# Module 1
# AVR Microcontrollers & Assembly Language Programming

**syllabus**

Basics of Microcontrollers, Compare and contrast Microcontrollers and Microprocessors, Overview of AVR family of Microcontrollers, AVR features, AVR Architecture with block diagram, The General Purpose Registers in the AVR, AVR Data Memory, Using instructions with the Data Memory, AVR status Register, AVR data formats, Program counter and Program ROM space, RISC and Harvard architecture in AVR, Branch instructions and Looping, Unconditional branch instruction, Call instructions and stack, AVR Time delay and Instruction pipeline

## Basics of Microcontrollers

A microcontroller is a single chip microcomputer made through VLSI fabrication. A microcontroller is also called an embedded controller because the microcontroller and its support circuits are often built into, or embedded in, the devices they control. A microcontroller is available in different word lengths like microprocessors (4bit,8bit,16bit,32bit,64bit and 128-bit microcontrollers are available today).

1) A microcontroller basically contains one or more of the following components:

- Central processing unit(CPU)
- Random Access Memory)(RAM)
- Read Only Memory(ROM)
- Input/output ports
- Timers and Counters
- Interrupt Controls
- Analog to digital converters
- Digital analog converters
- Serial interfacing ports
- Oscillatory circuits

2) A microcontroller internally consists of all the features required for a computing system and functions as a computer without adding any external digital parts in it.

3) Most of the pins in the microcontroller chip can be made programmable by the user.

4) A microcontroller has many bit handling instructions that can be easily understood by the programmer. Overview of AVR family of Microcontrollers

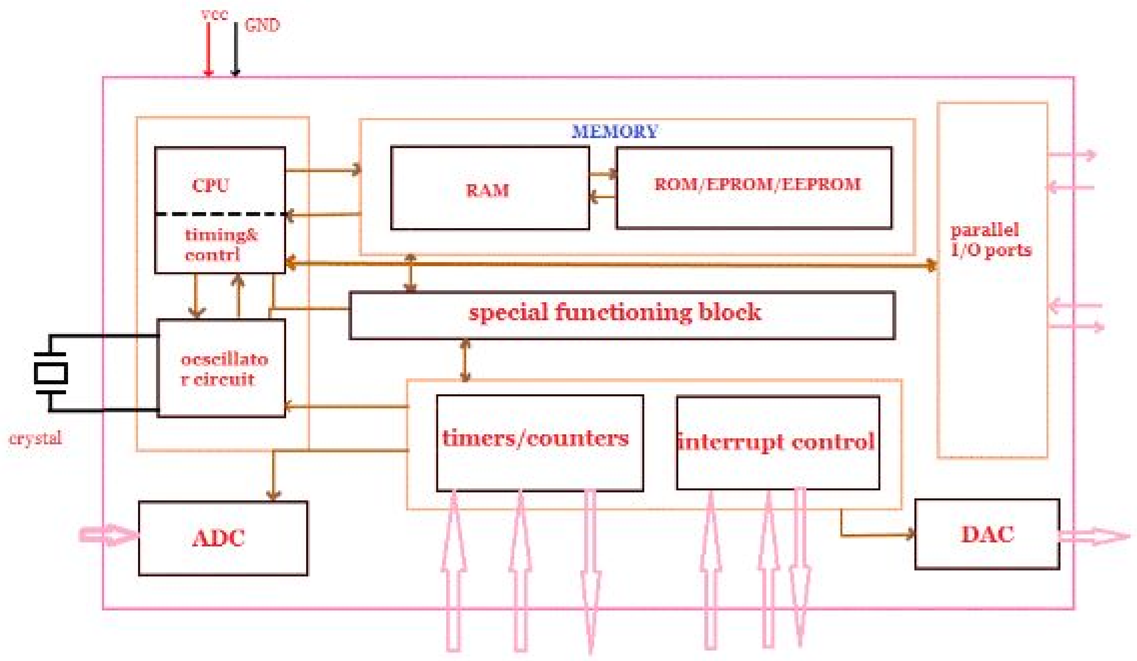
6) Higher speed and performance. Overview of AVR family of Microcontrollers

7) On-chip ROM structure in a microcontroller provides better firmware security.

8 ) Easy to design with low cost and small size.

Microcontroller structure

The basic structure and block diagram of a microcontroller is shown in the fig



- **CPU**

  CPU is the brain of a microcontroller. CPU is responsible for fetching the instruction, decodes it, then finally executed. CPU connects every part of a microcontroller into a single system. The primary function of CPU is fetching and decoding instructions. The instruction fetched from program memory must be decoded by the CPU

- **Memory**

  The function of memory in a microcontroller is the same as a microprocessor. It is used to store data and programs. A microcontroller usually has a certain amount of RAM and ROM (EEPROM, EPROM, etc) or flash memories for storing program source codes

- **Parallel input/output ports**

  Parallel input/output ports are mainly used to drive/interface various devices such as LCD'S, LED'S, printers, memories, etc to a microcontroller.

- **Serial ports**

Serial ports provide various serial interfaces between a microcontroller and other peripherals like parallel ports.

- **Timers/counters**

  This is one of the most useful functions of a microcontroller. A microcontroller may have more than one timer and counters. The timers and counters provide all timing and counting functions inside the microcontroller. The major operations of this section are performing clock functions, modulations, pulse generations, frequency measuring, making oscillations, etc. This also can be used for counting external pulses.

- **Analog to Digital Converter (ADC)**

  ADC converters are used for converting the analog signal to digital form. The input signal in this converter should be in analog form (e.g. sensor output) and the output from this unit is in digital form. The digital output can be used for various digital applications (e.g. measurement devices).

- **Digital to Analog Converter (DAC)**

  DAC performs reversal operation of ADC conversion.DAC converts the digital signal into analog format. It is usually used for controlling analog devices like DC motors, various drives, etc.

- **Interrupt control**

  The interrupt control used for providing interrupt (delay) for a working program. The interrupt may be external (activated by using interrupt pin) or internal (by using interrupt instruction during programming).

