256K × 1	16
41256-6	60 ns	256K	256K × 1	16
414256-10	100 ns	1M	256K × 4	20
511000P-8	80 ns	1M	1M × 1	18
514100-7	70 ns	4M	4M × 1	20

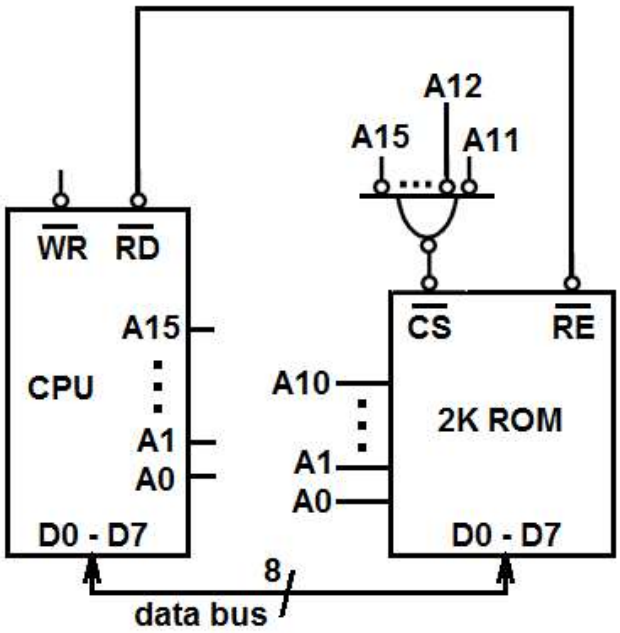
Logic Gate Address Decoding

Example1:

Connect a 2KByte ROM with an 8-bit data bus/16-bit address bus CPU.

Solution:

$2K=2^{11}$, then there are 11 direct address lines (A_0-A_{10}). The remainder of the 16 address lines (A_{11}-A_{15}) are connected to the NAND as shown in the figure.



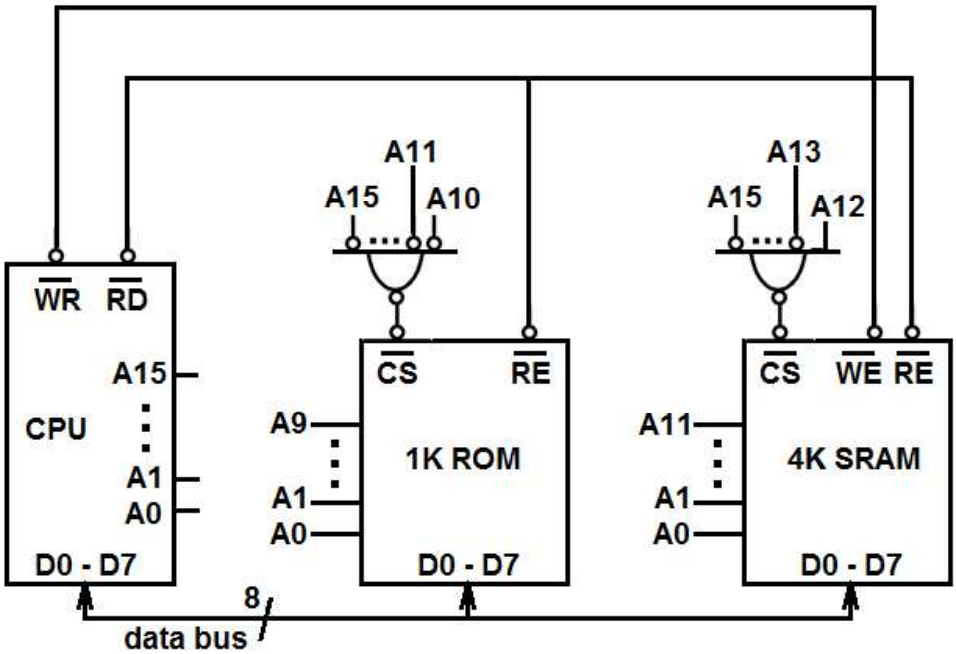
The address range of the locations is: 0x0000 to 0x07FF.

	A_{15}	A_{14}	A_{13}	A_{12}	A_{11}	A_{10}	A_9	A_8	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0
First location	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Last location	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1

Example2:

Connect a 1KByte ROM and a 4KByte SRAM with an 8-bit data bus/16-bit address bus CPU.

Solution:



The address range of the ROM is: 0x0000 to 0x03FF.

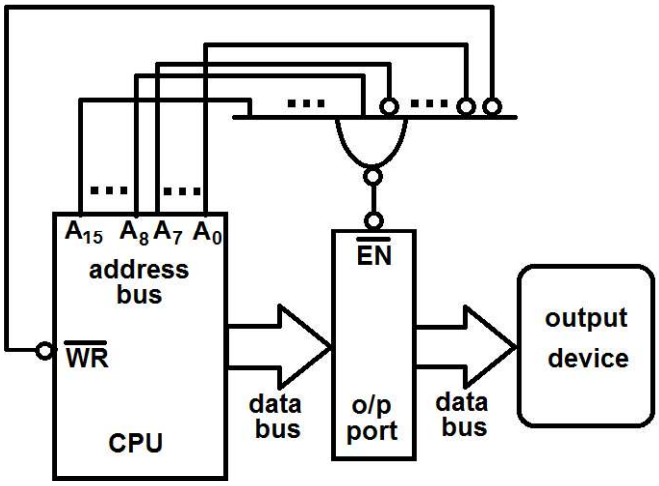
	A_{15}	A_{14}	A_{13}	A_{12}	A_{11}	A_{10}	A_9	A_8	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0
First location	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Last location	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1

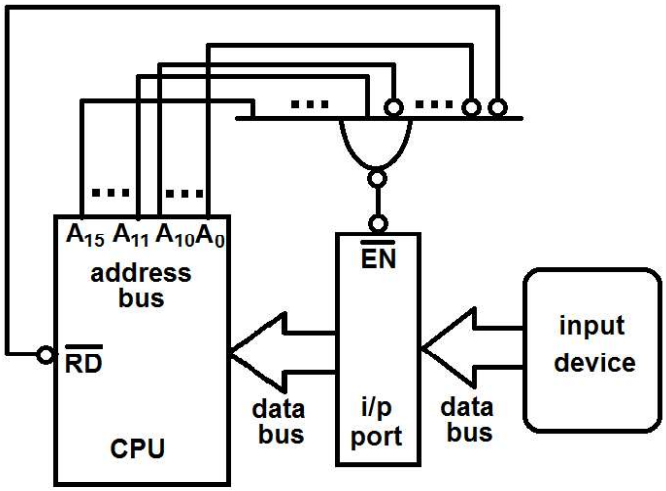
The address range of the SRAM is: 0x1000 to 0x1FFF

	A_{15}	A_{14}	A_{13}	A_{12}	A_{11}	A_{10}	A_9	A_8	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0
First location	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
Last location	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1

I/O Port Interfacing

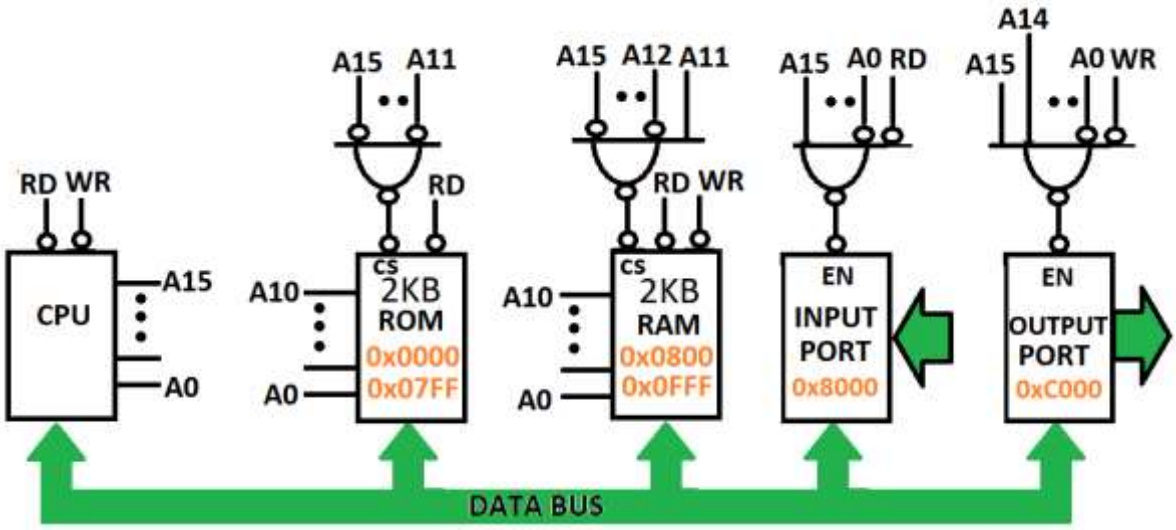
- **The data flow between the CPU and the I/O devices is performed through connection points called ports.**
- **The ports are under the control of the CPU.**
- **The CPU identifies (activates) an input or output device by placing its address on the address bus and enabling the associated port. Then, the data flows through this port.**
- **For example, in the figures below, the address of the output device is 0xFF00 and the input device is 0xF000.**





Example: Design a microcomputer system with the following components:

1. CPU (8-bit data bus and 16-bit address bus).
2. 2 Kbyte ROM.
3. 2 Kbytes RAM.
4. One input port and one output port.



CLASSWORK: Design, draw and specify the addresses of a microcomputer system with the following components:

- CPU (8-bit data bus and 16-bit address bus)
- 1 KB ROM and 4 KB RAM
- One input port and one output port (the ports are memory mapped)

Microcontroller versus general-purpose microprocessor

What is the difference between a microprocessor and a microcontroller? By microprocessor is meant the general-purpose microprocessors such as Intel's x86 family (8086, 80286, 80386, 80486, and the Pentium) or Motorola's PowerPC family. These microprocessors contain no RAM, no ROM, and no I/O ports on the chip itself. For this reason, they are commonly referred to as *general-purpose microprocessors*. See Figure 1-1.

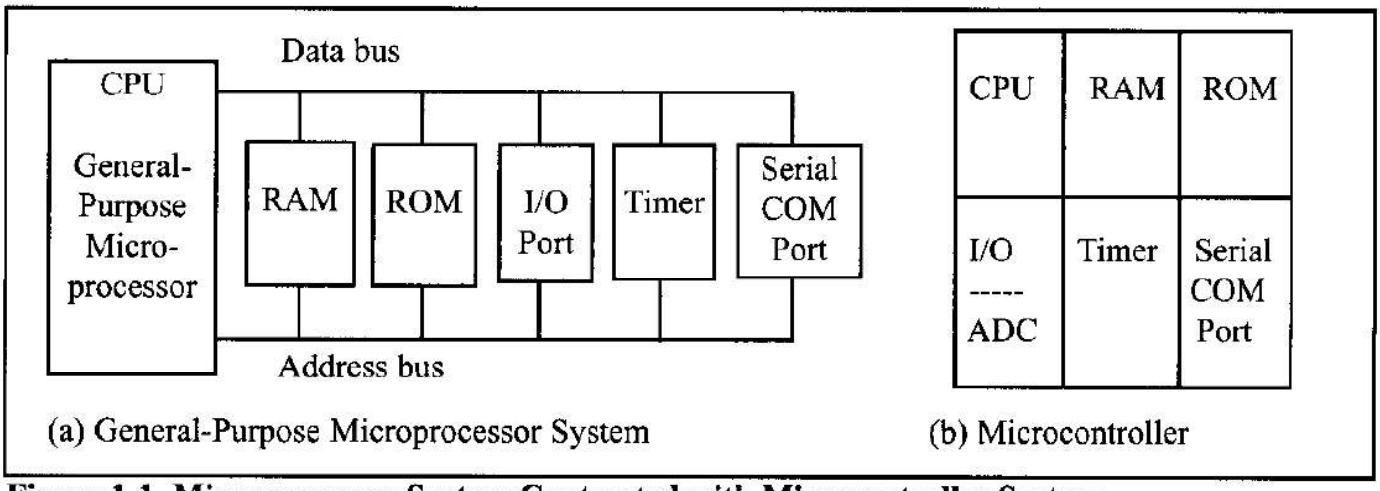


Figure 1-1. Microprocessor System Contrasted with Microcontroller System

A system designer using a general-purpose microprocessor such as the Pentium or the PowerPC must add RAM, ROM, I/O ports, and timers externally to make them functional. Although the addition of external RAM, ROM, and I/O ports makes these systems bulkier and much more expensive, they have the advantage of versatility, enabling the designer to decide on the amount of RAM, ROM, and I/O ports needed to fit the task at hand. This is not the case with microcontrollers. A microcontroller has a CPU (a microprocessor) in addition to a fixed amount of RAM, ROM, I/O ports, and a timer all on a single chip. In other words, the processor, RAM, ROM, I/O ports, and timer are all embedded together on one chip; therefore, the designer cannot add any external memory, I/O, or timer to it. The fixed amount of on-chip ROM, RAM, and number of I/O ports in microcontrollers makes them ideal for many applications in which cost and space are critical.

