Real Time Embedded Systems

Course code: CoSc3026/SEng4033

By- Dr. Mohammad Nasre Alam

Chapter 2

Embedded System Architecture

Contents

1. Hardware Architecture of ES
2. ARM Cortex M0+ Hardware Overview.
3. Communication
4. ATmega32 microcontroller Architecture.
5. Assembly language Programming with ATmega32 Instruction Set
6. Programming in C to Interface peripherals, Interrupts, ISR and Timers.

1. Hardware Architecture of ES

Key Component of Embedded System

Microprocessor (CPU):

- The central unit that performs computation and control tasks.
- Examples: ARM Cortex-M, AVR, PIC, etc.

Memory:

- ROM (Read-Only Memory): Stores firmware (non-volatile).
- RAM (Random Access Memory): Stores data and variables (volatile).
- EEPROM/Flash: Non-volatile memory for data storage.

Input/Output Interfaces:

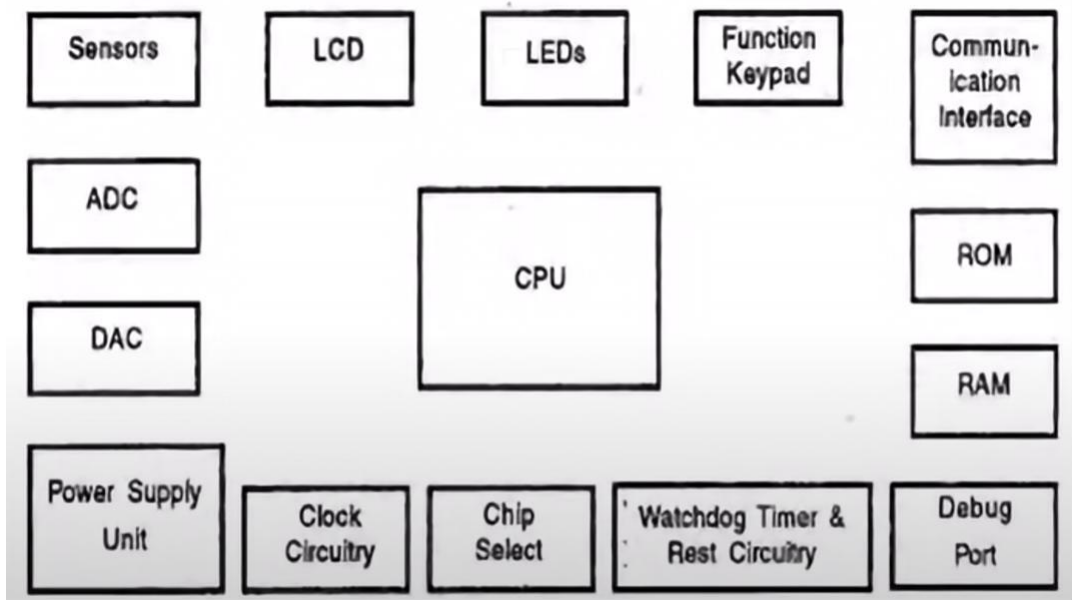
- Sensors/Actuators: Interaction with the physical world.
- Communication Ports: Serial (UART), USB, SPI, I2C, etc.
- Display/LEDs: Visual feedback for the user.

Power Supply:

- Provides the necessary voltage and current to the system.
- Can be battery-powered or use AC/DC adapters.

Peripherals:

- Specialized components like ADC/DAC converters, timers, and watchdogs.



Hardware Architecture of Embedded System

1. Hardware Architecture of ES Cont…

Block diagram of Processor

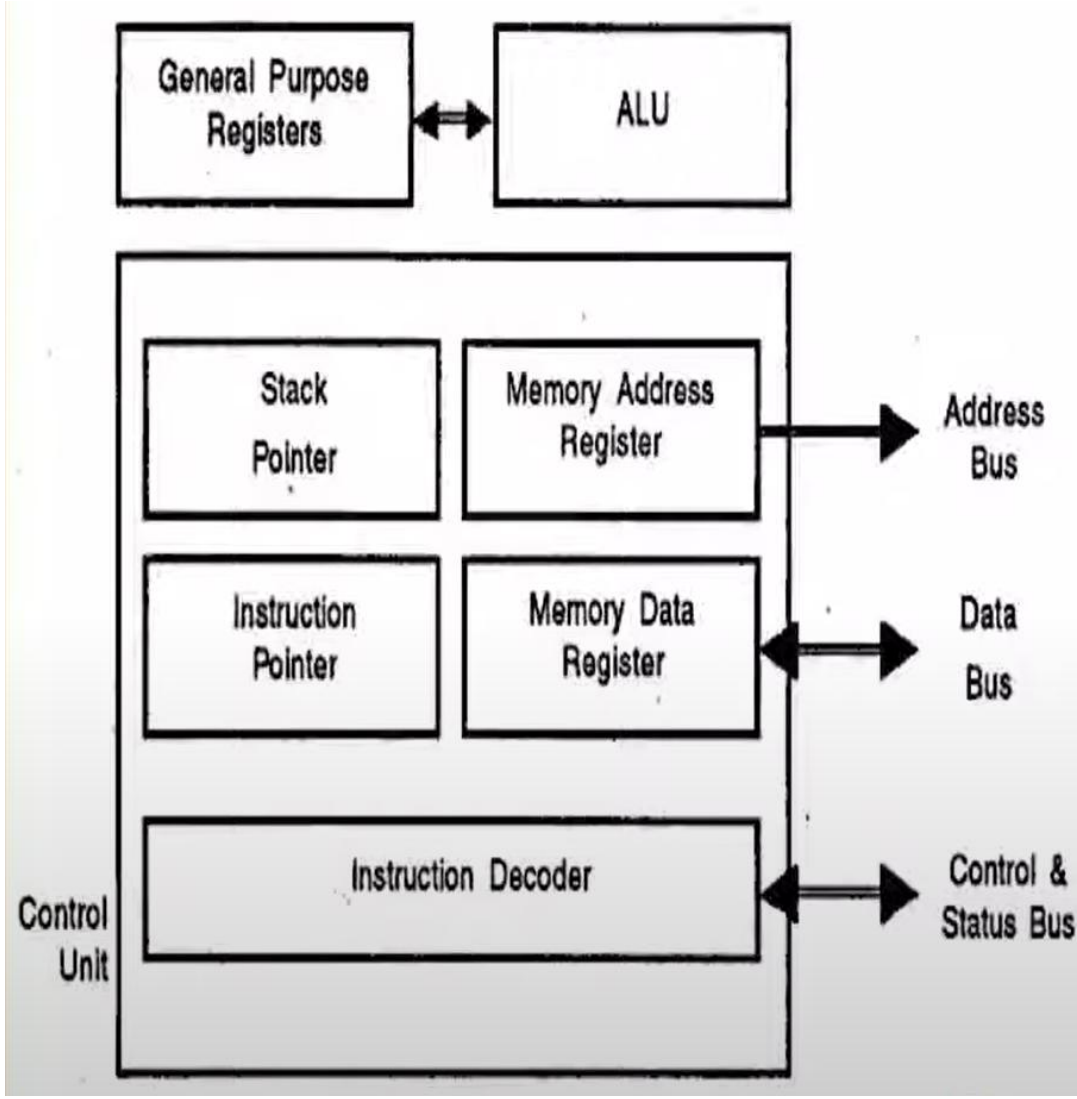
The CPU is the heart of the embedded system, responsible for executing instructions and managing control signals. It has two essential units –

Execution Unit (EU):

- **General Purpose Register:** Contain the current data and operands that are being manipulated by processor.
- **Arithmetic Logical Unit (ALU):** Responsible for performing Arithmetic and Logical instructions.

Control Unit:

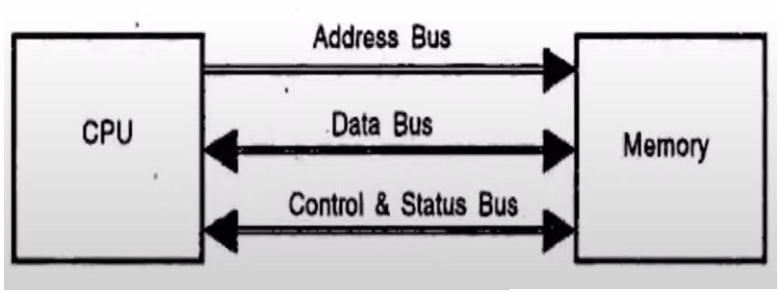
- **Stack Point:** It points to the instructions.
- **Memory Address Register:** It connected with address bus which is responsible for providing the address of that particular memory where addresses are stored.
- **Instruction Pointer:** It pointing to the next instruction.
- **Memory Data Register:** It is responsible for providing the data which is bidirectionally connected to the data bus.
- **Instruction Decoder:** It is useful for decoding the instruction that is also bidirectional connected to the control & Status Bus.



1. Hardware Architecture of ES Cont…

Architecture of Processor

To do function the processor communicates with other devices using three buses, a bus being a group of signals– Data Bus, Address Bus and Control and Status Bus.