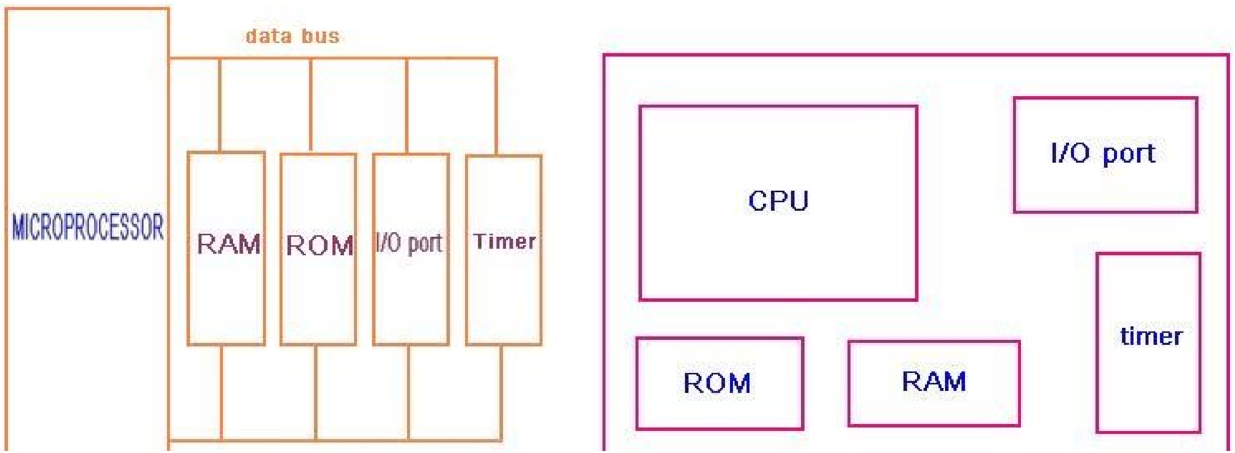
- **Special functioning block**

  Some microcontrollers used only for some special applications (e.g. space systems and robotics) these controllers containing additional ports to perform such special operations. This is considered a special functioning block.

**<u>Comparison between Microprocessor and Microcontroller</u>**

The main comparison between microprocessor and microcontroller shown in fig (1.2)

(



| Microprocessors | | Microcontrollers |
|---|---|---|
| 1 | It is only a general purpose computer CPU | It is a microcomputer itself |
| 2 | Memory, I/O ports, timers, interrupts are not available inside the chip | All are integrated inside the microcontroller chip |
| 3 | This must have many additional digital components to perform its operation | Can function as a microcomputer without any additional components. |
| 4 | Systems become bulkier and expensive. | Make the system simple, economic and compact |
| 5 | Not capable for handling Boolean functions | Handling Boolean functions |
| 6 | Higher accessing time required | Low accessing time |
| 7 | Very few pins are programmable | Most of the pins are programmable |
| 8 | Very few number of bit handling instructions | Many bit handling instructions |
| 9 | Widely Used in modern PC and laptops | widely in small control systems |
| E.g. | INTEL 8086,INTEL Pentium series | INTEL 8051,89960,PIC16F877 |

**Overview of AVR family of Microcontrollers**

AVR is a family of microcontrollers developed since 1996 by Atmel, acquired by microchip technology in 2016. These are modified Harvard architecture 8 bit RISC single-chip microcontrollers. AVR was one of the first microcontroller families to use on-chip flash memory for program storage.
AVR microcontrollers are obtainable in three categories

**Tiny AVR:** This microcontroller has Less memory, small in size, apt only for simpler applications.some characteristics are:

- program memory:1k to 8k bytes
- package: 8 to 28 pins
- limited peripheral set
- limited instruction set

**Mega AVR:** This microcontroller is the most popular one having a good amount of memory up to 256KB, higher no. of inbuilt peripherals and fit for modest to difficult applications.

some characteristics are:

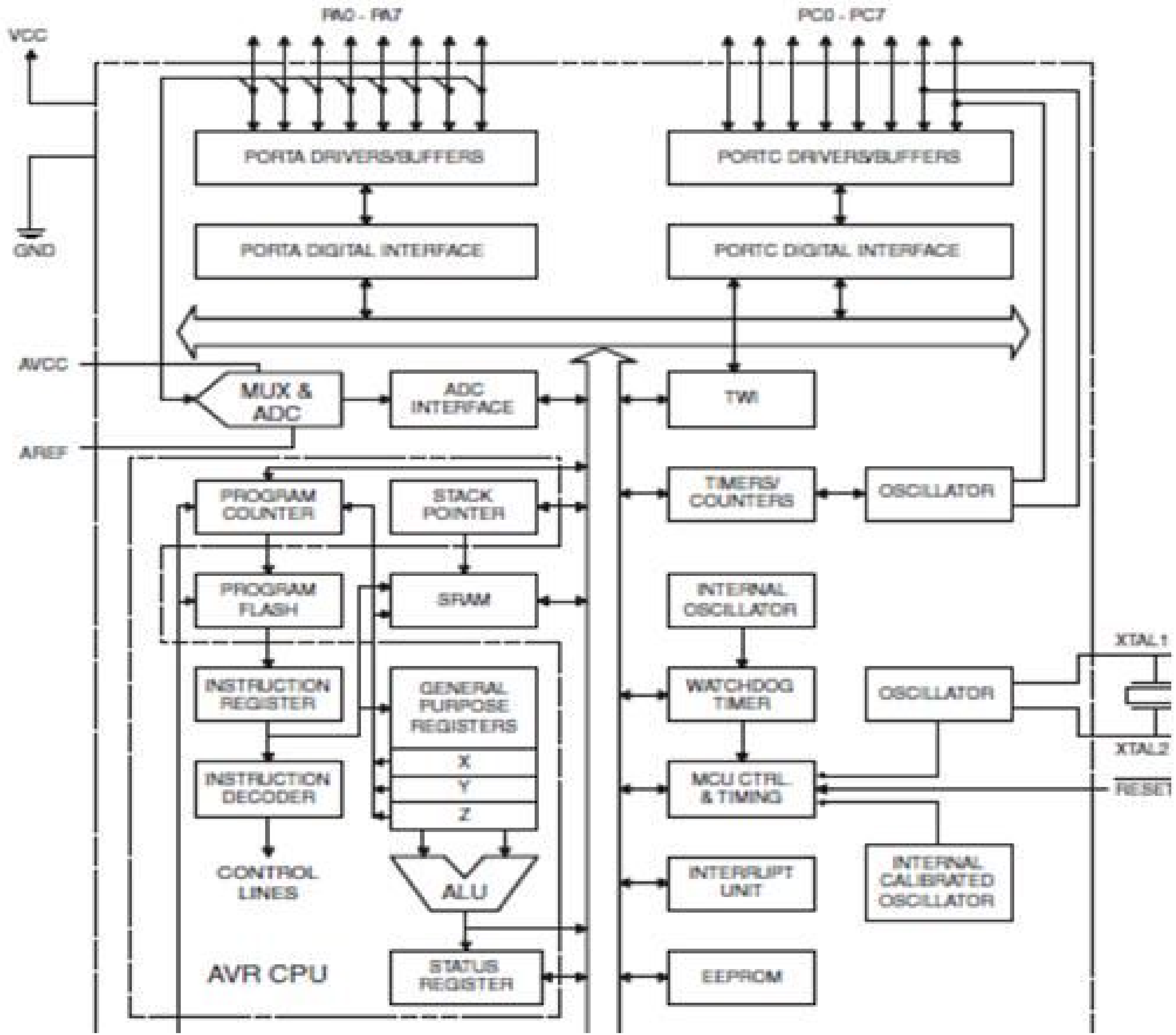
- package:28 to100 pins
- extensive peripheral set
- extended instruction set:they have rich instruction set