AVR MICROCONTROLLERS

- **The AVR is a modified Harvard architecture 8-bit RISC single-chip microcontroller, which was developed by Atmel in 1996.**

- **The AVR was one of the first microcontroller families to use on-chip flash memory for program storage, as opposed to one-time programmable ROM, EPROM, or EEPROM used by other microcontrollers.**

- **The AVR architecture was conceived by two students at the Norwegian Institute of Technology (NTH), Alf-Egil Bogen and Vegard Wollan.**

- **When the technology was sold to Atmel from Nordic VLSI, the internal architecture was further developed by Norway, a subsidiary of Atmel.**

- **It is commonly accepted that AVR stands for Alf and Vegard RISC processor.**

- **The AVR 8-bit microcontroller architecture was introduced in 1997. By 2003, Atmel had shipped 500 million AVR flash microcontrollers.**

BASIC FAMILIES

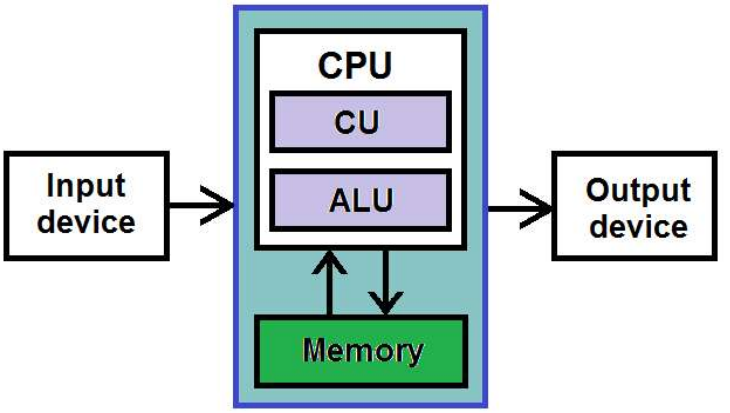
AVRs are generally classified into the following:

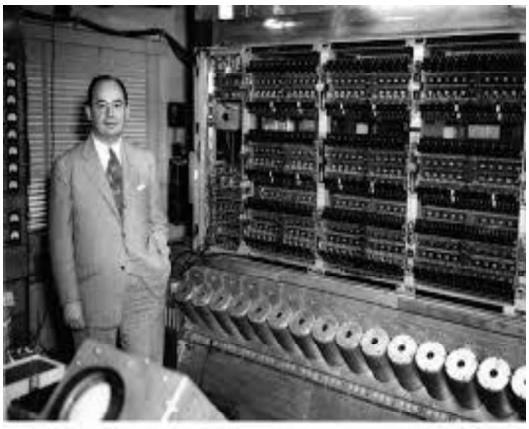
Family name	Series name	program memory	Pins in package	peripheral set	Extra features
tinyAVR	ATtiny	0.5–16 kB	6–32	Limited	
megaAVR	ATmega	4–512 KB	28–100	Extensive	Extended instruction set (multiply instructions and instructions for handling larger program memories)
XMEGA	ATxmega	16–384 KB	44–64–100	Extensive with ADCs	Extended performance features, such as DMA, "Event System", and cryptography support.
Application-specific AVR	megaAVRs with special features not found on the other members of the AVR family, such as LCD controller, USB controller, advanced PWM, CAN, etc.				
FPSLIC (AVR with FPGA)	• FPGA 5K to 40K gates • SRAM for the AVR program code, unlike all other AVRs • AVR core can run at up to 50 MHz				
32-bit AVRs	• In 2006 Atmel released microcontrollers based on the 32-bit AVR32 architecture. • They include SIMD and DSP instructions, along with other audio- and video-processing features. • This 32-bit family of devices is intended to compete with the ARM-based processors. • The instruction set is similar to other RISC cores, but it is not compatible with the original AVR or any of the various ARM cores.				

DEVICE GENERAL ARCHITECTURE

- **The AVR is a modified Harvard architecture machine, where program and data are stored in separate physical memory systems that appear in different address spaces, but having the ability to read data items from program memory using special instructions.**

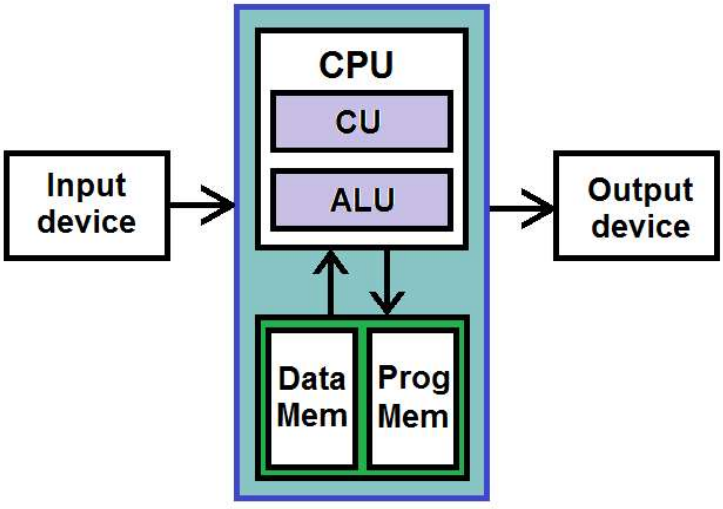
Von Neumann Model for Stored Program Computers





John Von Neumann (1903-1957)

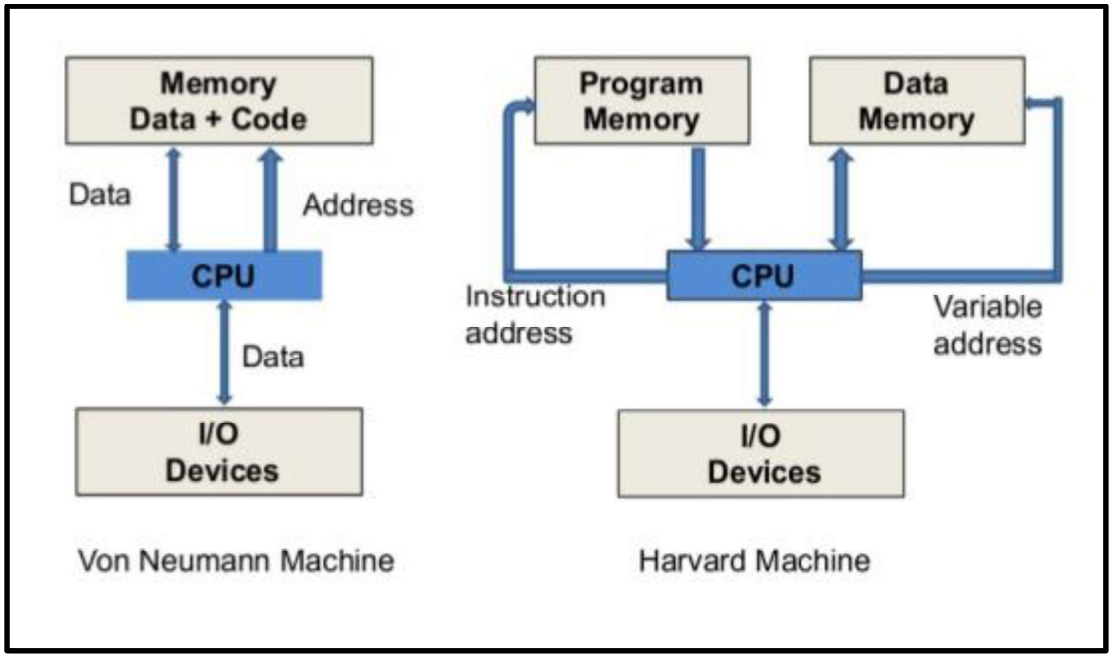
Harvard Architecture



Harvard Architecture Advantages

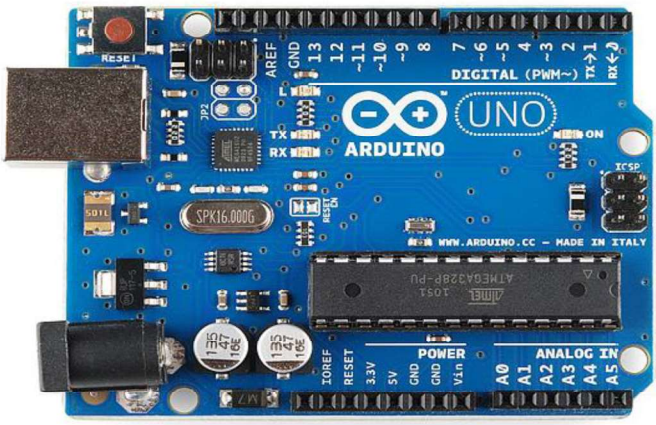
- **Separate instruction and data paths**
- **Simultaneous accesses to instructions & data**
- **Hardware can be optimized for access type and bus width.**

Von Neumann vs. Harvard Architecture



DEVELOPMENT BOARDS

- **Inexpensive development tools.**
- **Free development software.**

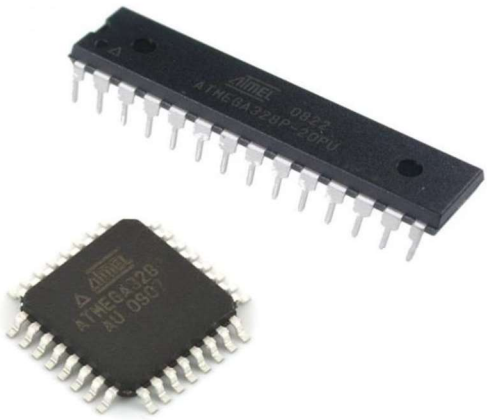


- **Many AVR based cards with different peripheral and memory combinations.**
- **Can be used with conventional programming environments (C, assembler, etc.).**
- **USB-based AVRs have been used in the Microsoft Xbox hand controllers.**

ATMEGA328

ATMEL 8-BIT MICROCONTROLLER WITH 32KBYTES IN-SYSTEM PROGRAMMABLE FLASH

Features



- High Performance, CMOS Low Power 8-Bit Microcontroller
- Advanced RISC Architecture
- 131 Powerful Instructions – Most Single Clock Cycle Execution
- 32 x 8 General Purpose Working Registers
- Fully Static Operation
- Up to 20 MIPS Throughput at 20MHz
- On-chip 2-cycle Multiplier
- High Endurance Non-volatile Memory.
- 4/8/16/32KBytes of In-System Self-Programmable Flash program memory
- 256/512/512/1KBytes EEPROM
- 512/1K/1K/2KBytes Internal SRAM
- Data retention: 20 years at 85^{o}C/100 years at 25^{o}C
- Optional Boot Code Section with Independent Lock Bits
- In-System Programming by On-chip Boot Program
- True Read-While-Write Operation
- Programming Lock for Software Security