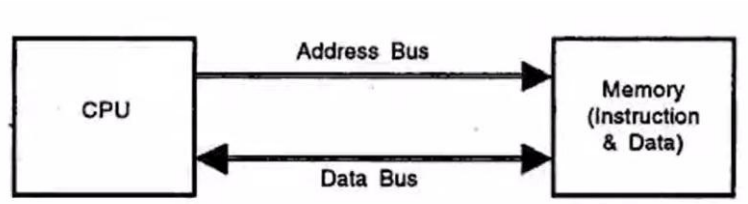


Fig: Von Neuman Architecture

1. **Fundamental Architecture-** The fundamental architecture of a computer system refers to the actual physical hardware components and how they are organized and connected to each other. It is three types -
   - i. **Von Neuman Architecture**:- CPU connected with memory and that memory contains instruction and data both.
   - ii. **Harvard Architecture**:- CPU connected with two memory, which is Program memory and Data memory. Program memory contains instructions and the data memory contains data only.
   - iii. **Super-Harvard Architecture**:- It is advanced version of Harvard architecture. It additionally include instruction cache in the CPU. The instruction cache is responsible for storing all the instructions.

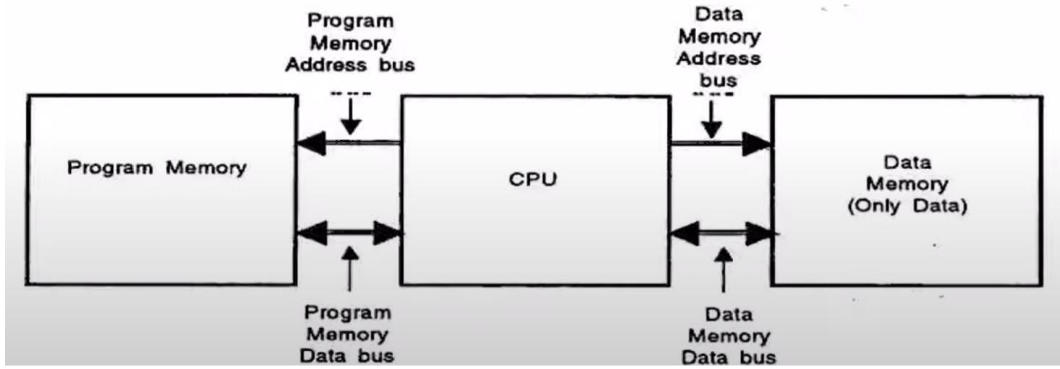


Fig: Harvard Architecture

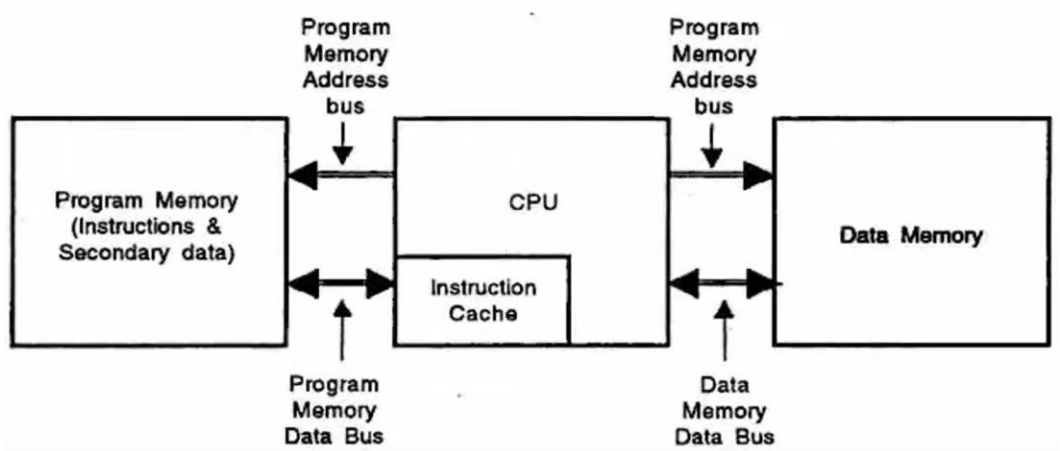


Fig: Super-Harvard Architecture

1. Hardware Architecture of ES Cont…

2. **Instruction Set Architecture (ISA):** - The ISA defines the set of instructions that the processor can execute. These instructions are the fundamental operations that the CPU performs, and they are used by software (like operating systems and applications) to interact with the hardware. The ARM ISA or the x86 instruction set used by Intel and AMD processors. It is categorized in two group-

   i. **Complex instruction set computers (CISC):-** A complex instruction set computer (CISC) is a computer architecture in which single instructions can execute several low-level operations (such as a load from memory, an arithmetic operation, and a memory store) or are capable of multi-step operations or addressing modes within single instructions.

   ii. **Reduced Instruction Set Computers (RISC)**:- A Reduced Instruction Set Computer is a type of microprocessor architecture that utilizes a small, highly-optimized set of instructions rather than the highly-specialized set of instructions.

Difference between the RISC and CISC Processors

RISC	CISC
It is a Reduced Instruction Set Computer.	It is a Complex Instruction Set Computer.
It requires multiple register sets to store the instruction.	It requires a single register set to store the instruction.
RISC has simple decoding of instruction.	CISC has complex decoding of instruction.
It uses a limited number of instructions	It uses a large number of instructions
It uses LOAD and STORE in the register-to-register a interaction of program.	It uses LOAD and STORE instruction in the memory-to-memory interaction of a program.
RISC has more transistors on memory registers.	CISC has transistors to store complex instructions.
The execution time of RISC is very short.	The execution time of CISC is longer.
It has fixed format instruction.	It has variable format instruction.
The program written for RISC architecture needs to take more space in memory.	Program written for CISC architecture tends to take less space in memory.
Example: ARM, Power Architecture, Alpha, AVR, ARC and the SPARC.	**Example:** VAX, Motorola 68000 family and the Intel x86 CPUs.

1. Hardware Architecture of ES Cont…

Memory: Information Storage Device.

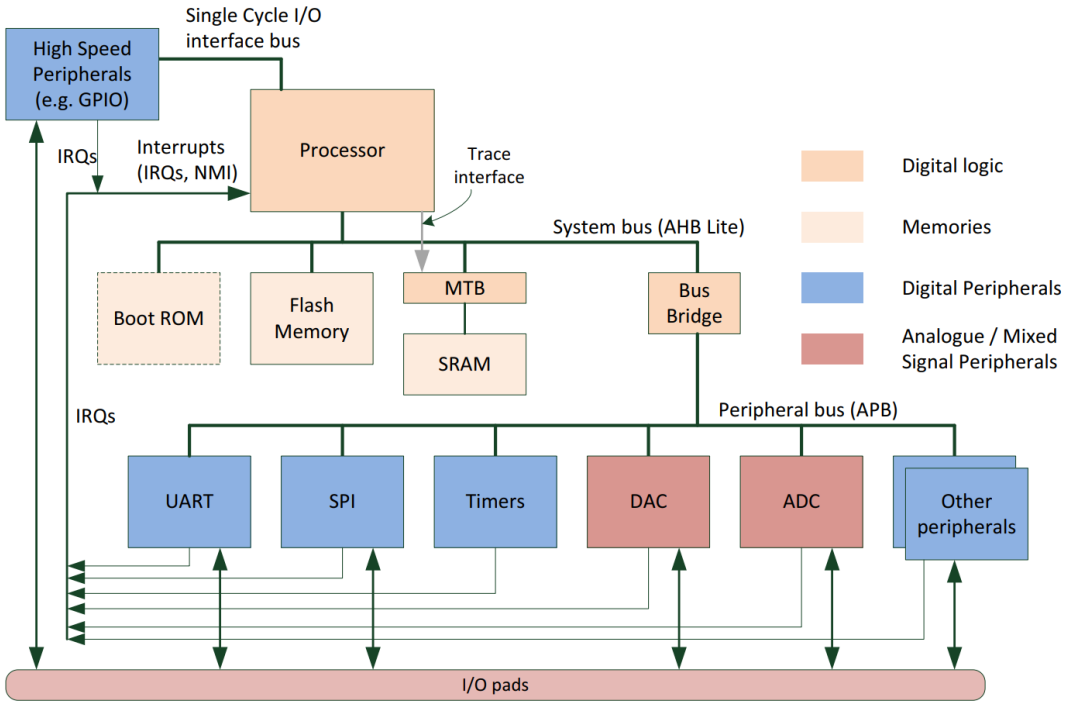
Memory Address	Data
0x201A 3241	10110111
0x201A 3242	01010100
.	.
.	.
.	.
.	.
.	.
0x201A 3244	1011 1101
0x201A 3243	1010 1101
0x201A 3242	1011 1100

- All Digital systems store information in binary digits called bits. These bits are used fo representing Operators, operands, and addresses. There are two states for a binary bit. A `1 represents the presence of a voltage and a `0' represents the absence of voltage. Thi representation is called positive logic.
- To store binary information, CMOS circuits can be built and a collection of a large numbe of such circuits is called memory. Memory stores data sequentially which is as shown i Figure.
- Based on read and write operations, memories are grouped in two categories namely-
  - i. **Read-Only Memory (ROM):** The memory that allows the processor to only read its contents is Read Only Memory (ROM). Another characteristic of ROM is that they can store information permanently even when no power is applied. The information in the ROM is therefore non volatile. Example- PROM, EPROM, EEPROM, Flash Memory.
  - ii. **Random Access Memory (RAM):** The memory which allows the processor to read from and write to its locations is known as Random Access Memory (RAM). One limitation of RAM is that information stored in it is lost as soon as the power applied to it is removed. On the other hand, RAM is not limited by the number of read and write cycles and is more suitable for storing data that is updated frequently. The read and write operations of RAMs are faster than those of ROMs. There are different types of RAMs namely-
    - a. **Dynamic RAM (DRAM):-** is the most commonly used type of RAM. Each memory cell of a DRAM, can store one bit of information. It is made up of two transistors and a capacitor. The transistor acts as a switch while the Capacitor holds the charge.
    - b. **Static RAM (SRAM):-** employs a flip-flop for storing a bit in a memory cell. A Flip-flop requires 4 to 6 transistors and does not require refreshing circuitry.