**Xmega AVR:** This microcontroller is used commercially for compound applications, which need large program memory and also high speed.

## AVR features

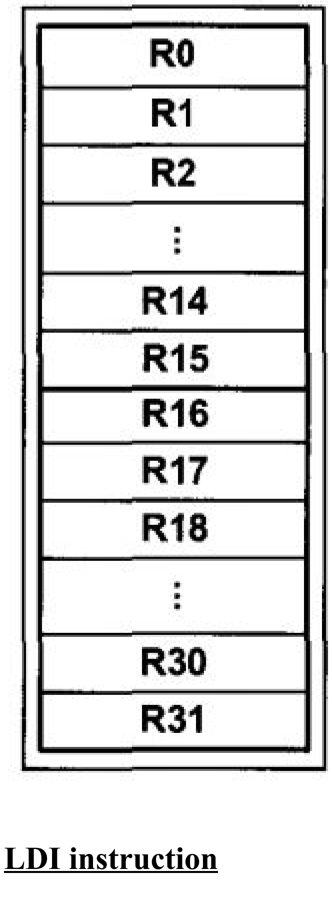
- 32 x 8 general working purpose registers.
- 32K bytes of in system self programmable flash program memory
- 2K bytes of internal SRAM
- 1024 bytes EEPROM
- Available in 40 pin DIP, 44 lead QTFP, 44-pad QFN/MLF
- 32 programmable I/O lines
- 8 Channel, 10 bit ADC
- Two 8-bit timers/counters with separate prescalers and compare modes
- One 16-bit timer/counter with separate prescaler, compare mode and capture mode.
- 4 PWM channels
- In system programming by on-chip boot program
- Programmable watchdog timer with separate on-chip oscillator.
- Programmable serial USART
- Master/slave SPI serial interface

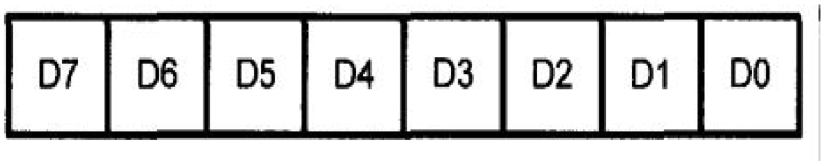
## AVR Architecture with block diagram



## The General Purpose Registers in the AVR

CPUs use many registers to store data temporarily. To program in Assembly language, we must understand the registers and architecture of a given CPU and the role they play in processing data. In this section we look at the general purpose registers (GPRs) of the AVR and we demonstrate the use of GPRs with simple instructions such as LDI and ADD. AVR microcontrollers have many registers for arithmetic and logic operations. In the CPU, registers are used to store information temporarily. That information could be a byte of data to be processed, or an address pointing to the data to be fetched. The vast majority of AVR registers are 8-bit registers. In the AVR there is only one data type: 8-bit. The 8 bits of a register are shown in the diagram below. These range from the MSB (most-significant bit) D7 to the LSB (least-significant bit) DO. With an 8-bit data type, any data larger than 8 bits must be broken into 8-bit chunks before it is processed. In AVR there are 32 general purpose registers. They are RO-R3 l and are located in the lowest location of memory address. See Figure 2-1. All of these registers are 8 bits. The general purpose registers in AVR are the same as the accumulator in other microprocessors. They can be used by all arithmetic and logic instructions. To understand the use of the general purpose registers, we will show it in the context of two simple instructions: LDI and ADD.





## 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 Rl6 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 Ox25 (25 in hex).

*LDI R20,0x25 ;load R20 with Ox25 (R20 - Ox25)*

When programming the GPRs of the AVR microcontroller with an immediate value, the following points should be noted:

I. If we want to present a number in hex, we put a dollar sign ($) or a 0x in front of it. If we put nothing in front of a number, it is in decimal. For example, in "LDI R16, 50", Rl6 is loaded with 50 in decimal, whereas in *"LDI Rl 6, 0x5 0 "*, RI 6 is loaded with 50 in hex.

2. If values 0 to F are moved into an 8-bit register such as GPRs, the rest of the bits are assumed to be all zeros. For example, in *"LDI R16, 0x5"* the result will be Rl6 = 0x05; that is, Rl6 = 00000101 in binary.