Peripheral Features

- Two 8-bit Timer/Counters with Separate Prescaler and Compare Mode
- One 16-bit Timer/Counter with Separate Prescaler, Compare Mode, and
- Capture Mode
- Real Time Counter with Separate Oscillator
- Six PWM Channels
- 6-channel 10-bit ADC
- Programmable Serial USART
- Master/Slave SPI Serial Interface
- Byte-oriented 2-wire Serial Interface (Philips I2C compatible)
- Programmable Watchdog Timer with Separate On-chip Oscillator
- On-chip Analog Comparator
- Interrupt and Wake-up on Pin Change

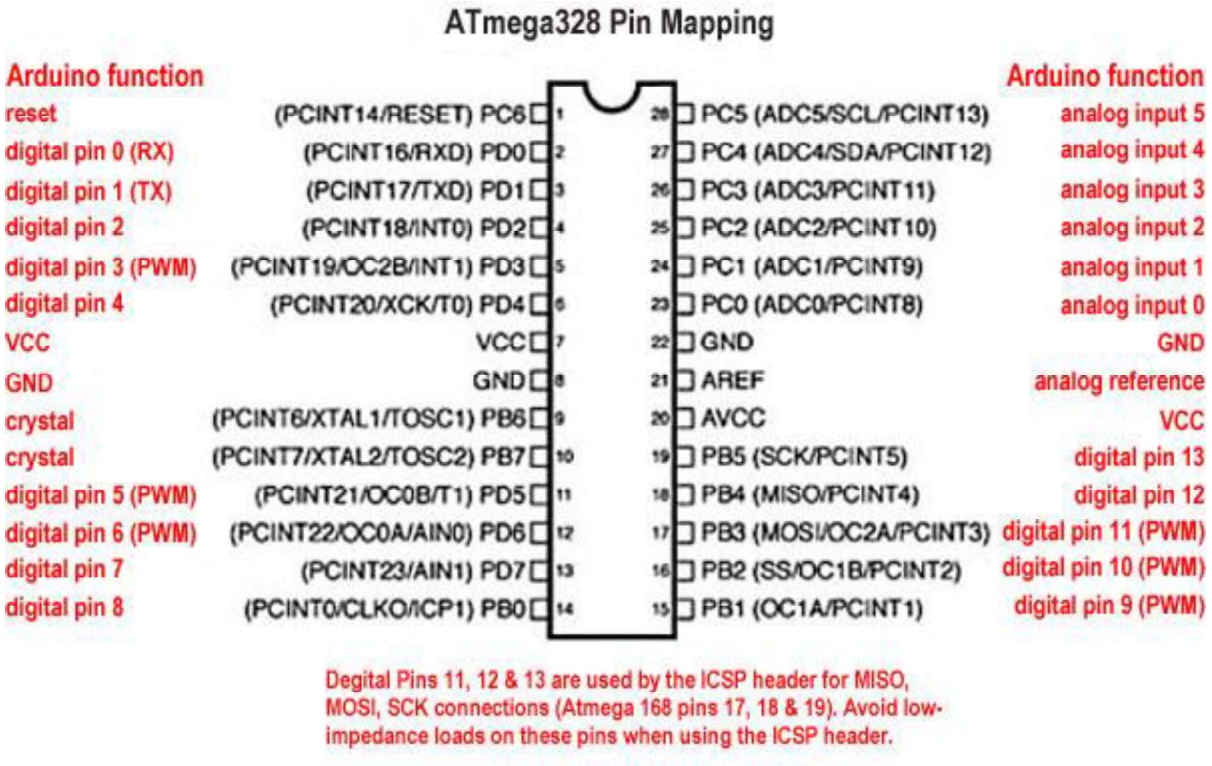
Special Microcontroller Features

- Power-on Reset
- Internal Calibrated Oscillator
- External and Internal Interrupt Sources
- 23 Programmable I/O Lines
- Operating Voltage:–1.8 - 5.5V
- Temperature Range:–-40°C to 85°C
- Speed Grade:–0 - 4MHz@1.8 - 5.5V, 0 - 10MHz@2.7 - 5.5.V, 0 - 20MHz @ 4.5 - 5.5V
- Power Consumption at 1MHz, 1.8V, 25°C

– Active Mode: 0.2mA
– Power-down Mode: 0.1µA
– Power-save Mode: 0.75µA

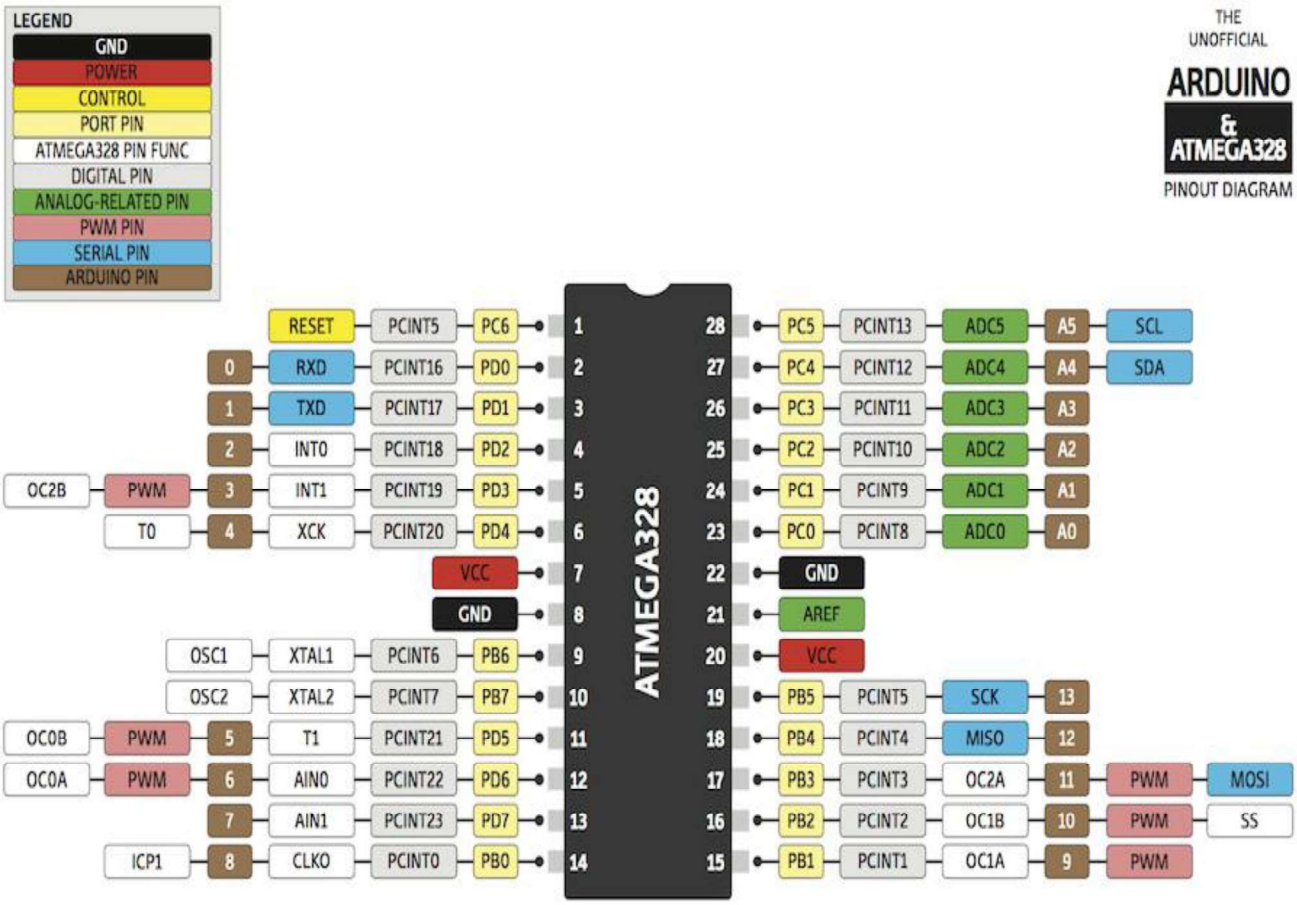
PIN CONFIGURATIONS

The Pinout of ATmega48A/PA/88A/PA/168A/PA/328/P is shown below:



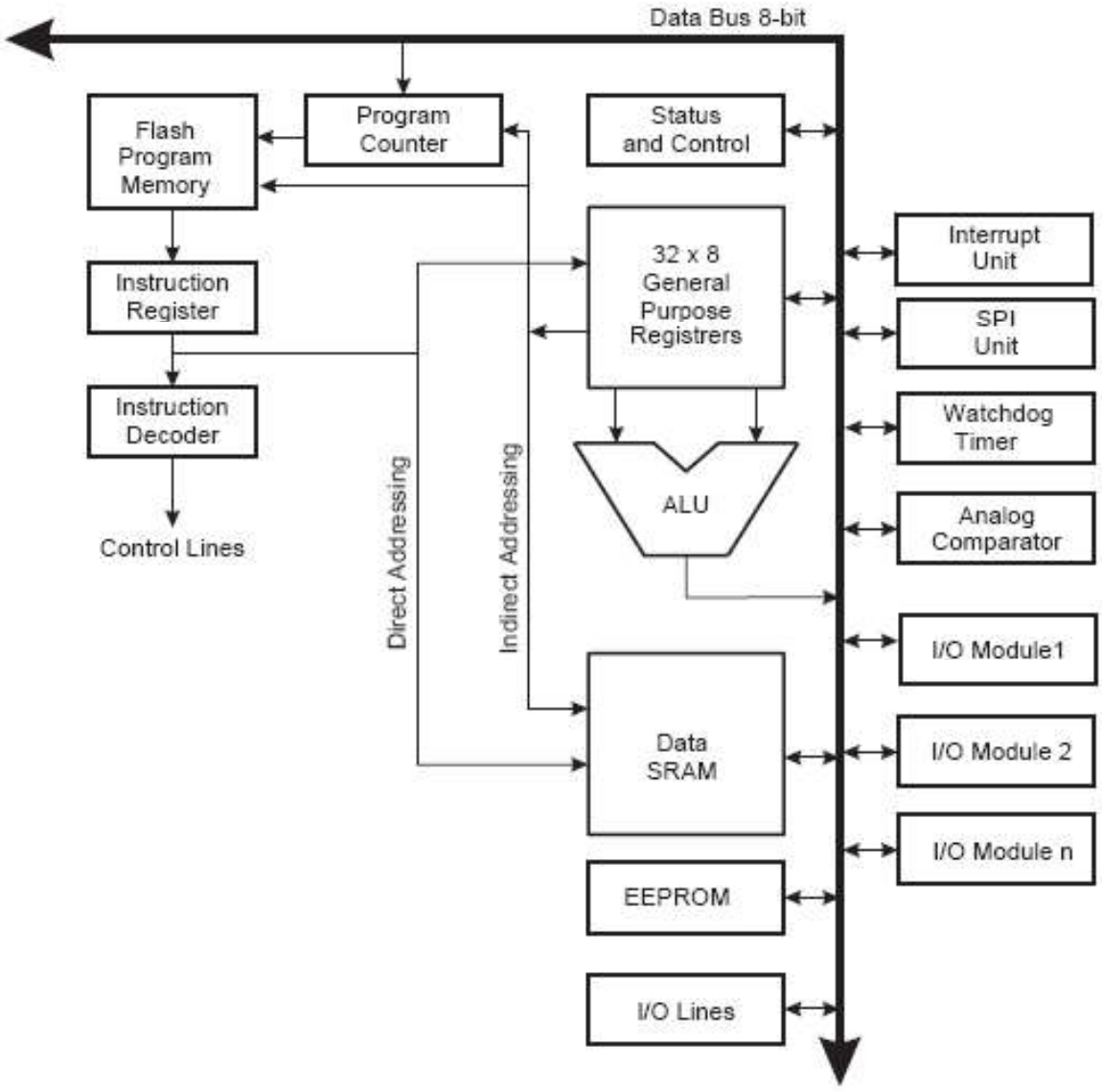
PIN DESCRIPTIONS

1. **VCC**: Digital supply voltage
2. **GND**: Ground
3. **Port B** (PB7:0)
4. **Port C** (PC5:0)
5. **PC6:** RESET
6. **Port D** (PD7:0)
7. **AVCC:** is the supply voltage pin for the A/D Converter.
8. **AREF:** is the analog reference pin for the A/D Converter



AVR CPU Core

This section discusses the AVR core architecture in general. The main function of the CPU core is to ensure correct program execution. The CPU must therefore be able to access memories, perform calculations, control peripherals, and handle interrupts.