2. ARM Cortex M0+ Hardware Overview

ARM Cortex M0+ Hardware Overview

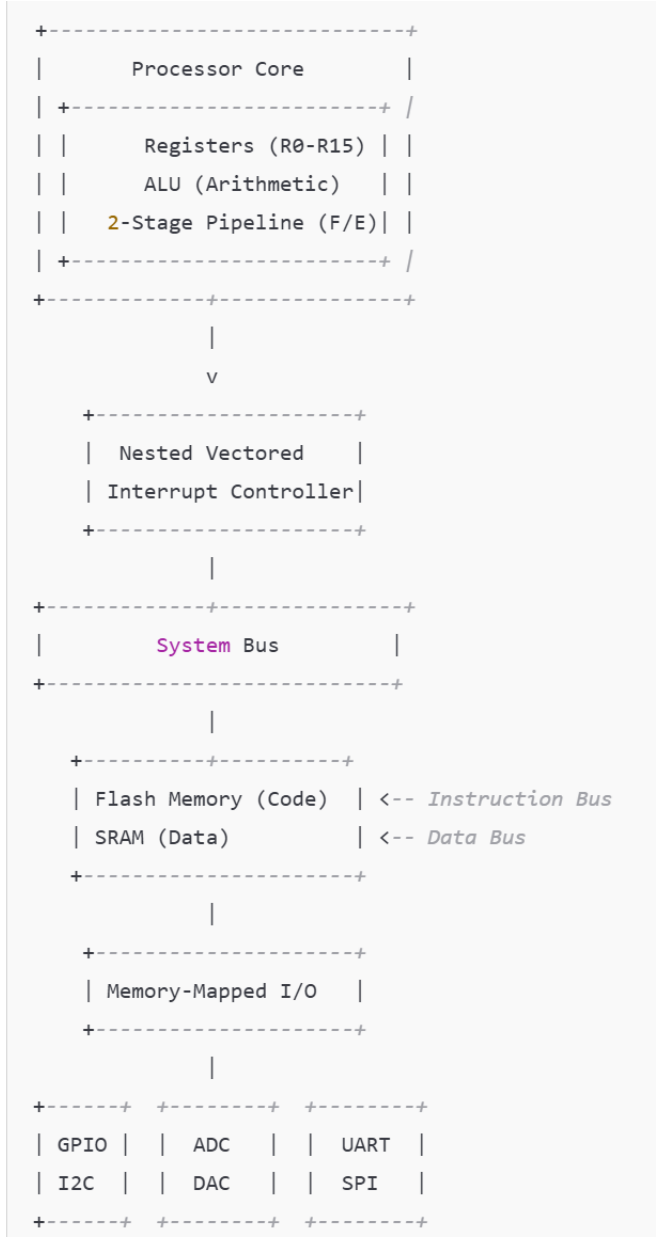
- The ARM Cortex-M0+ is a processor core that is integrated into microcontrollers by various manufacturers (e.g., STMicroelectronics, NXP, Silicon Labs).
- The specific hardware features like ports, GPIO, ADC, and DAC depend on the microcontroller implementation.
- It is 32-bit microprocessor core designed by ARM based on the ARMv6-M architecture has 2-stage pipeline (Fetch and Execute).
- Operates at clock frequencies up to 100MHz.
- It has low-power consumption, low-cost and high-efficiency operation.
- It is used in IoT devices, wearable electronics, motor control, automotive systems and industrial sensors.



2. ARM Cortex M0+ Hardware Overview Cont…

Core Components

1. **Processor Core-** Implements a 2-stage pipeline for efficient instruction execution.
   - **Fetch Stage:** Retrieves instructions from memory.
   - **Execute Stage**: Decodes and executes instructions.
2. **Register** – Operate as temporary storage within a program which is 32-bit wide.
   - **General purpose Register (R0-R12)** – total 13 registers used for storing data and intermediate results during program execution. Accessible by instructions like ADD, MOV, SUB, etc.
   - **Special Registers-** these are used for specific purpose and system level operation.
     - **R13 (Stack Pointer (SP)):-** Points to the top of the current stack frame. Cortex-M0+ supports two stack pointers- Main (MSP) and Process (PSP).
     - **R14 (Link Register (LR)):-** Holds the return address for function calls.
     - **R15 (Program Counter (PC)):-** Points to the address of the next instruction to execute.
     - **Program Status Register (PSR):-** Contains system state and condition flags (e.g. Zero (Z), Negative (N), Overflow(V) and Carry (C)).
3. **ALU (Arithmetic Logic Unit)** – Handles arithmetic and logical operations.



2. ARM Cortex M0+ Hardware Overview Cont…

Nested Vectored Interrupt Controller (NVIC)

- Manages up to 32 interrupts with priority levels.
- Fast interrupt handling.
- Supports tail-chaining for rapid interrupt processing.
- Ensures deterministic latency, critical for time-sensitive tasks.

Memory Interface

It has Havard Architecture i.e. separate instruction and data buses for concurrent access to code and data memory. Supports Code Memory, Data Memory (SRAM), and Peripherals Memory.

- **Code Memory –** Stores instructions (typically Flash).
- **Data Memory –** SRAM for variables and stack.
- **Peripheral Memory –** Mapped regions for peripherals like GPIO, ADC and Communication interfaces.

Ports

Ports is an ARM Cortex-M0+ microcontroller refer to the hardware interfaces used to interact with external devices. These ports allow the processor to connect with sensors, actuators, communication devices and other peripherals.

1. **General Purpose I/O Ports (GPIO) –** It is configurable as either input or output. They allow the microcontroller to interface directly with digital signals. Each GPIO pin can be individually controlled for input or output.
2. **Analog I/O Ports**
   - **Analog – to – Digital Converter (ADC) -** Converts analog signals (e.g. from a temperature sensor) into digital values for processing. Ports connected to ADC channels are used for analog inputs.
   - **Digital – to – Analog Converter (DAC) –** Converts digital values to analog signals e.g. for controlling actuators or sound output.

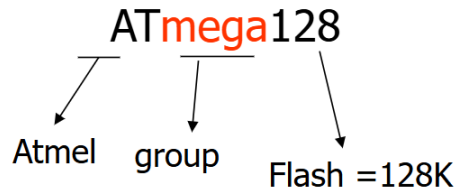
3. Communication

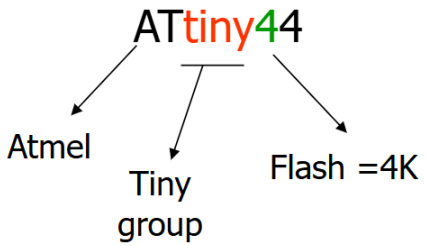
Contex-M0+ microcontroller supports multiple communication protocols, making it highly versatile for embedded applications. These communication peripherals allow data exchange wit sensors, actuators, other microcontrollers and external devices. The types of communication in Cortex-M0+ are –

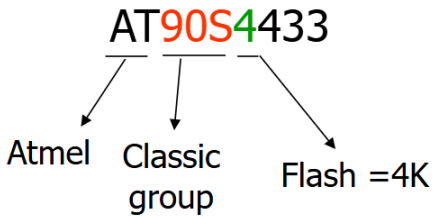
1. **Serial Communication Protocols-**
   a. **UART (Universal Asynchronous Receiver- Transmitter) –** Use for serial communication. Transmit (TX) and Receive (RX) pins used to send and receive data.
   b. **SPI (Serial Peripheral Interface) –** High speed synchronous communication with multiple devices. Ports are –
      - **MOSI:** Master Out, Slave In.
      - **MISO:** Master In, Slave Out.
      - **SCK:** Clock.
      - **SS/CS:** Slave Select/Chip Select.
   c. **I^2C (Inter-Integrated Circuit)/TWI (Two Wire Interface) –** It is two-way synchronous protocol, uses only SDA (Serial Data) and SCL (Serial Clock) line. It supports Multi-master, multi-slave. Use for EEPROMs, temperature sensors and real time clocks.
   d. **CAN (Controller Area Network) –** Robust protocol designed for automotive and industrial systems. It is use for error detection and fault tolerance.
2. **Wireless Communication (via External Modules)** – Although Contex-M0+ doesn't have built-in wireless communication, it can interface with external wireless modules:
   a. **Bluetooth:** Using UART (e.g. HC-05 module).
   b. **Wi-Fi:** Using SPI or UART (e.g. ESP8266, ESP32).
   c. **ZigBee/LoRa:** Using UART or SPI for long-range communication.
3. **Parallel Communication Protocols** – Cortex-M0+ doesn't include dedicated parallel communication peripherals, it can implement parallel communication through its General Purpose Input/Output (GPIO) pins.
4. **Ethernet Communication** – Many Cortex-M0+ microcontrollers equipped with a MAC (Media Access Controller) or an external MAC/PHY interface for Ethernet communication.