3. Moving a value larger than 255 (FF in hex) into the GPRs will cause an error.

*LDI Rl7, 0x7F2 ;ILLEGAL $7F2 > 8 bits ($FF)*

**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 ofRr 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:

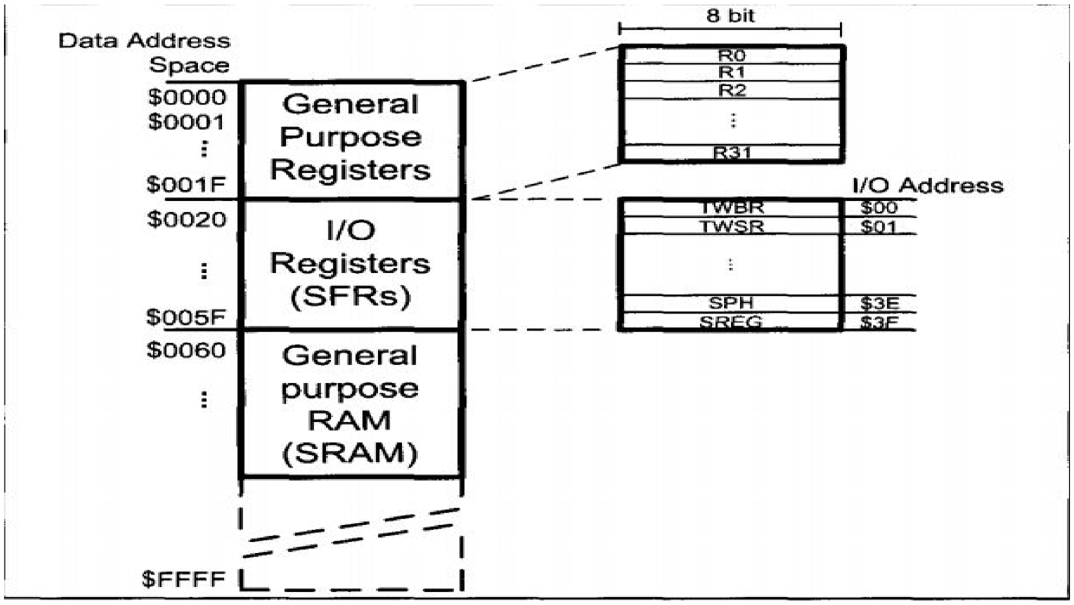
*LDI Rl6,0x25 ;load 0x25 into Rl6*

*LDI Rl 7, 0x34 ;load 0x34 into Rl7*

*ADD Rl6, Rl 7 ;add value Rl7 to Rl6 (Rl6 ~ Rl6 + Rl7), RI 6 = 0x59 (0x25 + 0x34 = 0x59)*

## AVR Data Memory

In AVR microcontrollers there are two kinds of memory space: code memory space and data memory space. Our program is stored in code memory space,whereas the data memory stores data.The data memory is composed of three parts: GPRs (general purpose registers), I/O memory, and internal data SRAM.



**GPRs (general purpose registers)**

The GPRs use 32 bytes of data memo-ry space. They always take the address location $00-$1 F in the data memory space, regardless of the AVR chip number.(details is mentioned above)

**1/0 memory (SFRs)**

The I/O memory is dedicated to specific functions such as status register, timers, serial communication, I/O ports, ADC, and so on. The function of each I/O memory location is fixed by the CPU designer at the time of design because it is used for control of the microcontroller or peripherals. The AVR I/O memory is made of 8-bit registers. The number of locations in the data memory set aside for 1/0 memory depends on the pin numbers and peripheral functions supported by that chip, although the number can vary from chip to chip even among members of the same family. However, all of the AYRs have at least 64 bytes of I/O memory locations. This 64-byte section is called standard 110 memory. In AVRs with more than 32 I/O pins (e.g., ATmega64, ATmegal28, and ATmega256) there is also an extended I/O memory, which contains the registers for controlling the extra ports and the extra peripherals. See Figures 2-3 and 2-4. In other microcontrollers the 1/0 registers are called SFRs (special function registers) since each one is dedicated to a specific function. In contrast to SFRs, the GPRs do not have any specific function and are used for storing general data.

**Internal data SRAM**

Internal data SRAM is widely used for storing data and parameters by AYR programmers and C compilers. Generally, this is called a scratch pad. Each location of the SRAM can be accessed directly by its address.

**Table 2-1: Data Memory Size for AVR Chips**

| | Data Memory (Bytes) | = | I/O Registers (Bytes) | + | SRAM (Bytes) | + | General Purpose Register |
|---|---|---|---|---|---|---|---|
| ATtiny25 | 224 | | 64 | | 128 | | 32 |
| ATtiny85 | 608 | | 64 | | 512 | | 32 |
| ATmega8 | 1120 | | 64 | | 1024 | | 32 |
| ATmega16 | 1120 | | 64 | | 1024 | | 32 |
| ATmega32 | 2144 | | 64 | | 2048 | | 32 |
| ATmega128 | 4352 | | 64+160 | | 4096 | | 32 |
| ATmega2560 | 8704 | | 64+416 | | 8192 | | 32 |

## <u>Using instructions with the Data Memory</u>

The instructions we have used so far worked with the immediate (constant) value of K and the GPRs. They also used the GPRs as their destination. We saw simple examples of using LDI and ADD earlier. The AYR allows direct access to other locations in the data memory. In this section we show the instructions accessing various locations of the data memory. This is one of the most important sections in the book for mastering the topic of AYR Assembly language programming.

<u>LDS instruction (LoaD direct from data Space)</u>

*LDS Rd, K ; load Rd with the contents of location K (0 -S: 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 1/0 registers, a location in the internal SRAM, or a GPR. For example, the *"LDS R20, 0x1"* instruction will copy the contents of location I (in hex) into R20.

The following instruction loads R5 with the contents of location 0x200.

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

The following program adds the contents of location Ox300 to location 0x302. To do so, first it loads RO with the contents of location Ox300 and RI with the contents of location Ox302, then adds RO to RI:

*LDS RO, Ox300 ;RO = the contents of location Ox300*

*LDS Rl, Ox302 ;Rl = the contents of location Ox302*

*ADD Rl, RO ; add RO to Rl*

<u>STS instruction (STore direct to data Space)</u>

*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 1/0 registers, a location in the SRAM, or a GPR. For example, the *"STS 0xl, Rl0"* instruction will copy the contents of RI 0 into location I.