ALU – Arithmetic & Logic Unit

- The high-performance AVR ALU operates in direct connection with all the 32 general purpose working registers.
- Within a single clock cycle, arithmetic operations between general purpose registers or between a register and an immediate are executed.
- The ALU operations are divided into three main categories – arithmetic, logical, and bit-functions.
- Some implementations of the architecture also provide a powerful multiplier supporting both signed/unsigned multiplication and fractional format. See the “Instruction Set” section for a detailed description.

Status Register

- The Status Register contains information about the result of the most recently executed arithmetic instruction.
- This information can be used for altering program flow in order to perform conditional operations.
- Note that the Status Register is updated after all ALU operations, as specified in the Instruction Set Reference.

SREG – AVR Status Register

The AVR Status Register – SREG – is defined as:

Bit	7	6	5	4	3	2	1	0
0x3F (0x5F)	I	T	H	S	V	N	Z	C
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Initial Value	0	0	0	0	0	0	0	0

Bit 7 – I: Global Interrupt Enable

- The Global Interrupt Enable bit must be set (=1) for the interrupts to be enabled. The individual interrupt enable control is then performed in separate control registers.
- If the Global Interrupt Enable Register is cleared (=0), none of the interrupts are enabled independent of the individual interrupt enable settings. The I-bit is cleared by hardware after an interrupt has occurred, and is set by the RETI instruction to enable subsequent interrupts.
- The I-bit can also be set and cleared by the application with the SEI and CLI instructions, as described in the instruction set reference.

Bit 6 – T: Bit Copy Storage

The Bit Copy instructions BLD (Bit LoaD) and BST (Bit STore) use the T-bit as source or destination for the operated bit. A bit from a register in the Register File can be copied into T by the BST instruction, and a bit in T can be copied into a bit in a register in the Register File by the BLD instruction.

Bit 5 – H: Half Carry Flag

The Half Carry Flag H indicates a Half Carry in some arithmetic operations. Half Carry Is useful in BCD arithmetic. See the "Instruction Set Description" for detailed information.

Bit 4 – S: Sign Bit, S = N・ ・ V

The S-bit is always an XOR between the Negative Flag N and the Two's Complement Overflow Flag V. See the "Instruction Set Description" for detailed information.

Bit 3 – V: Two's Complement Overflow Flag

The Two's Complement Overflow Flag V supports two's complement arithmetic. See the "Instruction Set Description" for detailed information.

Bit 2 – N: Negative Flag

The Negative Flag N indicates a negative result in an arithmetic or logic operation. See the "Instruction Set Description" for detailed information.

Bit 1 – Z: Zero Flag

The Zero Flag Z indicates a zero result in an arithmetic or logic operation. See the "Instruction Set Description" for detailed information.

Bit 0 – C: Carry Flag

The Carry Flag C indicates a carry in an arithmetic or logic operation. See the "Instruction Set Description" for detailed information.

General Purpose Register File

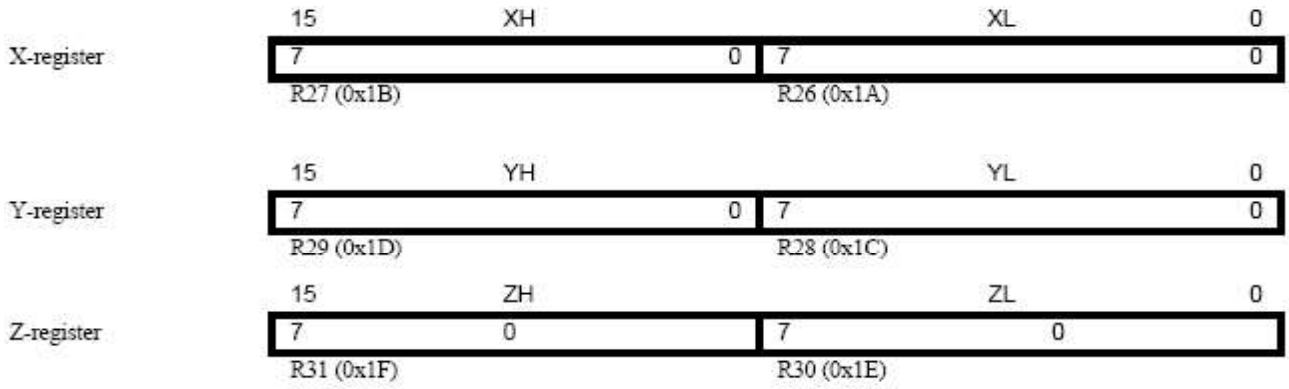
- The Register File is optimized for the AVR Enhanced RISC instruction set.
- In order to achieve the required performance and flexibility, the following input/output schemes are supported by the Register File:

- One 8-bit output operand and one 8-bit result input
- Two 8-bit output operands and one 8-bit result input
- Two 8-bit output operands and one 16-bit result input
- One 16-bit output operand and one 16-bit result input

7 0	Addr.	
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

The X-register, Y-register, and Z-register

- The registers R26...R31 have some added functions to their general purpose usage. These registers are 16-bit address pointers for indirect addressing of the data space.
- The three indirect address registers X, Y, and Z are defined as shown below.
- In the different addressing modes these address registers have functions as fixed displacement, automatic increment, and automatic decrement.



Stack Pointer

- **The Stack is mainly used for storing temporary data**, for storing local variables and for storing return addresses after interrupts and subroutine calls.
- Note that the Stack is implemented as growing from higher to lower memory locations. The Stack Pointer Register always points to the top of the Stack.
- A Stack PUSH command will decrease the Stack Pointer.
- Initial Stack Pointer value equals the last address of the internal SRAM.

- The AVR Stack Pointer is implemented as two 8-bit registers in the I/O space.

- Note that the data space in some implementations of the AVR architecture is so small that only SPL is needed. In this case, the SPH Register will not be present.

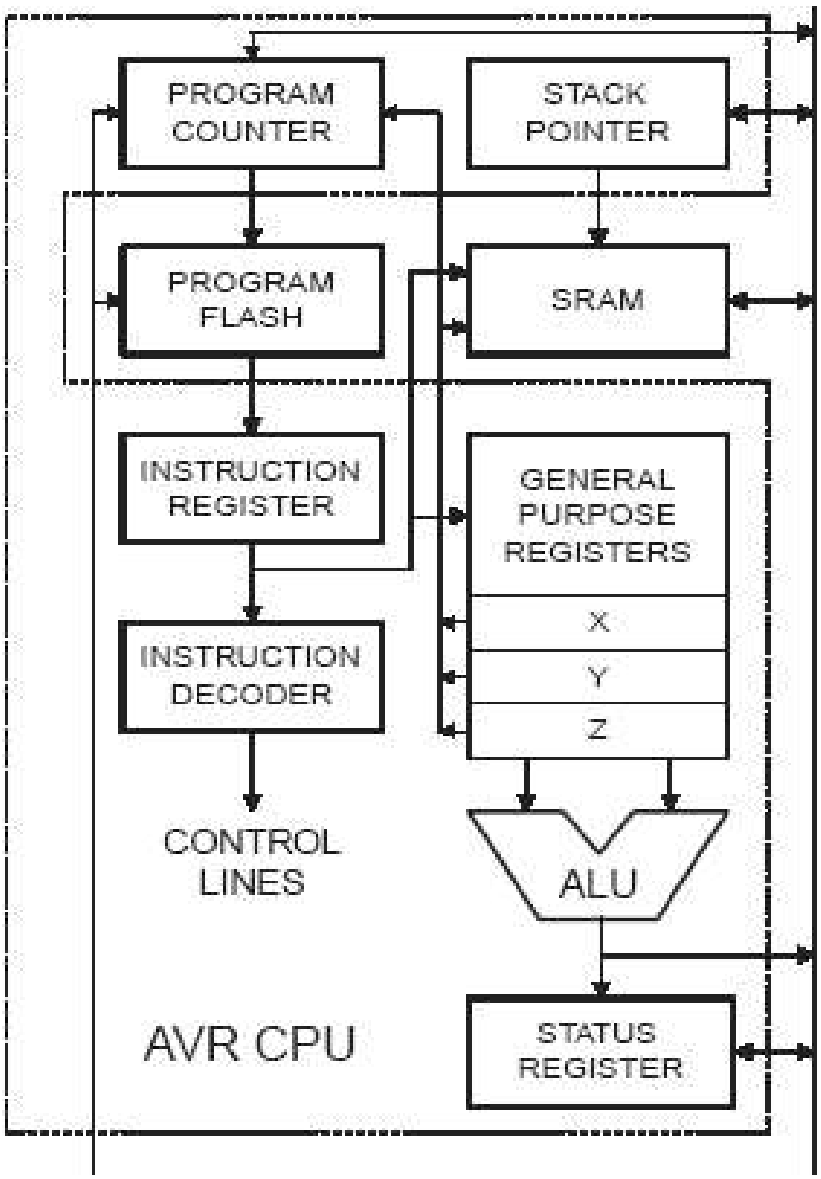
SPH and SPL – Stack Pointer High and Stack Pointer Low Register

Bit	15	14	13	12	11	10	9	8	
0x3E (0x5E)	SP15	SP14	SP13	SP12	SP11	SP10	SP9	SP8	SPH
0x3D (0x5D)	SP7	SP6	SP5	SP4	SP3	SP2	SP1	SP0	SPL
	7	6	5	4	3	2	1	0	
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	RAMEND	RAMEND	RAMEND	RAMEND	RAMEND	RAMEND	RAMEND	RAMEND	
	RAMEND	RAMEND	RAMEND	RAMEND	RAMEND	RAMEND	RAMEND	RAMEND	

Instruction Execution Timing

- The AVR CPU is driven by the CPU clock clk_{CPU}, directly generated from the selected clock source for the chip. No internal clock division is used.
- When the device is powered-on, the Program Counter is set to 0.
- The instruction at the location in Flash Memory at the address indicated by the Program Counter is fetched and placed in the Instruction Register
- The opcode and operands within the instruction are extracted by the Instruction Decoder

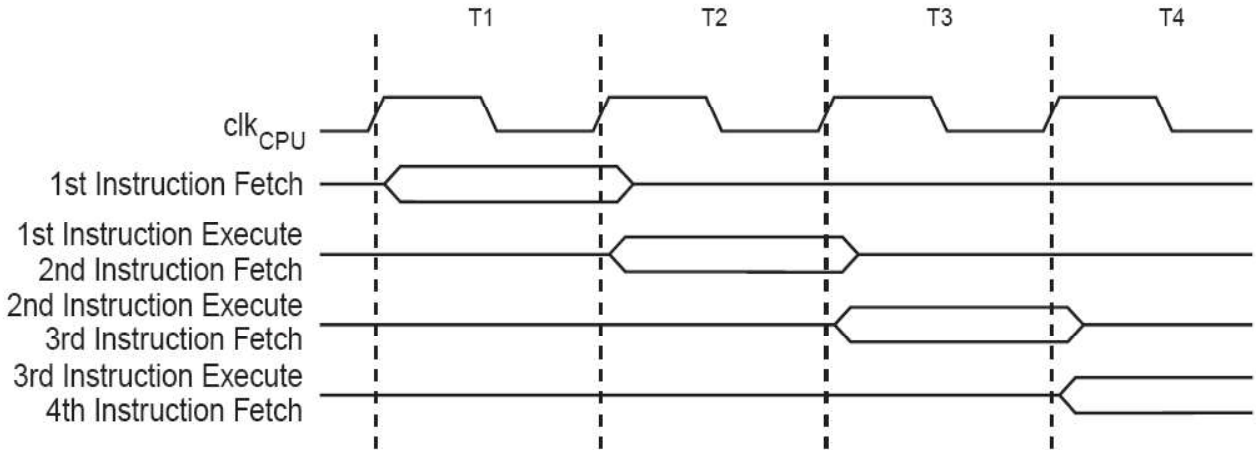
- The control lines from the I.D. activate the particular circuitry within the ALU that is capable of processing that particular opcode. The ALU executes the instruction.
- The Program Counter is automatically incremented and the cycle repeats.