4. ATmega32 microcontroller Architecture

- ATmega32 microcontroller is a member of AVR architecture.
- AVR architecture was designed by two students Alf-Egil Bogen and Vegard Wollan at the Norwegian Institute of Technology and then was bought and developed by Atmel Corporation in 1996.
- AVR stands for Advanced Virtual RISC or some call it as Alf and Vegard RISC.
- AVR is super Harward architecture 8-bit RISCS single chip microcontroller.
- AVR are generally classified into four board groups:
  - Classic – e.g. AT90S2313, AT90S443
  - Mega – e.g. ATmega8, ATmega32, ATmega128
  - Tiny – e.g. ATtiny13, ATtiny25
  - Special Purpose – e.g. AT90PWM216, AT90USB1287
- ATmega32 is most widely used and available.



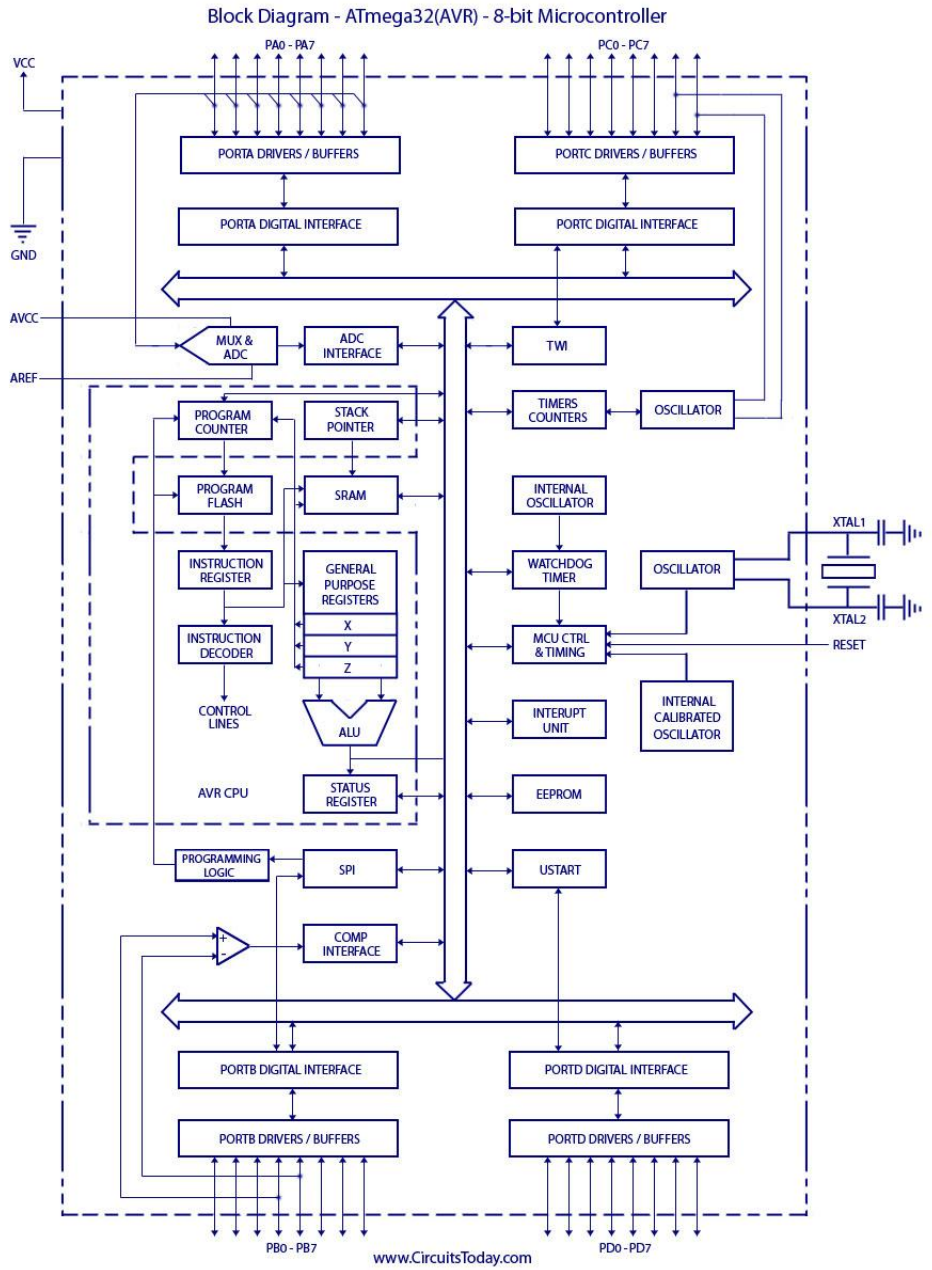




4. ATmega32 microcontroller Architecture Cont..

Pins and Ports Overview

- Atmega32 has got total 40 pins.
- Two for Power (pin no.10: +5v, pin no. 11: ground),
- two for oscillator (pin 12, 13), one for reset (pin 9),
- three for providing necessary power and reference voltage to its internal ADC.
- One for reset.
- 32 (4×8) I/O pins.

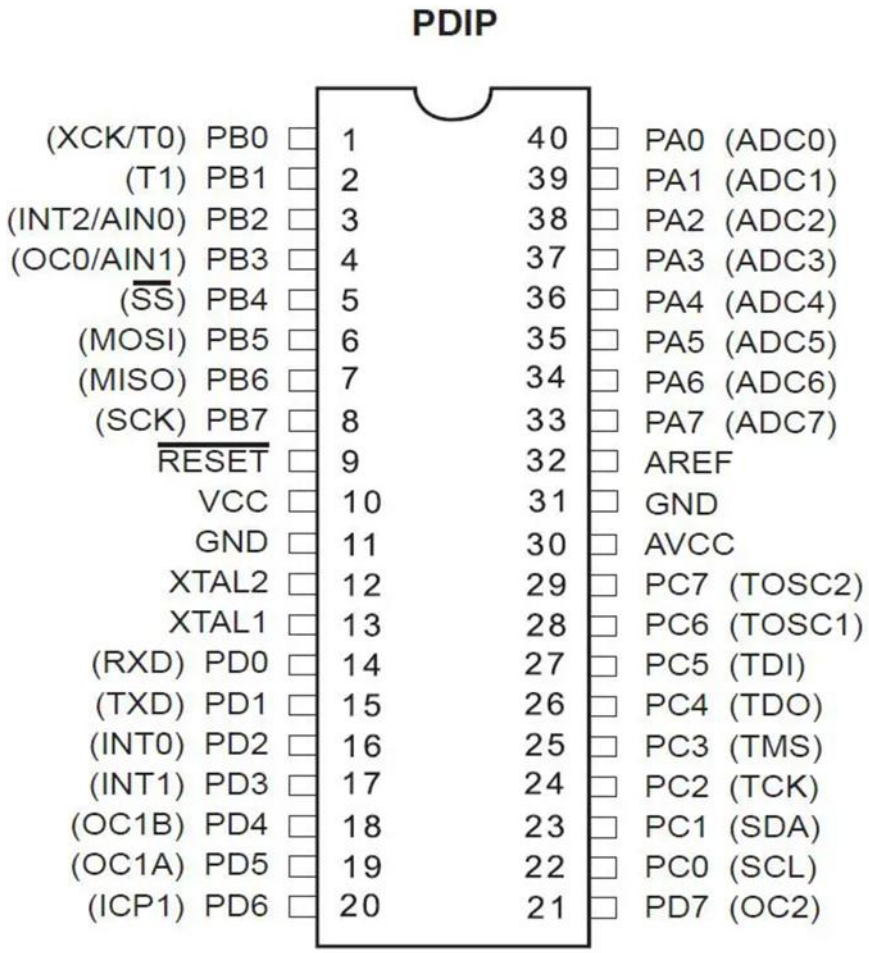


4. ATmega32 microcontroller Architecture Cont..

Pins and Ports Overview

- **VCC & GND**: Digital supply voltage for ATmega32. Typically operating Voltage 2.7 – 5.5V. Should never exceed 6.0 V.
- **XTAL1 &Xtal2:** Connections to ceramic/crystal on-chip oscillator for generating internal clocks.
- **Reset:** A low level on this pin for longer than 1.5 micro sec length will generate a reset. During Reset, all I/O Register are set to their initial values, & the program starts execution from the Reset vector.
- **AVCC**: The supply voltage pin for the ADC, should be externally connected to VCC.
- **AREF**: The analog reference pin for the ADC, connect to suitable voltage if external reference is chosen.
- **Port A (PA0-PA 7):** Serves as an 8-bit bi-directional digital I/O port. Port A can be programmed to serve as alternate function as analog input for the ADC.
- **Port B (PB0- PB 7):** Serves as an 8-bit bi-directional digital I/O port with optional internal pull-ups. Port B pins are tri-stated when reset. Port B can be programmed to serve as alternate function.
- **Port C (PC0 – PC7):** Serves as an 8-bit bi-directional digital I/O pins are tri-stated when reset/ Port C can be programmed to serve as alternate function.
- **Port D (PD0 – PD7):** Serve as an 8-bit bi-directional I/O port with optional internal pull-ups. Port D pins are tri-stated when reset. Port D can be programmed to serve as alternate function.