IN instruction (IN from 1/0 location)

*IN Rd, A ; load an I/O location to the GPR (0 s: d :<;;31), (0 :5: A s; 63)*
The IN instruction tells the CPU to load one byte from an I/0 register to the GPR. After this instruction is executed, the GPR will have the same value as the I/O register. For example, the "IN R20, Oxl6" instruction will copy the contents of location 16 (in hex) of the I/O memory into R20.

OUT instruction (OUT to 1/0 location)

*OUT A, Rr ; store register to I/O location (0 S r s 31), (0 SA s 63)*
The OUT instruction tells the CPU to store the GPR to the I/O register. After the instruction is executed, the I/O register will have the same value as the GPR. For example, the "OUT PORTD,RlO" instruction will copy the contents of RIO into PORTD (location 12 of the I/O memory).

MOV instruction

The MOV instruction is used to copy data among the GPR registers of RO-R3 l. 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 RI 0:
*MOV Rl0,R20 ;RlO = R20*
For instance, if R20 contains 60, after execution of the above instruction both R20 and RIO will contain 60.

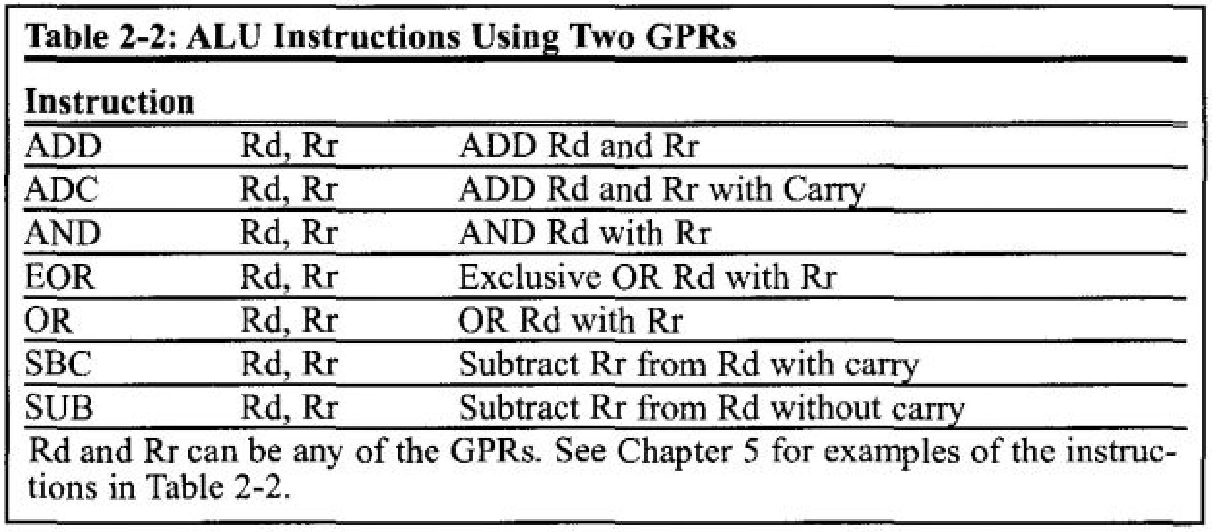


**Table 2-2: ALU Instructions Using Two GPRs**

| 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.

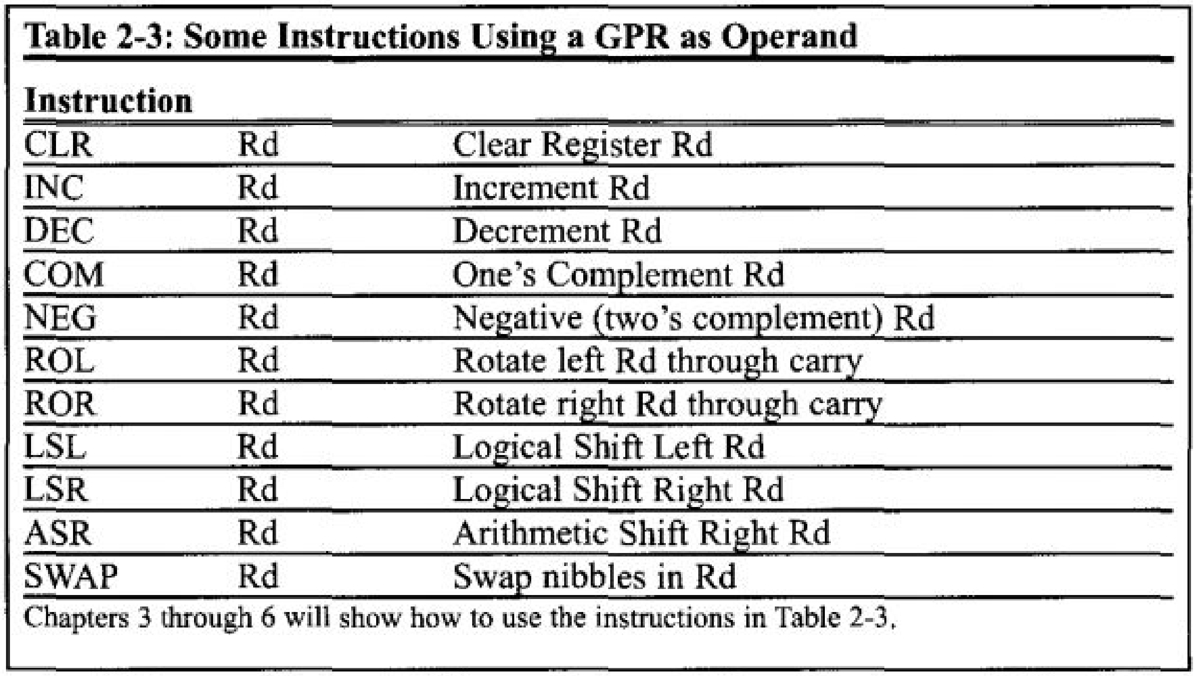


**Table 2-3: Some Instructions Using a GPR as Operand**

| 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.

## AVR status Register

The status register is an 8-bit register. It is also referred to as the flag register. See Figure 2-8 for the bits of the status register. The bits C, Z, N, Y, S, and H are called conditional flags, meaning that they indicate some conditions that result after an instruction is executed.