Example: ADD R5, R12

Pipelining

- The parallel instruction fetches and instruction executions are enabled by the Harvard architecture and the fast-access Register File concept. This is the basic pipelining concept to obtain up to 1 MIPS per MHz with the corresponding unique results for functions per cost, functions per clocks, and functions per power-unit.



Binary Representation of Instructions

Let's consider the case of how the ADD instruction is represented as a machine-executable instruction

<Homework: Mov R4, R6. Assume Mov opcode=11011>

Let's consider the case of how the ADD instruction is represented as a machine-executable instruction

Example: ADD R20, R5

- A 16-bit machine instruction is generated (by the assembler) for this particular case
  - Machine instructions consist of an numeric opcode and operands
  - The **opcode** in this case is 000011 (3)
  - Each register operand is represented by 5 bits
    - 5 bits are required to represent all possible register values from 0-31
    - ddddd represent the 5 bits that represent the destination register value (20)
    - rrrrr represent the 5 bits that represent the source register value (5)

0000	11rd	dddd	rrrr

add r20, r5 is *assembled* to:

0000	1101	0100	0101

Note: The bits ddddd and rrrrr are "split" in this instruction

This 16-bit binary word can be expressed in hexadecimal as 0x450d (with the "high" byte being rightmost)

Dr. M

The ALU can only directly operate on data that has been fetched into the Registers. It cannot directly operate on SRAM or EEPROM data.

In the assembly language instruction

```
add r20, r5
```

- We (the programmer) must first load some values into these registers. Once way of doing this is with the following instructions:

```
ldi r20, 2   ; load value 2 into r20
lds r5, 0x60 ; load value at SRAM addr 0x60 into r5
add r20, r5 ; add them; result is in r20
```

Following the add, we normally store the sum (the value in r20) someplace (like in SRAM). More on how to do that later…how would you guess it might work?

AVR Memories

- The AVR architecture has two main memory spaces, the Data Memory and the Program Memory space.

- In addition, the ATmega 328 features an EEPROM Memory for data storage.

- All **three** memory spaces are linear and regular.

(1) In-System Reprogrammable Flash Program Memory

- The ATmega 328 contains 32Kbytes On-chip In-System Reprogrammable Flash memory for program storage.

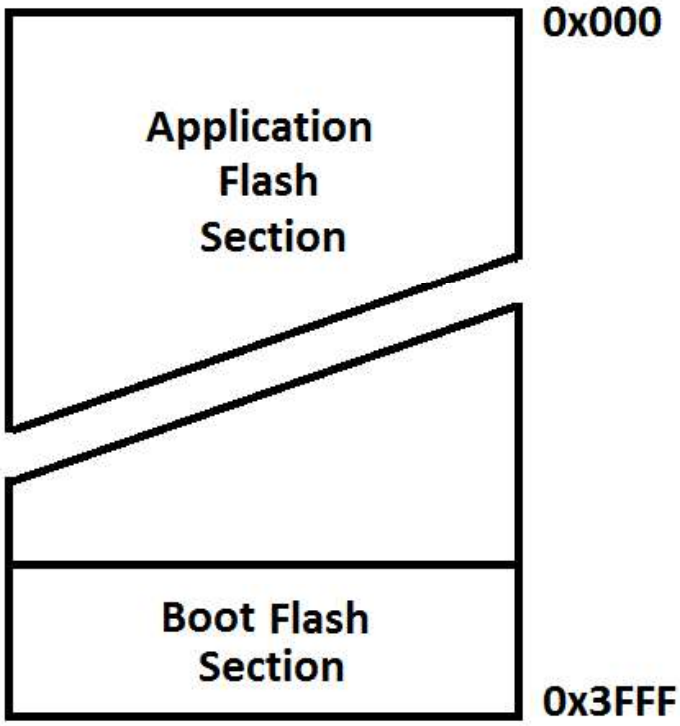
- Since all AVR instructions are 16 or 32 bits wide, the Flash is organized as 16K x 16.

- For software security, the Flash Program memory space is divided into two sections, Boot Loader Section and Application Program Section.

- The Flash memory has an endurance of at least 10,000 write/erase cycles.

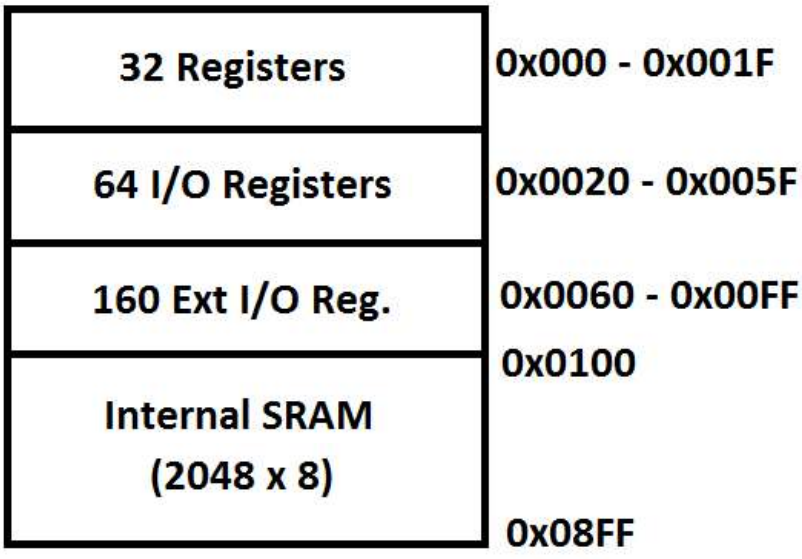
- The ATmega 328 Program Counter (PC) is 14 bits wide, thus addressing the 16K program memory locations.

The following figure shows the Program Memory Map of ATmega 328



(2) SRAM Data Memory

- The 32 general purpose working registers, 64 I/O Registers, 160 Extended I/O Registers, and the 2048 bytes of internal data SRAM in the ATmega 328 are all accessible through all these addressing modes. The Data Memory Map is shown below



(3) EEPROM Data Memory

- The ATmega 328 contains 1 Kbyte of data EEPROM memory.
- It is organized as a separate data space, in which single bytes can be read and written.
- The EEPROM has an endurance of at least 100,000 write/erase cycles.

AVR ARCHITECTURE AND ASSEMBLY LANGUAGE PROGRAMMING

LDI instruction

Simply stated, the LDI instruction copies 8-bit data into the general purpose registers. It has the following format:

```
LDI Rd,K     ;load Rd (destination) with Immediate value K
             ;d must be between 16 and 31
```

K is an 8-bit value that can be 0–255 in decimal, or 00–FF in hex, and Rd is R16 to R31 (any of the upper 16 general purpose registers). The I in LDI stands for "immediate." If we see the word "immediate" in any instruction, we are dealing with a value that must be provided right there with the instruction. The following instruction loads the R20 register with a value of 0x25 (25 in hex).

```
LDI R20,0x25              ;load R20 with 0x25 (R20 = 0x25)
```

The following instruction loads the R31 register with the value 0x87 (87 in hex).

```
LDI R31,0x87              ;load 0x87 into R31  (R31 = 0x87)
```

The following instruction loads R25 with the value 0x15 (15 in hex and 21 in decimal).

```
LDI R25,0x79              ;load 0x79 into R25 (R25 = 0x79)
```

Note: We cannot load values into registers R0 to R15 using the LDI instruction. For example, the following instruction is not valid:

```
LDI R5,0x99               ;invalid instruction
```

ADD instruction

The ADD instruction has the following format:

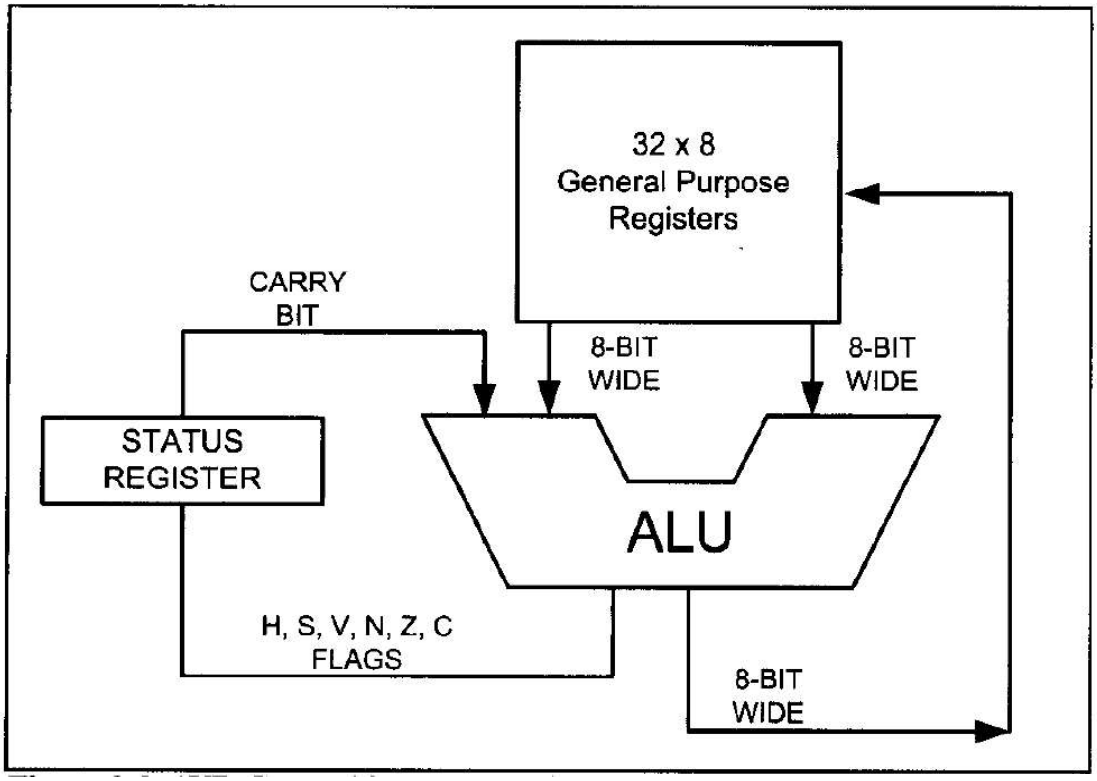
```
ADD  Rd,Rr  ;ADD Rr to Rd and store the result in Rd
```

The ADD instruction tells the CPU to add the value of Rr to Rd and put the result back into the Rd register. To add two numbers such as 0x25 and 0x34, one can do the following:

```
LDI R16,0x25      ;load 0x25 into R16
LDI R17,0x34      ;load 0x34 into R17
ADD R16,R17       ;add value R17 to R16 (R16 = R16 + R17)
```

Executing the above lines results in R16 = 0x59 (0x25 + 0x34 = 0x59)

The affect of arithmetic and logic operations on the status register



Review Questions

1. Write instructions to move the value 0x34 into the R29 register.
2. Write instructions to add the values 0x16 and 0xCD. Place the result in the R19 register.