Atmega32 pin diagram



5. Assembly language Programming with ATmega32 Instruction Set

AVR Data Types

- Hex Number – There is two way to represent hex number:
    1. Put 0x or 0X in the front of the number – LDI R16, 0x99 OR LDI R16, 0X99.
    2. Put $ in front of the number – LDI R28, $75.
- Binary Numbers – There is only one way to represent binary numbers in an AVR assembler.

  LDI R16, 0b10011001 or LDI R16 0B10011001
- Decimal Numbers – To indicate decimal numbers in an AVR assembler we simply use the decimal and nothing before or after it.

  LDI R17, 12
- ASCII Characters – To represent ASCII data in an AVR assembler we use single quotes as follows

  LDI R23, ‘2’ .

5. Assembly language Programming with ATmega32 Instruction Set Con..

Assembler Directives

- While the instructions tell the CPU what to do, directives give direction to the assembler.
- These are also called as pseudo instructions.
- The following are mostly used assembler directives in AVR programming.
  - **.EQU-** This directive is use to define/name a constant or a fixed address. The .EQU directive does not set aside storage for a data item/address label, but associates a constant number with a data or an address label, so that when the label appears in the program, its constant will be substituted for the label.

    Example

    .EQU COUNT = 0x25……. LDI R21, COUNT

    .EQU SUM = 0x12…….LDI R18, SUM
  - **.SET –** This directive is used to define a constant value or a fixed address. .SET and .EQU directives are identical. The only difference is that the value assigned by the .SET directive may be reassigned later.
  - **.ORG-** The .ORG directive is used to indicate the beginning of t he address. It can be used for both code and data.

    Example

    .ORG ox200 …… start ad address 200.
  - **.INCLUDE-** The include directive tells the AVR assembler to add the contents of a file to the program.

    Example

    When we want to use ATmega32, we must write the following instruction at the beginning of your program: .INCLUDE "M32DEF.INC".
  - **.DB –** This directive is used to define byte for the given address location or define set of bytes for the consecutive memory locations. The .DB directive reserves memory resource in the program memory or the EEPROM memory.
  - **.DEF-** The .DEF directive allows the registers to be referred to through symbols. A defined symbol can be used in the rest of the program to refer to the register it is assigned to. A register can have several symbolic names attached to it. The symbol can be redefined later in the program.

    Example

    .DEF temp = R16
    .DEF ior = R0
  - **.UNDEF** – The UNDEF directive is used to un-define a symbol previously defined with the .DEF directive. This provides a way to obtain a simple scoping of register definitions, to avoid warnings about register reuse.

    Example

    .DEF var1 = R16
    LDI var1, 0x20….. Do something more with var1
    .UNDEF var1

5. Assembly language Programming with ATmega32 Instruction Set Con..

Instruction Set

1. **Arithmetic and Logical Instructions**
2. **Branch /Jump/Call Instructions**
3. **Data Transfer Instructions.**

1. Arithmetic and Logical Instructions

ADD (Addition without Carry)

- **Description:** Adds two registers without the C flag and places the result in the destination register Rd.
- **Operation:** Rd ← Rd + Rr
- **Syntax:** ADD Rd, Rr 0 ≤ d ≤ 31, 0 ≤ r ≤ 31
- **Cycles:** 1

```
ADD R1,R2          ; Add r2 to r1 (r1=r1+r2)
ADD R28,R28        ; Add r28 to itself (r28=r28+r28)
```

ADC (Addition with Carry)

- **Description:** Adds two registers and the contents of the C flag and places the result in the destination register Rd.
- **Operation:** Rd ← Rd + Rr + C
- **Syntax:** ADC Rd, Rr 0 ≤ d ≤ 31, 0 ≤ r ≤ 31
- **Cycles:** 1

```
Add R1:R0 to R3:R2
ADD R2,R0          ; Add low byte
ADC R3,R1          ; Add with carry high byte
```

ADIW (Add Immediate to Word)

- **Description:** Adds an immediate value (0-63) to a register pair and places the result in the register pair. This instruction operates on the upper THREE register pairs, and is well suited for operations on the pointer registers.
- This instruction is not available in all devices.
- **Operation:** [Rd+1:Rd] ← [Rd+1:Rd] + K
- **Syntax:** ADIW Rd+1:Rd, K

  where, d={26,28,30}, 0 <= K <= 63

SUB (Subtract without Carry)

- **Description:** Subtracts two registers and places the result in the destination register Rd.
- **Operation:** Rd ← Rd - Rr
- **Syntax:** SUB Rd, Rr where, 0 ≤ d ≤ 31, 0 ≤ r ≤ 31
- **Cycles:** 1
- **Example:**

```
SUB R13,R12        ; Subtract R12 from R13
BRNE noteq         ; Branch if R12⇔R13
...
noteq: nop         ; Branch destination (do nothing)
```

SBC (Subtract with Carry)

- **Description:** Subtracts two registers and subtracts with the C flag and places the result in the destination register Rd.
- **Operation:** Rd ← Rd - Rr - C
- **Syntax:** SBC Rd, Rr where, 0 ≤ d ≤ 31, 0 ≤ r ≤ 31
- **Cycles:** 1
- **Example:**

```
Subtract R1:R0 from R3:R2

SUB R2,R0          ; Subtract low byte
SBC R3,R1          ; Subtract with carry high byte
```

SBIW (Subtract Immediate from Word)

- **Description:** Subtracts an immediate value (0-63) from a register pair and places the result in the register pair. This instruction operates on the upper four register pairs, and is well suited for operations on the pointer registers.
- This instruction is not available in all devices.
- **Operation:** [Rd+1:Rd] ← [Rd+1:Rd] - K
- **Syntax:** SBIW Rd+1:Rd, K d={26,28,30}, 0 ≤ K ≤ 63

5. Assembly language Programming with ATmega32 Instruction Set Con..

1. Arithmetic and Logical Instructions Cont..

SUBI (Subtract Immediate)

- **Description:** Subtracts a register and a constant and places the result in the destination register Rd. This instruction is working on Register R16 to R31 and is very well suited for operations on the X, Y and Z pointers.
- **Operation:** Rd ← Rd - K
- **Syntax:** SUBI Rd, K where, 16 ≤ d ≤ 31, 0 ≤ K ≤ 255

SBCI (Subtract Immediate with Carry)

- **Description:** Subtracts a constant from a register and subtracts with the C flag and places the result in the destination register Rd.
- **Operation:** Rd ← Rd - K - C
- **Syntax:** SBCI Rd, K where, 16 ≤ d ≤ 31, 0 ≤ K ≤ 255
- **Cycles:** 1
- **Example:**

Subtract $4F23 from R17:R16

```
SUBI R16,$23  ; Subtract low byte
SBCI R17,$4F  ; Subtract with carry high byte
```

MUL (Multiply Unsigned)

- **Description:** This instruction performs *8-bit × 8-bit → 16-bit* unsigned multiplication.
- The **multiplicand Rd** and the **multiplier Rr** are two registers containing unsigned numbers. The 16-bit unsigned product is placed in R1 (high byte) and R0 (low byte). Note that if the multiplicand or the multiplier is selected from R0 or R1 the result will overwrite those after multiplication.
- **Operation:** R1:R0 ← Rd × Rr (unsigned ← unsigned × unsigned)
- **Syntax:** MUL Rd, Rr where, 0 ≤ d ≤ 31, 0 ≤ r ≤ 31

MULS (Multiply Signed)

- **Description:** This instruction performs 8-bit × 8-bit → 16-bit signed multiplication.
- The multiplicand Rd and the multiplier Rr are two registers containing signed numbers. The 16-bit signed product is placed in R1 (high byte) and R0 (low byte).
- **Operation:** R1:R0 ← Rd × Rr (signed ← signed × signed)
- **Syntax:** MULS Rd, Rr where,16 ≤ d ≤ 31, 16 ≤ r ≤ 31