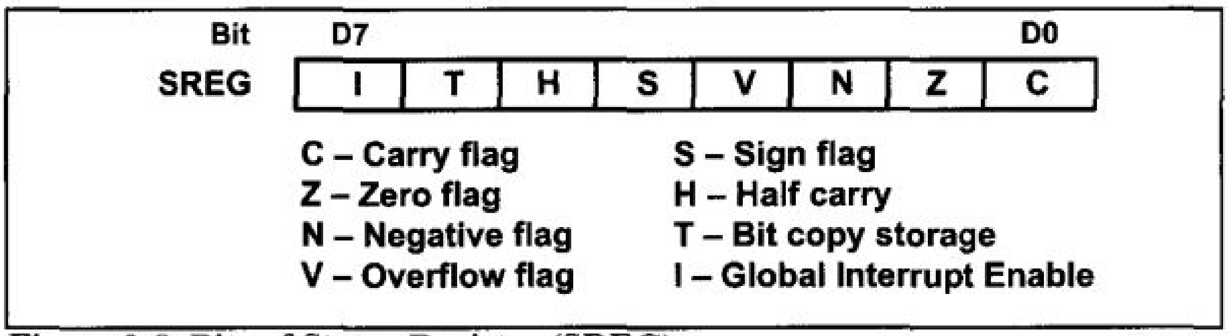


**Figure 2-8. Bits of Status Register (SREG)**

C, the carry flag

This flag is set whenever there is a carry out from the D7 bit. This flag bit is affected after an 8-bit addition or subtraction.

Z, the zero flag

The zero flag reflects the result of an arithmetic or logic operation. If the result is zero, then Z = 1. Therefore, Z = 0 if the result is not zero.

N, the negative flag

Binary representation of signed numbers uses D7 as the sign bit. The negative flag reflects the result of an arithmetic operation. If the D7 bit of the result is zero, then N = 0 and the result is positive. If the D7 bit is one, then N = 1 and the result is negative.

<u>V, the overflow flag</u>

This flag is set whenever the result of a signed number operation is too large, causing the high-order bit to overflow into the sign bit. In general, the carry flag is used to detect errors in unsigned arithmetic operations while the overflow flag is used to detect errors in signed arithmetic operations.

<u>S, the Sign bit</u>

This flag is the result of Exclusive-ORing (XOR) of N and V flags.

<u>H, Half carry flag</u>

If there is a carry from D3 to D4 during an ADD or SUB operation, this bit is set; otherwise, it is cleared. This flag bit is used by instructions that perform BCD (binary coded decimal) arithmetic. In some microprocessors this is called the AC flag (Auxiliary Carry flag).

<u>Global Interrupt Enable</u>

Used to enable and disable interrupts. – 1: enabled. 0: disabled.

## AVR data formats

<u>AVR data type</u>

The AVR microcontroller has only one data type. It is 8 bits, and the size of each register is also 8 bits. It is the job of the programmer to break down data larger than 8 bits (00 to OxFF, or 0 to 255 in decimal) to be processed by the CPU. The data types used by the AVR can be positive or negative.

<u>Data format representation</u>

There are four ways to represent a byte of data in the AVR assembler. The numbers can be in hex, binary, decimal, or ASCII formats. The following are examples of how each works.

<u>Hex numbers</u>

There are two ways to show hex numbers:

1. Put Ox (or OX) in front of the number like this: LDI RI 6, ox99
2. Put S in front of the number, like this: R22, $99.

<u>Binary numbers</u>

There is only one way to represent binary numbers in an AVR assembler.
It is as follows:

*LDI R16,0b10011001;; R16= 10011001 or 99 in hex*

The uppercase B will also work. Here are some examples of how to use it:

*LDI R23, 0b00100101 ;;;R23=$25*
*SUBI R25, 0B00010001;;R23=$25-$11=$14*

Decimal numbers

To indicate decimal numbers in an AVR assembler we simply use the decimal and nothing before or after it. Here are some examples of how to
use it:

*LDI R17, 12 ;;00010010 in binary and 0C in hex*
*SUBI R17, 2 ;;R17= 12-2=10 where 10 is equal to OxOA*

ASCII Characters

To represent ASCII data in an AVR assembler we use single quotes as follows:
*LDI R23,'2' ;;R23 - 00110010 or 32 in hex.*
This is the same as other assemblers such as the 8051 and x86. Here are

## **Assembler directives**

While instructions tell the CPU what to do, directives (also called pseudo insirucrions) give directions to the assembler. For example, the LDI and ADD instructions are commands to the CPU but .EQU, .DEVlCE and .ORG) are directives to the assembler. The following sections present some more widely used directives of the AVR and how they are used. The directives help us develop our program easier and make our program legible (more readable).

.EQU (equate)

This is used to define a constant value or a fixed address. The .EQU directive does not set aside storage for a data item 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. The following uses .EQU for the counter constant. and then the constant is used to load the R21 register:

.SET

This directive is used to define a constant value or a fixed address. in this regard. the .SET and .EQU directives are identical. The only difference is that the value assigned by the .SET directive may be reassigned later.

.ORG (origin)

The .ORG directive is used to indicate the beginning of the address. It can be used for both code and data.

.INCLUDE directive

The .include directive tells the AVR assembler to add the contents of a file to our program (like the #include directive in C language).

## **Program counter and Program ROM space**

Program counter in the AVR

The most important register in the AYR microcontroller is the PC (program counter). The program counter is used by the CPU to point to the address of the next instruction to be executed. As the CPU fetches the opcode from the program ROM, the program counter is incremented automatically to point to the next instruction. In the case of the AVR microcontrollers (that is, all members regardless of the family and variation), the microcontroller wakes up at memory address 0000 when it is powered up.