LDS instruction (LoaD direct from data Space)

```
LDS    Rd, K   ;load Rd with the contents of location K (0 ≤ d ≤ 31)
               ;K is an address between $0000 to $FFFF
```

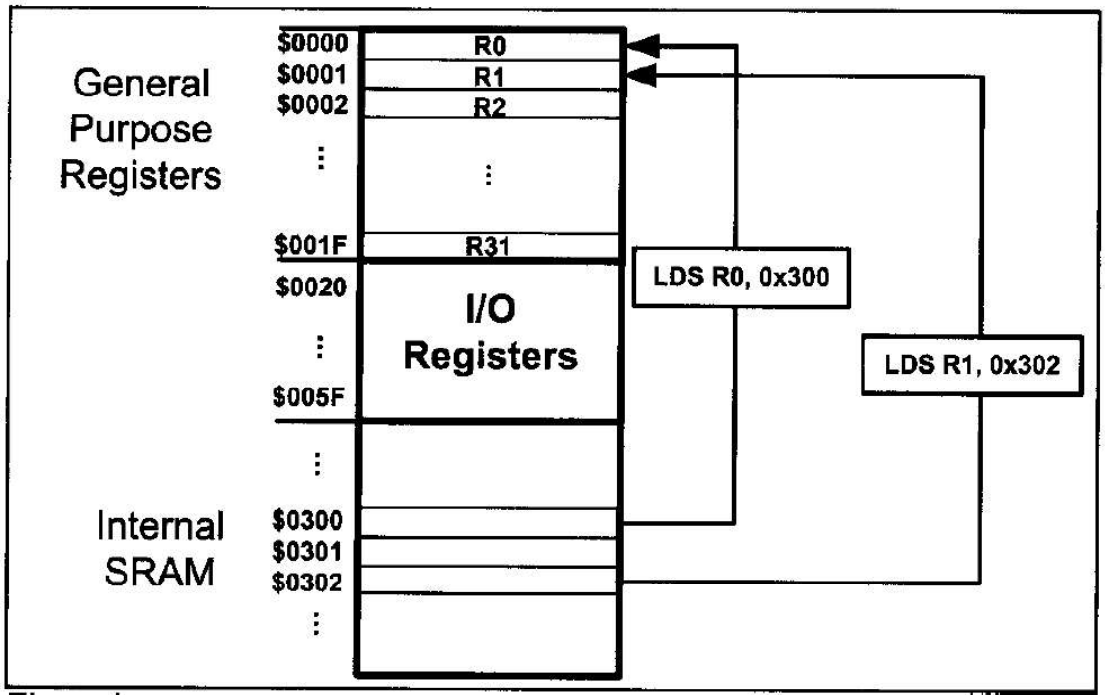
The LDS instruction tells the CPU to load (copy) one byte from an address in the data memory to the GPR. After this instruction is executed, the GPR will have the same value as the location in the data memory. The location in the data memory can be in any part of the data space; it can be one of the I/O registers, a location in the internal SRAM, or a GPR. For example, the "`LDS R20,0x1`" instruction will copy the contents of location 1 (in hex) into R20. As you can see in Figure 2-3, location 1 of the data memory is in the GPR part, and it is the address of R1. So, the instruction copies R1 to R20.

The following instruction loads R5 with the contents of location 0x200. As you can see in Figure 2-3, 0x200 is located in the internal SRAM:

```
LDS R5,0x200 ;load R5 with the contents of location $200
```

The following program adds the contents of location 0x300 to location 0x302. To do so, first it loads R0 with the contents of location 0x300 and R1 with the contents of location 0x302, then adds R0 to R1:

```
LDS   R0, 0x300   ;R0 = the contents of location 0x300
LDS   R1, 0x302   ;R1 = the contents of location 0x302
ADD   R1, R0      ;add R0 to R1
```



STS instruction (STore direct to data Space)

```
STS   K, Rr ;store register into location K
            ;K is an address between $0000 to $FFFF
```

The STS instruction tells the CPU to store (copy) the contents of the GPR to an address location in the data memory space. After this instruction is executed, the location in the data space will have the same value as the GPR. The location can be in any part of the data memory space; it can be one of the I/O registers, a location in the SRAM, or a GPR. For example, the "`STS 0x1,R10`" instruction will copy the contents of R10 into location 1. As you can see in Figure 2-3, location 1 of the data memory is in the GPR part, and it is the address of R1. So, the instruction copies R10 to R1.

The following instruction stores the contents of R25 to location 0x230. As you can see in Figure 2-3, 0x230 is located in the internal SRAM:

```
STS 0x230, R25    ;store R25 to data space location 0x230
```

The following program first loads the R16 register with value 0x55, then moves this value around to I/O registers of ports B, C, and D. As shown in Figure 2-7, the addresses of PORTB, PORTC, and PORTD are 0x38, 0x35, and 0x32, respectively:

```
LDI   R16, 0x55   ;R16 = 55 (in hex)
STS   0x38, R16   ;copy R16 to Port B (PORTB = 0x55)
STS   0x35, R16   ;copy R16 to Port C (PORTC = 0x55)
STS   0x32, R16   ;copy R16 to Port D (PORTD = 0x55)
```

Notice that you cannot copy (store) an immediate value directly into the SRAM location in the AVR. This must be done via the GPRs.

The following program adds the contents of location 0x220 to location 0x221, and stores the result in location 0x221:

```
LDS   R30, 0x220  ;load R30 with the contents of location 0x220
LDS   R31, 0x221  ;load R31 with the contents of location 0x221
ADD   R31, R30    ;add R30 to R31
STS   0x221, R31  ;store R31 to data space location 0x221
```

Example 2-1

State the contents of RAM locations $212 to $216 after the following program is executed:

```
LDI   R16, 0x99   ;load R16 with value 0x99
STS   0x212, R16
LDI   R16, 0x85   ;load R16 with value 0x85
STS   0x213, R16
LDI   R16, 0x3F   ;load R16 with value 0x3F
STS   0x214, R16
LDI   R16, 0x63   ;load R16 with value 0x63
STS   0x215, R16
LDI   R16, 0x12   ;load R16 with value 0x12
STS   0x216, R16
```

Solution:

After the execution of `STS 0x212, R16` data memory location $212 has value 0x99; after the execution of `STS 0x213, R16` data memory location $213 has value 0x85; after the execution of `STS 0x214, R16` data memory location $214 has value 0x3F; after the execution of `STS 0x215, R16` data memory location $215 has value 0x63; and so on, as shown in the chart.

Address	Data
$212	0x99
$213	0x85
$214	0x3F
$215	0x63
$216	0x12

Example 2-2

State the contents of R20, R21, and data memory location 0x120 after the following program:

```
LDI   R20, 5      ;load R20 with 5
LDI   R21, 2      ;load R21 with 2
ADD   R20, R21    ;add R21 to R20
ADD   R20, R21    ;add R21 to R20
STS   0x120, R20  ;store in location 0x120 the contents of R20
```

Solution:

The program loads R20 with value 5. Then it loads R21 with value 2. Then it adds the R21 register to R20 twice. At the end, it stores the result in location 0x120 of data memory.

Location	Data
R20	5
R21	
0x120	

After LDI R20, 5

Location	Data
R20	5
R21	2
0x120	

After LDI R21, 2

Location	Data
R20	7
R21	2
0x120	

After ADD R20, R21

Location	Data
R20	9
R21	2
0x120	

After ADD R20, R21

Location	Data
R20	9
R21	2
0x120	9

After STS 0x120, R20

MOV instruction

The MOV instruction is used to copy data among the GPR registers of R0–R31. It has the following format:

```
MOV   Rd,Rr          ;Rd = Rr (copy Rr to Rd)
                     ;Rd and Rr can be any of the GPRs
```

For example, the following instruction copies the contents of R20 to R10:

```
MOV   R10,R20        ;R10 = R20
```

For instance, if R20 contains 60, after execution of the above instruction both R20 and R10 will contain 60.

Example

The following program adds 0x19 to the contents of location 0x220 and stores the result in location 0x221:

```
LDI    R20, 0x19    ;load R20 with 0x19
LDS    R21, 0x220   ;load R21 with the contents of location 0x220
ADD    R21, R20     ;R21 = R21 + R20
STS    0x221, R21   ;store R21 to location 0x221
```

INC instruction

```
INC    Rd     ;increment the contents of Rd by one (0 ≤ d ≤ 31)
```

The INC instruction increments the contents of Rd by 1. For example, the following instruction adds 1 to the contents of R2:

```
INC    R2           ;R2 = R2 + 1
```

The following program increments the contents of data memory location 0x430 by 1:

```
LDS    R20, 0x430   ;R20 = contents of location 0x430
INC    R20          ;R20 = R20 + 1
STS    0x430, R20   ;store R20 to location 0x430
```

SUB instruction

The SUB instruction has the following format:

```
SUB   Rd,Rr          ;Rd = Rd - Rr
```

The SUB instruction tells the CPU to subtract the value of Rr from Rd and put the result back into the Rd register. To subtract 0x25 from 0x34, one can do the following:

```
LDI   R20, 0x34    ;R20 = 0x34
LDI   R21, 0x25    ;R20 = 0x25
SUB   R20, R21     ;R20 = R20 - R21
```

The following program subtracts 5 from the contents of location 0x300 and stores the result in location 0x320:

```
LDS   R0, 0x300    ;R0 = contents of location 0x300
LDI   R16, 0x5     ;R16 = 0x5
SUB   R0, R16      ;R0 = R0 - R16
STS   0x320,R0     ;store the contents of R0 to location 0x320
```

The following program decrements the contents of R10, by 1:

```
LDI   R16, 0x1     ;load 1 to R16
SUB   R10, R16     ;R10 = R10 - R16
```

DEC instruction

The DEC instruction has the following format:

```
DEC   Rd           ;Rd = Rd - 1
```

The DEC instruction decrements (subtracts 1 from) the contents of Rd and puts the result back into the Rd register. For example, the following instruction subtracts 1 from the contents of R10:

```
DEC   R10          ;R10 = R10 - 1
```

In the following program, we put the value 3 into R30. Then the value in R30 is decremented.

```
LDI   R30, 3       ;R30 = 3
DEC   R30          ;R30 has 2
DEC   R30          ;R30 has 1
DEC   R30          ;R30 has 0
```

More instructions

Table 2-2: ALU Instructions Using Two GPRs

Instruction		
ADD	Rd, Rr	ADD Rd and Rr
ADC	Rd, Rr	ADD Rd and Rr with Carry
AND	Rd, Rr	AND Rd with Rr
EOR	Rd, Rr	Exclusive OR Rd with Rr
OR	Rd, Rr	OR Rd with Rr
SBC	Rd, Rr	Subtract Rr from Rd with carry
SUB	Rd, Rr	Subtract Rr from Rd without carry

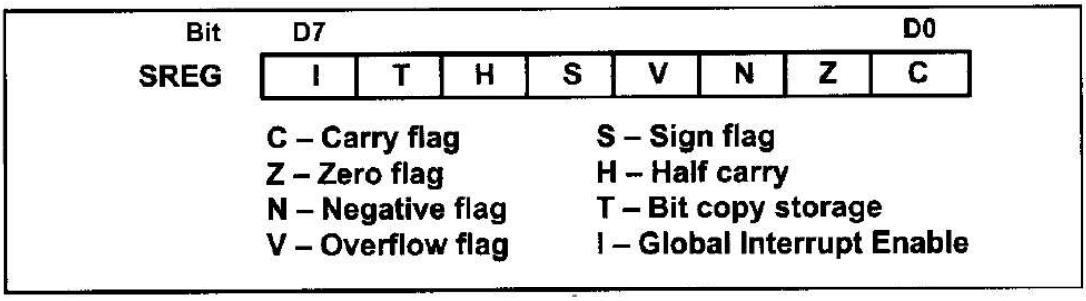
Rd and Rr can be any of the GPRs. See Chapter 5 for examples of the instructions in Table 2-2.