No DIVIDE instruction in AVR

- AVR has no instruction for divide operation.
- The division operation can be performed by doing subtraction operation repeatedly.
- The **quotient is the number of times we subtracted** and the **remainder is in the register upon completion.**

INC (Increment)

- **Description:** Adds one '1' to the contents of register Rd and places the result in the destination register Rd.
- **Operation:** Rd ← Rd + 1
- **Syntax:** INC Rd where, 0 ≤ d ≤ 31
- **Cycles:** 1
- **Example:**

```
INC R22     ; increment R22
```

DEC (Decrement)

- **Description:** Subtracts one '1' from the contents of register Rd and places the result in the destination register Rd.
- **Operation:** Rd ← Rd - 1
- **Syntax:** DEC Rd where, 0 ≤ d ≤ 31
- **Cycles:** 1
- **Example:**

```
DEC R17 ; Decrement R17
```

5. Assembly language Programming with ATmega32 Instruction Set Con..

1. Arithmetic and Logical Instructions Cont..

AND (Logical AND)

- **Description:** Performs the logical AND between the contents of register Rd and register Rr and places the result in the destination register Rd.
- **Operation:** Rd ← Rd • Rr
- **Syntax:** AND Rd, Rr where, 0 ≤ d ≤ 31, 0 ≤ r ≤ 31
- **Cycles:** 1
- **Example:**

```
AND R2,R3       ; Bitwise and R2 and R3, result in R2

LDI R16,1       ; Set bitmask 0000 0001 in R16
AND R2, R16     ; Isolate bit 0 in R2
```

ANDI (Logical AND with Immediate)

- **Description:** Performs the logical AND between the contents of register Rd and a constant and places the result in the destination register Rd.
- **Operation:** Rd ← Rd • K
- **Syntax:** ANDI Rd, K where, 16 ≤ d ≤ 31, 0 ≤ K ≤ 255
- **Cycles:** 1
- **Example:**

```
ANDI R17,$0F    ; Clear upper nibble of r17
ANDI R18,$10    ; Isolate bit 4 in r18
ANDI R19,$AA    ; Clear odd bits of r19
```

OR (Logical OR)

- **Description:** Performs the logical OR between the contents of register Rd and register Rr and places the result in the destination register Rd.
- **Operation:** Rd ← Rd v Rr
- **Syntax:** OR Rd, Rr where, 0 ≤ d ≤ 31, 0 ≤ r ≤ 31
- **Cycles:** 1
- **Example:**

```
OR R15,R16 ; Do bitwise 'OR' between registers R15 & R16
```

ORI (Logical OR with Immediate)

- **Description:** Performs the logical OR between the contents of register Rd and a constant and places the result in the destination register Rd.
- **Operation:** Rd ← Rd v K
- **Syntax:** ORI Rd, K where, 16 ≤ d ≤ 31, 0 ≤ K ≤ 255
- **Cycles:** 1
- **Example:**

```
ORI R16,$F0     ; Set high nibble of r16
ORI R17,1       ; Set bit 0 of r17
```

EOR - Exclusive OR

- **Description:** Performs the logical EOR between the contents of register Rd and register Rr and places the result in the destination register Rd.
- **Operation:** Rd ← Rd ⊕ Rr
- **Syntax:** EOR Rd, Rr where, 0 ≤ d ≤ 31, 0 ≤ r ≤ 31
- Cycles: 1
- Example:

```
EOR R4,R4       ; Clear R4
EOR R0,R22      ; Bitwise exclusive or between R0 and R22
```

COM (One's Complement)

- **Description:** This instruction performs a one's complement of register Rd.
- **Operation:** Rd ← $FF - Rd
- **Syntax:** COM Rd where, 0 ≤ d ≤ 31
- **Cycles:** 1
- **Example:**

```
COM R4      ; Take one's complement of R4
```

NEG (Two's Complement)

- **Description:** Replaces the contents of register Rd with its two's complement; the value $80 is left unchanged.
- **Operation:** Rd ← $00 - Rd
- **Syntax:** NEG Rd where, 0 ≤ d ≤ 31
- Example:

```
NEG R11     ; Take two's complement of R11
```

CLR (Clear Register)

- **Description:** Clears a register. This instruction performs an **Exclusive OR** between a register and itself. This will clear all bits in the register.
- **Operation:** Rd ← Rd ⊕ Rd
- **Syntax:** CLR Rd where, 0 ≤ d ≤ 31
- **Cycles:** 1
- **Example:**

```
CLR R18     ; clear R18
```

SER (Set all bits in Register)

- **Description:** Loads $FF directly to register Rd.
- **Operation:** Rd ← $FF
- **Syntax:** SER Rd where, 16 ≤ d ≤ 31
- **Cycles:** 1
- **Example:**

```
SER R17     ; Set R17
```

5. Assembly language Programming with ATmega32 Instruction Set Con..

2. Branch /Jump/Call Instructions

BRBS (Branch if Bit in SREG is Set)

- **Description:** Conditional relative branch. Tests a single bit in SREG and ***branches relatively to PC*** if the bit is set.
- This instruction branches relatively to PC in either direction (**PC - 63 ≤ destination ≤ PC + 64**). The parameter k is the offset from PC and is represented in two's complement form.
- **Operation:** If SREG(s) = 1 then PC ← PC + k + 1, else PC ← PC + 1
- **Syntax:** **BRBS s, k** where, 0 ≤ s ≤ 7, -64 ≤ k ≤ +63

BRBS (Branch if Bit in SREG is Set)

- **Cycles:** 1 if condition is false 2 if condition is true
- **Example:**

```
BSTR 0,3          ; Load T bit with bit 3 of r0
BRBS 6, bitset    ; Branch T bit was set
...
bitset: nop       ; Branch destination (do nothing)
```

BRBC (Branch if Bit in SREG is Cleared)

- **Description:** Conditional relative branch. Tests a single bit in SREG and ***branches relatively to PC*** if the bit is cleared.
- This instruction branches relatively to PC in either direction (**PC - 63 ≤ destination ≤ PC + 64**). The parameter k is the offset from PC and is represented in two's complement form.
- **Operation:** If SREG(s) = 1 then PC ← PC + k + 1, else PC ← PC + 1
- **Syntax:** **BRBC s, k** where, 0 ≤ s ≤ 7, -64 ≤ k ≤ +63

BREQ (Branch if Equal)

- **Description:** Conditional relative branch. Tests the Zero flag (Z) and ***branches relatively to PC if Z is set.*** If the instruction is executed immediately after any of the instructions CP, CPI, SUB or SUBI, the branch will occur if and only if the unsigned or signed binary number represented in Rd **(Rd=Rr)** was **equal** to the unsigned or signed binary number represented in Rr.
- This instruction branches relatively to PC in either direction (**PC - 63 ≤ destination ≤ PC + 64**). The parameter k is the offset from PC and is represented in two's complement form.
- **Operation:** If Rd = Rr (Z = 1) then PC ← PC + k + 1, else PC ← PC + 1
- **Syntax:** **BREQ k** -64 ≤ k ≤ +63; PC ← PC + k + 1, if condition is true and PC ← PC + 1, if condition is false
- **Cycles:** 1 if condition is false 2 if condition is true
- **Example:**

```
CP R1, R0         ; Compare registers R1 and R0
BREQ equal        ; Branch if registers equal
...
equal: nop        ; Branch destination (do nothing)
```

BRNE (Branch if Not Equal)

- **Description:** Conditional relative branch. Tests the Zero flag (Z) and ***branches relatively to PC if Z is cleared.*** If the instruction is executed immediately after any of the instructions CP, CPI, SUB or SUBI, the branch will occur if and only if the unsigned or signed binary number represented in Rd **(Rd≠Rr)** was **not equal** to the unsigned or signed binary number represented in Rr.
- This instruction branches relatively to PC in either direction (**PC - 63 ≤ destination ≤ PC + 64**). The parameter k is the offset from PC and is represented in two's complement form.
- **Operation:** If Rd ≠ Rr (Z = 1) then PC ← PC + k + 1, else PC ← PC + 1
- **Syntax:** **BRNE k** -64 ≤ k ≤ +63; PC ← PC + k + 1, if condition is true and PC ← PC + 1, if condition is false
- **Cycles:** 1 if condition is false 2 if condition is true
- **Example:**

```
CPI R16, 5        ; Compare register R16 with 5
BRNE nequal       ; Branch if they are not equal
...
nequal: nop       ; Branch destination (do nothing)
```

BRCS - Branch if Carry Set

- **Description:**
- Conditional relative branch. ***Tests the Carry flag (C) and branches relatively to PC if C is set.*** This instruction branches relatively to PC in either direction (***PC - 63 ≤ destination ≤ PC + 64***). The parameter k is the offset from PC and is represented in two's complement form.
- **Operation:** If C = 1 then PC ← PC + k + 1, else PC ← PC + 1
- **Syntax:** **BRCS k** where,-64 ≤ k ≤ +63