Placing code in program ROM

To get a better understanding of the role of the program counter in fetching and executing a program, we examine the action of the program counter as each instruction is fetched and executed. First, we examine once more the list file of the sample program and show how the code is placed into the Flash ROM of the AVR chip. As we can see, the opcode and operand for each instruction are listed on the left side of the list file. After the program is burned into ROM of an AYR family member such as ATmega32 or ATtiny 11, the opcode and operand are placed in ROM memory loca-
tions starting at 0000

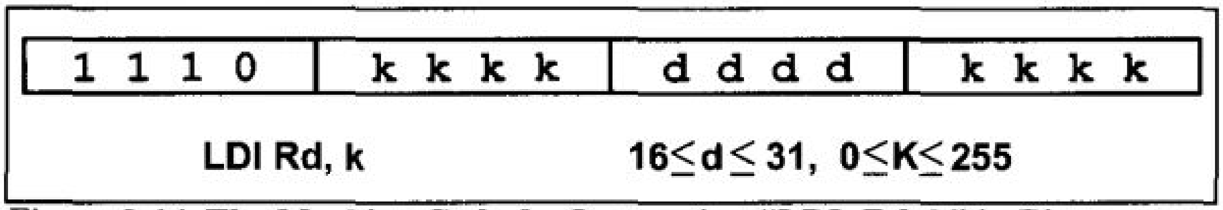


**Figure 2-14. The Machine Code for Instruction "LDI Rd, k" in Binary**

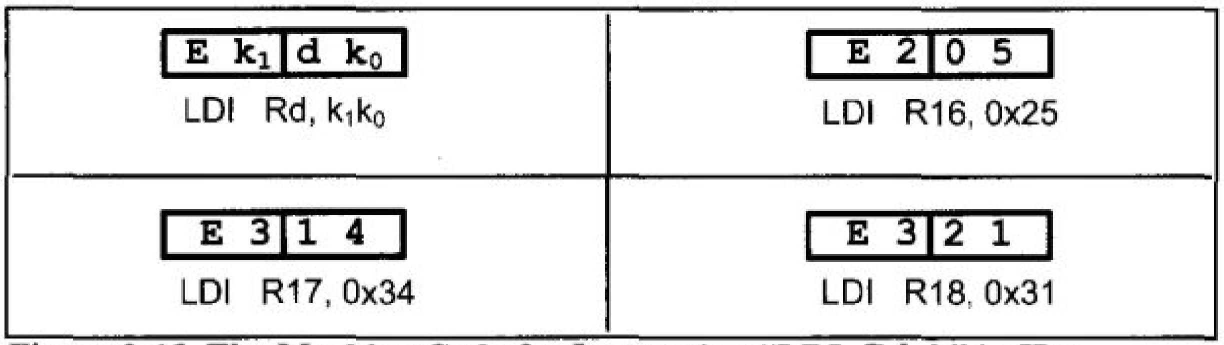
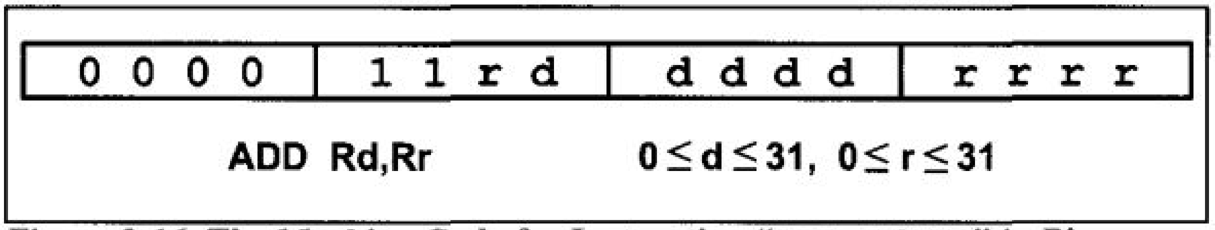


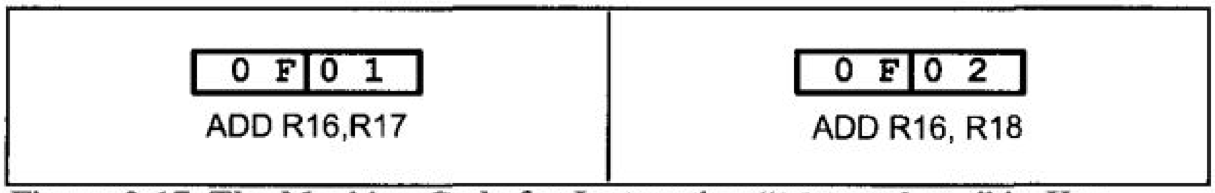
**Figure 2-15. The Machine Code for Instruction "LDI Rd, k" in Hex**



| | | | |
|---|---|---|---|
| 0 0 0 0 | 1 1 r d | d d d d | r r r r |

ADD Rd,Rr $0 \leq d \leq 31$, $0 \leq r \leq 31$

**Figure 2-16. The Machine Code for Instruction "ADD Rd,Rr" in Binary**

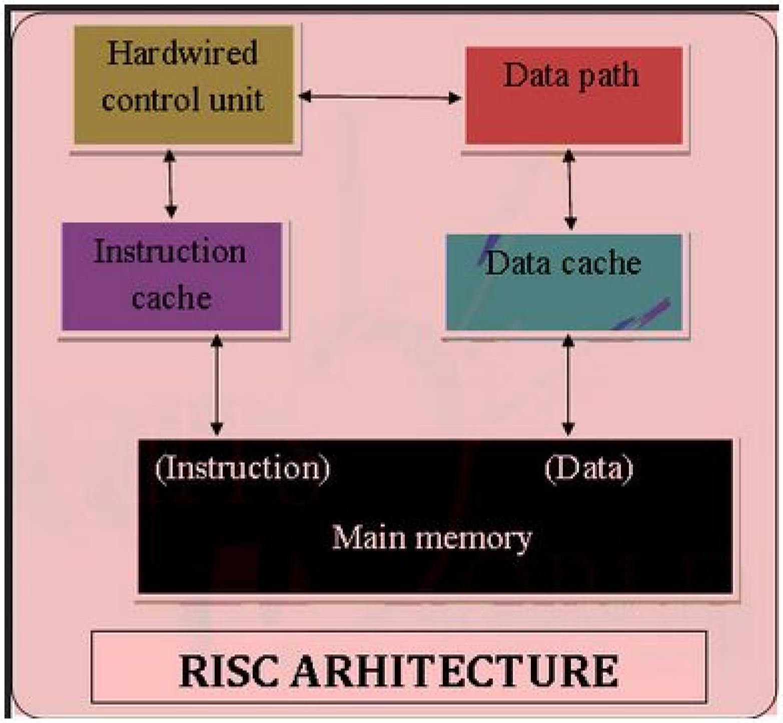


| | |
|---|---|
| 0 F 0 1 | 0 F 0 2 |
| ADD R16,R17 | ADD R16, R18 |

**Figure 2-17. The Machine Code for Instruction "ADD Rd,Rr" in Hex**

## <u>RISC and Harvard architecture in AVR</u>

RISC (Reduced Instruction Set Computer) is used in portable devices due to its power efficiency. For Example, Apple iPod and Nintendo DS. RISC is a type of microprocessor architecture that uses a highly-optimized set of instructions. RISC does the opposite, reducing the cycles per instruction at the cost of the number of instructions per program. Pipelining is one of the unique features of RISC. It is performed by overlapping the execution of several instructions in a pipeline fashion. It has a high performance advantage over CISC.