Table 2-3: Some Instructions Using a GPR as Operand

Instruction		
CLR	Rd	Clear Register Rd
INC	Rd	Increment Rd
DEC	Rd	Decrement Rd
COM	Rd	One's Complement Rd
NEG	Rd	Negative (two's complement) Rd
ROL	Rd	Rotate left Rd through carry
ROR	Rd	Rotate right Rd through carry
LSL	Rd	Logical Shift Left Rd
LSR	Rd	Logical Shift Right Rd
ASR	Rd	Arithmetic Shift Right Rd
SWAP	Rd	Swap nibbles in Rd

Chapters 3 through 6 will show how to use the instructions in Table 2-3.

Bits of Status Register



Example 2-5

Show the status of the Z flag during the execution of the following program:

```
LDI   R20,4        ;R20 = 4
DEC   R20          ;R20 = R20 - 1
DEC   R20          ;R20 = R20 - 1
DEC   R20          ;R20 = R20 - 1
DEC   R20          ;R20 = R20 - 1
```

Solution:

The Z flag is one when the result is zero. Otherwise, it is cleared (zero). Thus:

After	Value of R20	The Z flag
LDI R20, 4	4	0
DEC R20	3	0
DEC R20	2	0
DEC R20	1	0
DEC R20	0	1

Example 2-6

Show the status of the C, H, and Z flags after the addition of 0x38 and 0x2F in the following instructions:

```
LDI   R16, 0x38
LDI   R17, 0x2F
ADD   R16, R17    ;add R17 to R16
```

Solution:

```
  $38      0011 1000
+ $2F      0010 1111
  $67      0110 0111     R16 = 0x67
```

C = 0 because there is no carry beyond the D7 bit.
H = 1 because there is a carry from the D3 to the D4 bit.
Z = 0 because the R16 (the result) has a value other than 0 after the addition.

Example 2-7

Show the status of the C, H, and Z flags after the addition of 0x9C and 0x64 in the following instructions:

```
LDI   R20, 0x9C
LDI   R21, 0x64
ADD   R20, R21    ;add R21 to R20
```

Solution:

$9C	1001 1100	
+ $64	0110 0100	
$100	0000 0000	R20 = 00

C = 1 because there is a carry beyond the D7 bit.
H = 1 because there is a carry from the D3 to the D4 bit.
Z = 1 because the R20 (the result) has value 0 in it after the addition.

Example 2-8

Show the status of the C, H, and Z flags after the addition of 0x88 and 0x93 in the following instructions:

```
LDI   R20, 0x88
LDI   R21, 0x93
ADD   R20, R21    ;add R21 to R20
```

Solution:

$ 88	1000 1000	
+ $ 93	1001 0011	
$11B	0001 1011	R20 = 0x1B

C = 1 because there is a carry beyond the D7 bit.
H = 0 because there is no carry from the D3 to the D4 bit.
Z = 0 because the R20 has a value other than 0 after the addition.

Not all instructions affect the flags

Instructions That Affect Flag Bits

Instruction	C	Z	N	V	S	H
ADD	X	X	X	X	X	X
ADC	X	X	X	X	X	X
ADIW	X	X	X	X	X	
AND		X	X	X	X	
ANDI		X	X	X	X	
CBR		X	X	X	X	
CLR		X	X	X	X	
COM	X	X	X	X	X	
DEC		X	X	X	X	
EOR		X	X	X	X	
FMUL	X	X				
INC		X	X	X	X	
LSL	X	X	X	X		X
LSR	X	X	X	X		
OR		X	X	X	X	
ORI		X	X	X	X	
ROL	X	X	X	X		X
ROR	X	X	X	X		
SEN			1			
SEZ		1				
SUB	X	X	X	X	X	X
SUBI	X	X	X	X	X	X
TST		X	X	X	X	

More AVR Assembly Instructions

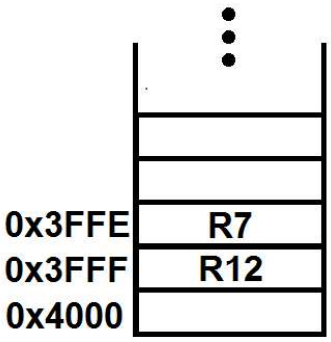
STACK MEMORY

Mnemonic	Operation	Description
PUSH Rr	Stack ← Rr	- Decrement SP by 1 - Copy Rr to the top of stack memory
POP Rd	Rd ← Stack	- Copy the top of stack memory to Rd - Increment SP by 1

Note that the Stack Pointer (SP) must be initialized before any instruction can deal with stack memory.

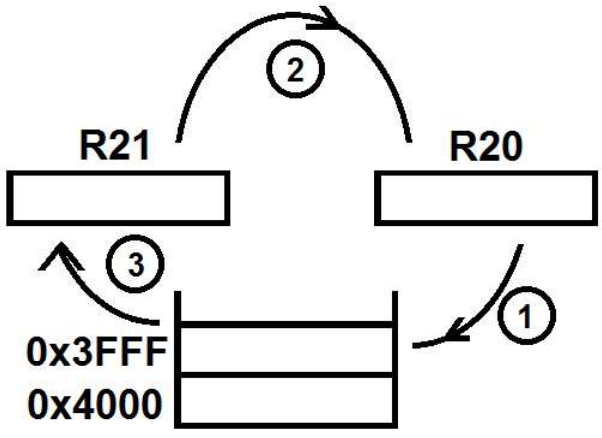
Example: Use the stack memory to store the contents of R12 and R7 temporarily. The address of the last location in SRAM is 0x4000.

```
LDI SPL, 0x00     ;  Load SP lower byte
LDI SPH, 0x40     ;  Load SP higher byte. Now SP=0x4000
PUSH R12          ;  Decrement SP (SP=0x3FFF)
                     Store R12 in stack memory (0x3FFF)
PUSH R7           ;  Decrement SP (SP=0x3FFE)
                     Store R7 in stack memory (0x3FFE)
.
.                 ;  Other instructions
.
POP R7            ;  Load R7 from stack memory (0x3FFE)
                     Increment SP (SP=0x3FFF)
POP R12           ;  Load R12 from stack memory (0x3FFF)
                     Increment SP (SP=0x4000)
END               ;  Stop
```



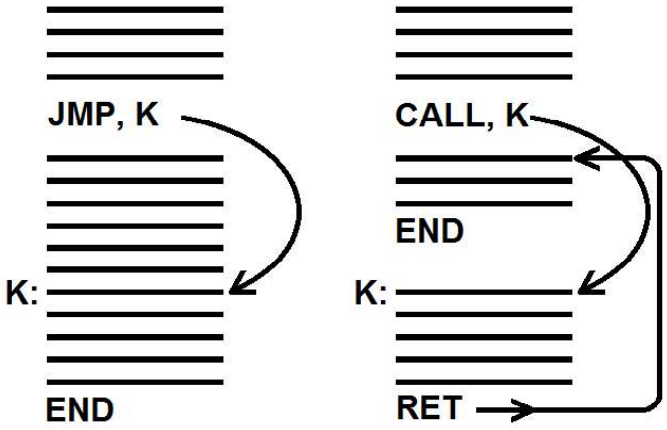
Example: Swap the contents of R20 and R21 via stack memory. The address of the last location in SRAM is 0x4000.

```
LDI SPL, 0x00
LDI SPH, 0x40
PUSH R20
MOV R20, R21
POP R21
END
```



UNCONDITIONAL BRANCHING

Mnemonic	Operation	Description
JMP, K	PC ← K	Jump to address K
CALL, K	stack ← PC PC ← K	- PUSH PC to stack memory - Jump to address K
RET	PC ← stack	Return



JMP	CALL-RET
One-way jump	Two-way jump
---	Store PC in stack
Used to skip instructions	Used to call and execute subprograms

COMPARISON

Mnemonic	Operation	Description
CP Rd, Rr	Rd - Rr	Update C & Z flags according to the result of Rd-Rr
CPI Rd, K	Rd - K	Update C & Z flags according to the result of Rd-K

Note that after CP Rd, Rr is executed, the following can be concluded from the C and Z flags:

C	Z	Conclusion
0	1	Rd = Rr
0	0	Rd > Rr
1	0	Rd < Rr

Example: Specify the contents of R20, R21, C and Z flags after the execution of each instruction in the following program. Suppose that all registers and flags are reset initially.

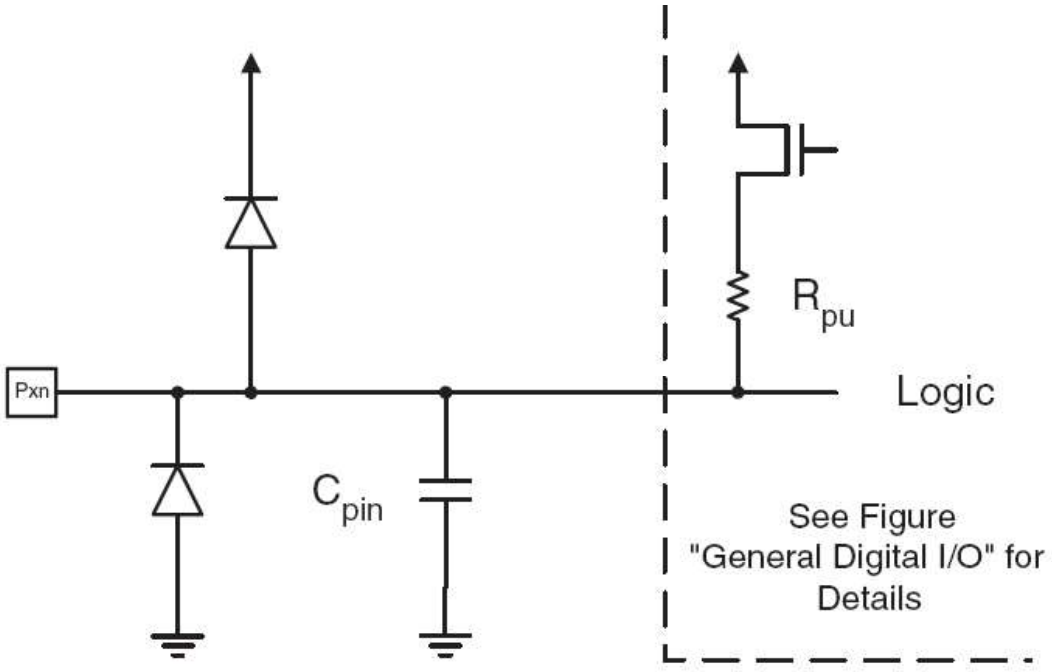
	R20	R21	C	Z
LDI R20, 0x3A	**0x3A**	**0x00**	**0**	**0**
LDI R21, 0x3A	**0x3A**	**0x3A**	**0**	**0**
CP R20, R21	**0x3A**	**0x3A**	**0**	**1**
CPI R20, 0xFF	**0x3A**	**0x3A**	**1**	**0**