BRCC - Branch if Carry Cleared

- **Description:**
- Conditional relative branch. Tests the Carry flag (C) and branches relatively to PC ***if C is cleared.*** This instruction branches relatively to PC in either direction (***PC - 63 ≤ destination ≤ PC + 64***). The parameter k is the offset from PC and is represented in two's complement form.
- **Operation:** If C = 0 then PC ← PC + k + 1, else PC ← PC + 1
- **Syntax:** **BRCC k** where, -64 ≤ k ≤ +63

BRIE - Branch if Global Interrupt is Enabled

- **Description:**
- Conditional relative branch. ***Tests the Global Interrupt flag (I)*** and branches relatively to PC if I is set. This instruction branches relatively to PC in either direction (***PC - 63 ≤ destination ≤ PC + 64***). The parameter k is the offset from PC and is represented in two's complement form.
- **Operation:** If I = 1 then PC ← PC + k + 1, else PC ← PC + 1
- Syntax: **BRIE k** where, -64 ≤ k ≤ +63

BRID - Branch if Global Interrupt is Disabled

- **Description:**
- Conditional relative branch. ***Tests the Global Interrupt flag (I)*** and branches relatively to PC if I is cleared. This instruction branches relatively to PC in either direction (***PC - 63 ≤ destination ≤ PC + 64***). The parameter k is the offset from PC and is represented in two's complement form.
- **Operation:** If I = 0 then PC ← PC + k + 1, else PC ← PC + 1
- **Syntax:** **BRID k** where, -64 ≤ k ≤ +63

5. Assembly language Programming with ATmega32 Instruction Set Con..

2. Branch /Jump/Call Instructions Cont..

CP- Compare

- **Description:**
- This instruction performs a compare between two registers Rd and Rr. None of the registers are changed. All conditional branches can be used after this instruction.
- **Operation:** Rd - Rr
- **Syntax:** CP Rd, Rr where, 0 ≤ d ≤ 31, 0 ≤ r ≤ 31
- **Cycles:** 1

CPC- Compare with Carry

- **Description:**
- This instruction performs a compare between two registers Rd and Rr and also takes into account the previous carry. None of the registers are changed. All conditional branches can be used after this instruction.
- **Operation:** Rd - Rr - C
- **Syntax:** CPC Rd, Rr where, 0 ≤ d ≤ 31, 0 ≤ r ≤ 31
- **Cycles:** 1

CPI- Compare with Immediate

- **Description:**
- This instruction performs a compare between register Rd and a constant. The register is not changed. All conditional branches can be used after this instruction.
- **Operation:** Rd - K
- **Syntax:** CPI Rd, K where,16 ≤ d ≤ 31, 0≤ K ≤ 255
- **Cycles:** 1

5. Assembly language Programming with ATmega32 Instruction Set Con..

2. Branch /Jump/Call Instructions Cont..

JMP (Jump)

- **Description:** It is an unconditional jump that can jump to any address within the entire 4M (words) program memory of AVR.
- It is a 4-byte instruction in which 10bits are used for opcode and the other 22 bits (0×000000 to 0×3FFFFF) represent the target address location.
- **Operation:** PC ← k
- **Syntax:** JMP k where, 0 ≤ k < 4M
- **Cycles:** 3

```
JMP farplc        ; Unconditional jump ...
farplc:nop        ; Jump destination (do nothing)
```

RJMP (Relative Jump)

- **Description:** It is an unconditional jump which jumps to an address within PC - 2K +1 and PC + 2K (words). In the assembler, labels are used instead of relative operands.
- It is 2-byte instruction, in which the first 4-bits are opcode and the remaining 12-bits is the relative address of the target location.
- The relative address range of 0×000 to 0×FFF is divided into forward and backward jumps; **within the -2048 to +2047** words of memory relative to current PC value.
- **Operation:** PC ← PC + k + 1
- **Syntax:** RJMP k -2K ≤ k < 2K
- **Cycles:** 2
- **Example:**

```
CPI R16,$42          ; Compare R16 to $42
BRNE error           ; Branch if R16 ⇔ $42
RJMP ok              ; Unconditional branch
error: ADDr16,R17    ; Add R17 to R16
INC R16              ; Increment R16
ok: nop              ; Destination for RJMP(do nothing)
```

IJMP (Indirect Jump)

- **Description:** Indirect jump to the address ***pointed by the Z (16 bits) pointer register*** in the register file, unconditionally. The Z pointer register is 16 bits wide and allows jump within the 128K bytes section of program memory.
- In other jump instruction the target address is static, but in the case of IJMP the target ***address can be changed dynamically*** by changing the Z-registers content through the program.
- **Operation:** PC ← Z(15:0) Devices with 16 bits PC, 128K bytes program memory maximum.
- **Syntax:** IJMP
- **Cycles:** 2
- **Example:**

```
MOV R30,R0       ; Set offset to jump table
IJMP             ; Jump to routine pointed to by R31:R30
```

5. Assembly language Programming with ATmega32 Instruction Set Con..

2. Branch /Jump/Call Instructions Cont..

CALL (Long Call to a Subroutine)

- **Description:** Calls to a subroutine within the entire program memory. The ***return address (address next to the instruction after the CALL) will be stored onto the stack.*** Then the execution jumps to the subroutine called.
- The stack pointer uses a post-decrement scheme during CALL.
- When the subroutine execution has finished and executes the RET instruction, the address of the instruction below the CALL is loaded into the PC and the instruction below the CALL instruction is executed.
- **Operation:** PC ←k Devices with 16 bits PC, 128K bytes program memory maximum.
- **Syntax:** CALL k where, 0 ≤ k < 64K
- **Cycles:** 4
- **Example:**

```
MOV R16,R0          ; Copy r0 to r16
CALL check          ; Call subroutine
nop                 ; Continue (do nothing)
...
check: CPI R16, $42 ; Check if r16 has a special value
BREQ error          ; Branch if equal
RET                 ; Return from subroutine
...
error: RJMP error   ; Infinite loop
```

RCALL (Relative Call to Subroutine)

- **Description:** It is the instruction which calls subroutine having address within PC - 2K +1 and PC + 2K (words). In the assembler, labels are used instead of relative operands.
- It is 2-byte instruction, in which the first 4-bits are opcode and the remaining 12-bits is the relative address of the target location of the subroutine to be called.
- The relative address range of 0×000 to 0×FFF is divided into forward and backward calls; **within the -2048 to +2047** words of memory relative to current PC value.
- **Operation:**

(i)PC ← PC + k + 1 Devices with 16 bits PC, 128K bytes program memory maximum.

- **Syntax:** RCALL k -2K ≤ k < 2K
- **Cycles:** 3

```
RCALL routine       ; Call subroutine
...
routine: PUSH R14   ; Save R14 on the stack
...
POP R14             ; Restore R14
RET                 ; Return from subroutine
```

ICALL (Indirect Call to Subroutine)

- **Description:** Indirect call of a subroutine pointed by the Z (16 bits) pointer register in the register file. The Z pointer register is 16 bits wide and allows call to a subroutine within the lowest 64K words (128K bytes) section in the program memory space.
- The stack pointer uses a post-decrement scheme during ICALL.
- **Operation:** PC(15:0) ← Z(15:0) Devices with 16 bits PC, 128K bytes program memory maximum.
- **Syntax:** ICALL ;STACK ← PC + 1; SP ← SP - 2 (2 bytes, 16 bits)
- **Cycles:** 3
- **Example:**

```
MOV R30,R0          ; Set offset to call table
ICALL               ; Call routine pointed to by R31:R30
```

RET (Return from Subroutine)

- **Description:** Returns from subroutine. The return address is loaded from the Stack. The stack pointer uses a pre-increment scheme during RET.
- **Operation:** PC(15:0) ← STACK ,Devices with 16 bits PC, 128K bytes program memory maximum.
- **Syntax:** RET ;SP←SP + 2, (2bytes,16 bits) **Cycles:** 4

```
RCALL routine       ; Call subroutine
...
routine: PUSH R14   ; Save R14 on the stack
...
POP R14             ; Restore R14
RET                 ; Return from subroutine
```

RETI (Return from Interrupt)

- **Description:** Returns from interrupt. The return address is loaded from the Stack and the global interrupt flag is set.
- Note that the ***status register is not automatically stored when entering an interrupt routine,*** and it is not restored when returning from an interrupt routine. This must be handled by the application program. The stack pointer uses a pre-increment scheme during RETI.
- **Operation:** PC(15:0) ← STACK, Devices with 16 bits PC, 128K bytes program memory maximum.
- **Syntax:** RETI ;SP ← SP + 2 (2 bytes, 16 bits)
- **Example:**

```
...
extint: PUSH R0     ; Save R0 on the stack
...
POP R0              ; Restore R0
RETI                ; Return and enable interrupts
```

5. Assembly language Programming with ATmega32 Instruction Set Con..

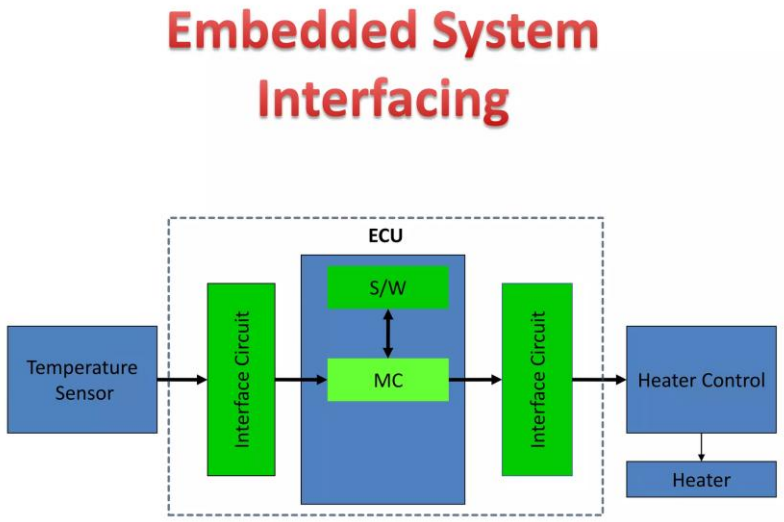
3. Data Transfer Instructions

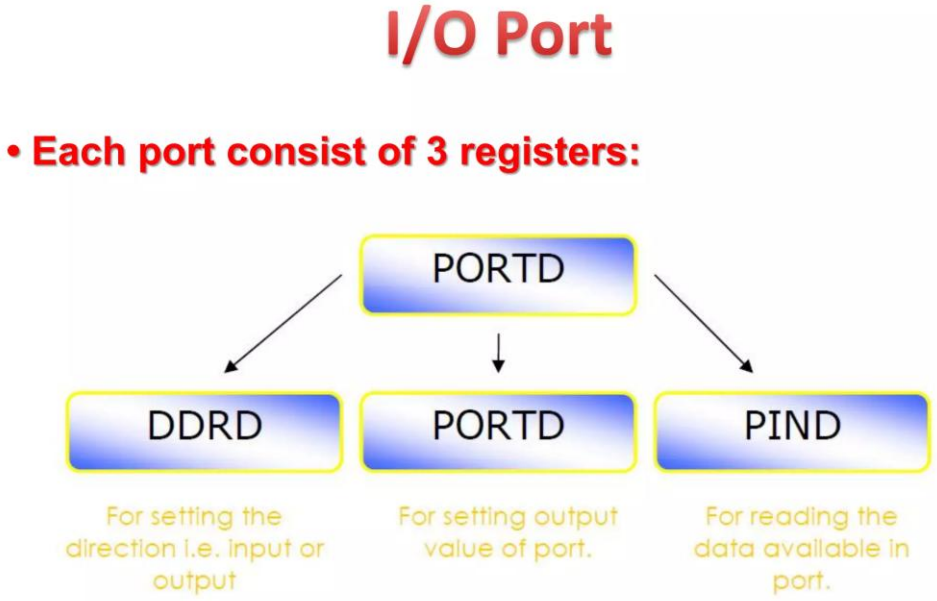
Instruction	Operand	Explanation	Example
MOV	D, S	D = S	MOV D, S
LDS	D, K(memory location)	D = Value at K	LDS D, K
LD	D, S	D = Value at memory location stored in S	LD D, S
LDI	D, K(constant)	D = K	LDI D, K
LPM	D, Z(flash memory)	Store the value in register Z from flash memory into the memory location stored in the D register	LPM D, Z
IN	D, A	Stores the value in register A in D. where A is from [0, 63](64 I/O Registers)	IN D, A
OUT	A, D	Stores the value in register D in A. where A is from [0, 63](64 I/O Registers)	OUT A, D
STS	K, S	Stores the value in register S into memory location K.	STS K, S
ST	D, S	Store the value in register S into the memory location stored in the D register	ST D, S
PUSH	D	Pushes the content of D on the top of the stack	PUSH D
POP	D	Removes the topmost entry from the stack and transfers that value to D	POP D

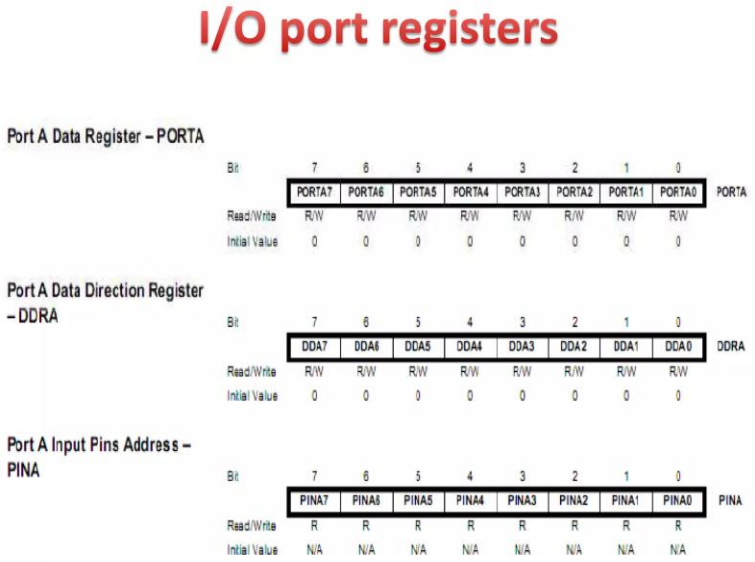
6. Programming in C to Interface peripherals, Interrupts, ISR and Timers

Interfacing Peripherals

- Interfacing handles how to connect with peripherals to another Embedded System. It includes how to integrate and communicate ES with each other. It also includes designing software handlers that perform the communication through the peripherals.
- In ATmega32 has 32 Programmable I/O lines – PortA, PortB, PortC and PortD.
- Each Port consists three registers :
  - Data Direction Register (DDRx)
  - Output Register (PORTx)
  - Input Register (PINx)







6. Programming in C to Interface peripherals, Interrupts, ISR and Timers Cont..

Interfacing Peripherals Cont..

Input/Output Programming

- Programmer decide which port is Input and which is output.
  - Configure the port direction by using register DDRx.
    1 for Output and 0 for Input.
  - In Read: Use register PINx.
  - In Write: Use register PORTx.

I/O port Programing 'cont

```
DDRA = 0xFF; //initialize portA as output
DDRB = 0x00; //initialize portB as input

if ((PINB & 0b00000001) == 1) //read a switch on PB0
{
      PORTA = 0xFF;       //All LEDs on
}
else
      PORTA = 0x00;       //All LEDs off
```

As Output:
– LEDs
– Buzzer
– Relay
… –

• As Input:
– Switches
– Digital sensors
– Signal from another uC
– PC parallel port

6. Programming in C to Interface peripherals, Interrupts, ISR and Timers Cont..

Interrupts

Types

1. Exception handling - Reset and Software Interrupts.
2. Non maskable Interrupts – It doesn't depend on global interrupt in processor status word. Usually it's external interrupts.
3. Maskable Interrupts – Depend on global interrupt enable in processor status. It may be
   - External interrupt from external pin.
   - Internal interrupt from peripheral.

What happens when an interrupt occurs?

- The current instruction finishes execution.
- The address of the next instruction is calculated and pushed on the stack.
- All the CPU registers are pushed onto the stack.
- The program counter is loaded with the address pointed to by the interrupt vector and execution continues.

Interrupt Service Routine (ISR)

- When an interrupt is invoked, the micro-controller runs the ISR.
- For every interrupt, there is a fixed location in memory that holds the address of its ISR.
- It is a specialized function or routine that is called when an interrupt is triggered by a hardware device.
- Example-

- Set global interrupt enable: (Allow)

```
sei();
```

- Clear global interrupt enable: (prevent)

```
cli();
```

- ISR

```
ISR(xxxxxxxxx)
{
      //code of int. handler
}
```

6. Programming in C to Interface peripherals, Interrupts, ISR and Timers Cont..

Timer Module

The timer of the AVR can be monitor Three events through State Register (TIMSK):

- **Timer Overflow** – means that the counter has counted up to its maximum value and is reset to zero in the next timer clock cycle. The resolution of the timer determines the maximum value of that timer. The timer overflow event causes the Timer Overflow Flag (TOVx) to be set in the Timer Interrupt Flag Register (TIFR).

- **Compare Match** – In case where it is not sufficient to monitor a timer overflow. The compare match interrupt can be used. The Output Compare Register (OCRx) can be loaded with a value [0.. MaxVal] which the timer will be checked against every timer cycle. When the timer reaches the compare value, the corresponding Output Compare Flag (OCFx) in the TIFR register is set. The Timer can be configured to clear the count register to “0” on a compare match.

- **Input capture** – The AVR has an input pin to trigger the input capture event. A signal change at this pin causes the timer value to be read and saved in the Input Capture Register (ICRx). At the same time the ICFx in the TIFR will be set. This is useful to measure the width of external pulses.

THANK YOU VERY MUCH

CHAPTER 2 COMPLETED