CONDITIONAL BRANCHING

I/O Ports

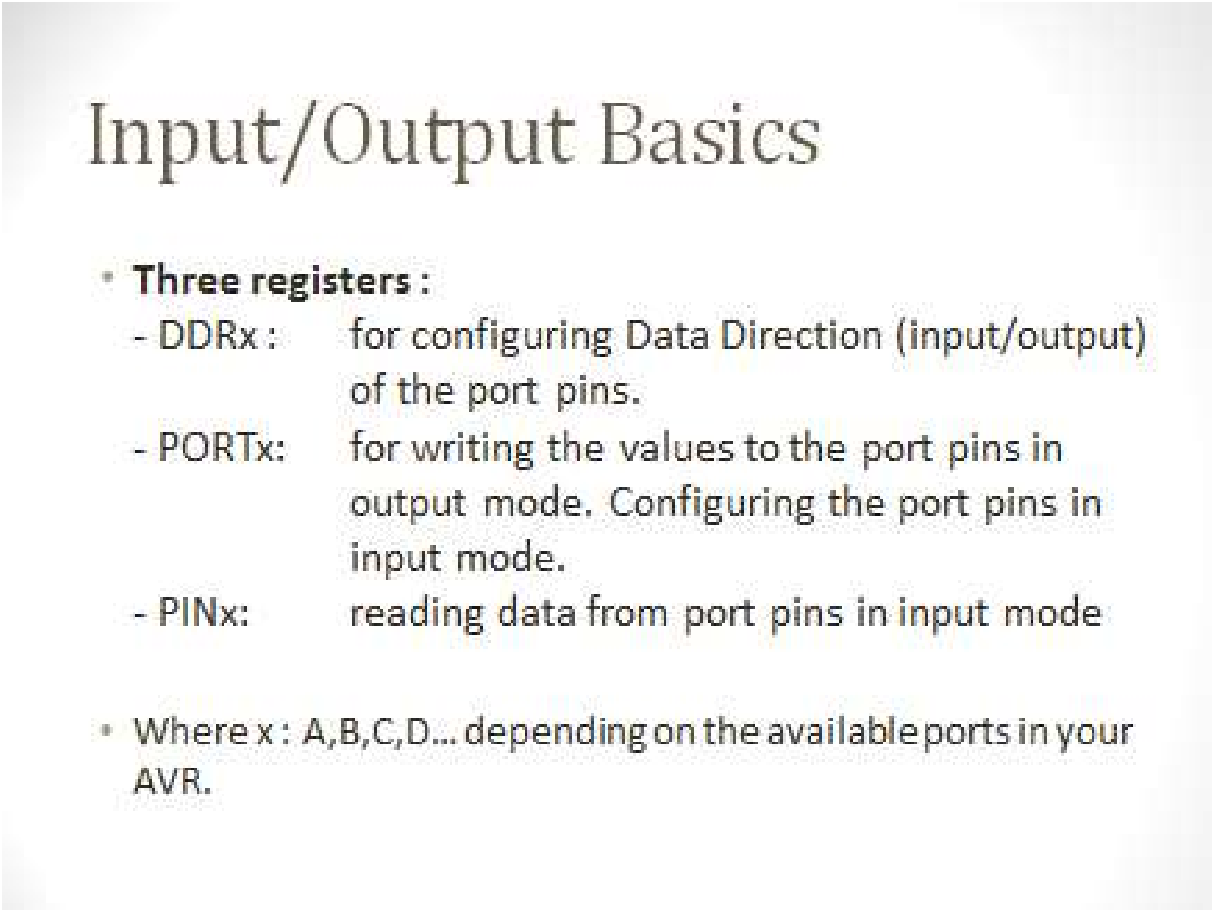
- All AVR ports have true Read-Modify-Write functionality when used as general digital I/O ports.
- This means that the direction of one port pin can be changed without unintentionally changing the direction of any other pin.
- Each output buffer has symmetrical drive characteristics with both high sink and source capability. The pin driver is strong enough to drive LED displays directly.
- All port pins have individually selectable pull-up resistors with a supply-voltage invariant resistance.
- All I/O pins have protection diodes to both VCC and Ground as indicated in the figure below.

I/O Pin Equivalent Schematic



- **In the Atmega328, there are three 8-bit I/O Ports**
  - Port B, Port C & Port D
  - Pins identified as PB.n, PC.n or PD.n (n=0…7)
- **Each pin can be configured as:**
  - Input with internal pull-up
  - Input with no pull-up (tri-state)
  - Output low
  - Output high

There are three registers associated with each I/O port, as folloows.



DDR – Data Direction Register

- Configures data direction of the port - Input / Output
- DDRx.n = 0 > makes corresponding port pin as input
  DDRx.n = 1 > makes corresponding port pin as output

- Examples :
  - to make all pins of port A as input pins :
    ```
    DDRA = 0b00000000;
    ```
  - to make all pins of port A as output pins
    ```
    DDRA = 0b11111111;
    ```
  - to make lower nibble of port B as output and higher nibble as input
    ```
    DDRB = 0b00001111;
    ```

PIN register

- Used to read data from port pins, when port is configured as input.
- First set DDRx to zero, then use PINx to read the value.
- If PINx is read, when port is configured as output, it will give you data that has been outputted on port.
- There two input modes :
  - Tristated input
  - Pullup input

  This will be explained shortly

- Example :

```
DDRA = 0x00;      //Set PA as input
x = PINA;  //Read contents of PA
```

PORT register

- Used for two purposes ...

1) for data output, when port is configured as output:

- Writing to PORTx.n will immediately (in same clock cycle) change state of the port pins according to given value.
- Do not forget to load DDRx with appropriate value for configuring port pins as output.
- Examples :
  - to output 0xFF data on PB

```
DDRB = 0b11111111;      //set all pins of
port b as outputs
PORTB = 0xFF;     //write data on port
```

  - to output data in variable x on PA

```
DDRA = 0xFF;       //make port a as output
PORTA = x;       //output 8 bit variable on
port
```

PORT register

2) for configuring pin as tristate/pullup, when port is configured as input) :

- When port is configures as input (i.e DDRx.n=0), then PORTx.n controls the internal pull-up resistor.
- PORTx.n = 1 : Enables pullup for nth bit
  PORTx.n = 0 : Disables pullup for nth bit, thus making it tristate
- Examples :
  - to make PA as input with pull-ups enabled and read data from PA

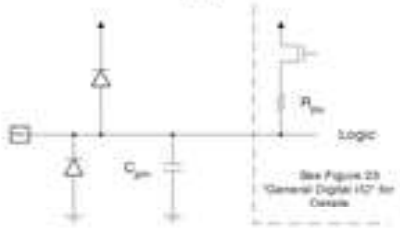
```
DDRA  = 0x00;    //make port a as input
PORTA = 0xFF;    //enable all pull-ups
y = PINA;        //read data from port a pins
```

  - to make PB as tri stated input

```
DDRB  = 0x00;  //make port b as input
PORTB = 0x00;  //disable pull-ups and make it
tri state
```

What is pull-up ?

- Pull-up resistor is used to ensure that tri-stated input always reads HIGH (1) when it is not driven by any external entity.



- Pull-up is very important when you are using tri-stated input buffers.
- Tri-state input pin offers very high impedance and thus can read as logic 1/ logic 0 because of minute static charges on nearby objects.
- Pin state changes rapidly and this change is unpredictable.
- This may cause your program to go haywire if it depends on input from such tri-state pin.

Input/Output Basics

Summary

- **Following table lists register bit settings and resulting function of port pins**

register bits → pin function↓	DDRx.n	PORTx.n	PINx.n

tri stated input	0	0	read data bit(s)
			x = PINx.n;
			y = PINx;

pull-up input	0	1	read data bit(s)
			x = PINx.n;
			y = PINx;

output	1	write data bit(s) PORTx.n = x; PORTx = y;	n/a

I/O Basics – Exercise

1. Configure PB as output port
2. Configure PC as tri-stated input and read value into variable x.
3. Configure PA as pullup input and read lower nibble into variable y and higher nibble into variable x.
4. Make higher nibble of PA as pullup inputs and lower nibble as output.
5. Read, only input pins of above mentioned port and get the binary number present on those pins into variable x.
6. Write four bit number 0x5 onto output pins of above mentioned port

Answers to Exercise

1. DDRB = 0xFF (or) 0b11111111 (or) 255;
2. DDRC = 0x00; PORTC = 0x00; x = PINC;
3. DDRA = 0x00; PORTA = 0xFF; y = PINA & 0b00001111; x = (PINA & 0b11110000) / 2^4
4. DDRA = 0x0F; PORTA = 0xF0;
5. x = (PINA & 0b11110000) / 2^4
6. PORTA = PORTA | 0x05;

RESET AND INTERRUPT HANDLING

- The AVR provides several different interrupt sources. These interrupts and the separate Reset Vector each have a separate program vector in the program memory space.
- All interrupts are assigned individual enable bits which must be written logic one together with the Global Interrupt Enable bit in the Status Register in order to enable the interrupt.

Interrupt Vectors in ATmega328 and ATmega328P

VectorNo.	Program Address(2)	Source	Interrupt Definition
1	0x0000(1)	RESET	External Pin, Power-on Reset, Brown-out Reset and Watchdog System Reset
2	0x0002	INT0	External Interrupt Request 0
3	0x0004	INT1	External Interrupt Request 1
4	0x0006	PCINT0	Pin Change Interrupt Request 0
5	0x0008	PCINT1	Pin Change Interrupt Request 1
6	0x000A	PCINT2	Pin Change Interrupt Request 2
7	0x000C	WDT	Watchdog Time-out Interrupt
8	0x000E	TIMER2 COMPA	Timer/Counter2 Compare Match A
9	0x0010	TIMER2 COMPB	Timer/Counter2 Compare Match B
10	0x0012	TIMER2 OVF	Timer/Counter2 Overflow
11	0x0014	TIMER1 CAPT	Timer/Counter1 Capture Event
12	0x0016	TIMER1 COMPA	Timer/Counter1 Compare Match A
13	0x0018	TIMER1 COMPB	Timer/Coutner1 Compare Match B
14	0x001A	TIMER1 OVF	Timer/Counter1 Overflow
15	0x001C	TIMER0 COMPA	Timer/Counter0 Compare Match A
16	0x001E	TIMER0 COMPB	Timer/Counter0 Compare Match B
17	0x0020	TIMER0 OVF	Timer/Counter0 Overflow
18	0x0022	SPI, STC	SPI Serial Transfer Complete
19	0x0024	USART, RX	USART Rx Complete
20	0x0026	USART, UDRE	USART, Data Register Empty
21	0x0028	USART, TX	USART, Tx Complete
22	0x002A	ADC	ADC Conversion Complete
23	0x002C	EE READY	EEPROM Ready
24	0x002E	ANALOG COMP	Analog Comparator
25	0x0030	TWI	2-wire Serial Interface
26	0x0032	SPM READY	Store Program Memory Ready

- The lower the address the higher is the priority level. RESET has the highest priority, and next is INT0 – the External Interrupt Request 0.

- When an interrupt occurs, the Global Interrupt Enable I-bit is cleared and all interrupts are disabled. The user software can write logic one to the I-bit to enable nested interrupts.

- All enabled interrupts can then interrupt the current interrupt routine. The I-bit is automatically set when a Return from Interrupt instruction – RETI – is executed.

Interrupt Response Time

- The interrupt execution response for all the enabled AVR interrupts is four clock cycles minimum.

- After four clock cycles the program vector address for the actual interrupt handling routine is executed.

- During this four clock cycle period, the Program Counter is pushed onto the Stack. The vector is normally a jump to the interrupt routine, and this jump takes three clock cycles.

- If an interrupt occurs during execution of a multi-cycle instruction, this instruction is completed before the interrupt is served.

- If an interrupt occurs when the MCU is in sleep mode, the interrupt execution response time is increased by four clock cycles. This increase comes in addition to the start-up time from the selected sleep mode.

- A return from an interrupt handling routine takes four clock cycles. During these four clock cycles, the Program Counter (two bytes) is popped back from the Stack, the Stack Pointer is incremented by two, and the I-bit in SREG is set.

Analog Comparator

- The Analog Comparator compares the input values on the positive pin AIN0 and negative pin AIN1.

- When the voltage on the positive pin AIN0 is higher than the voltage on the negative pin AIN1, the Analog Comparator output, ACO, is set (=1).

Analog-to-Digital Converter

Features:
• 10-bit Resolution
• 13 - 260µs Conversion Time
• 6 Multiplexed Single Ended Input Channels from the pins of Port A.
• 0 - VCC ADC Input Voltage Range
• Interrupt on ADC Conversion Complete
• Sleep Mode Noise Canceler

----------(THE END)----------