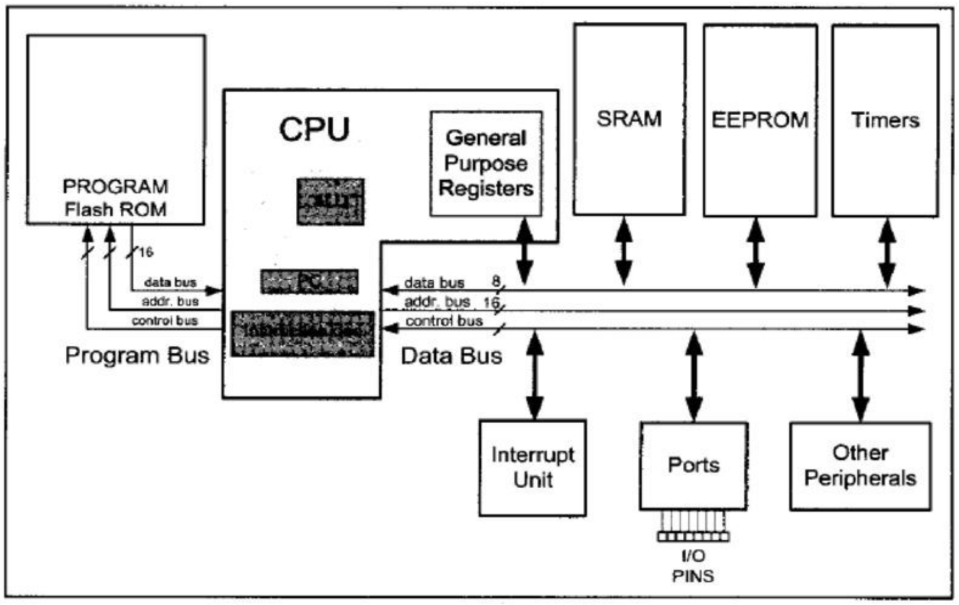


RISC Architecture

Features of RISC

- RISC processors have a fixed instruction size.
- One of the major characteristics of RISC architecture is a large number of registers. All RISC architectures have at least 32 registers. Of these 32 registers, only a few are assigned to a dedicated function. One advantage of a large number of registers is that it avoids the need for a large stack to store parameters. Although a stack can be implemented on a RISC processor, it is not as essential as in CISC because so many registers are available. In the AVR microcontrollers the use of 32 general purpose registers satisfies this RISC feature.
- RISC processors have a small instruction set. RISC processors have only basic instructions such as ADD, SUB, MUL, LOAD, STORE, AND, OR, EOR,CALL, JUMP, and so on.
- The most important characteristic of the RISC processor is that more than 95% of instructions are executed with only one clock cycle, in contrast to CISC instructions. Even some of the 5% of the RISC instructions that are executed with two clock cycles can be executed with one clock cycle by juggling instructions around (code scheduling).
- RISC processors have separate buses for data and code.
- RISC instructions, however, due to the small set of instructions, are implemented using the hardwire method.Hardwiring of RISC instructions takes no more than 10% of the transistors.
- RISC uses load/store architecture.

## Harvard architecture in the AVR



The Harvard architecture is a term for a computer system that contains two separate areas for commands or instructions and data. In the Harvard architecture, the media, format and nature of the two different parts of the system may be different, as the two systems are represented by two separate structures.

**Branch instructions and Looping**

Loops

Repeating a sequence of instructions or an operation a certain number of times is called a loop. The loop is one of most widely used programming techniques. In the AVR, there are several ways to repeat an operation many times. we use a loop inside a loop, which is called a nested loop.

Using BRNE instruction for looping

The BRNE (branch if not ~equal!) instruction uses the zero flag in the status register. The BRNE instruction is used as follows:

```
BACK: ......... ;start of the loop
. . . . . . . . . ; body of the loop
. . . . . . . . . ;body of the loop
DEC Rn ;decrement Rn, Z = 1 if Rn 0
```

BRNE BACK ;branch to BACK if Z ~ 0

BREQ (branch if equal, branch if Z = 1)

In this instruction, the Z flag is checked. If it is high, the CPU jumps to the target address. For example, look at the following code.
IN R20, PINB ;read PINE and put it in R20
TST R2 0 ;set the flags according to R20
BREQ OVER ;jump if R20 is zero

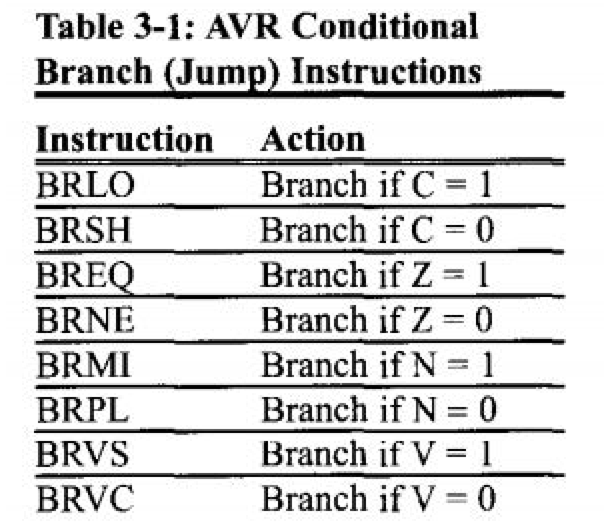


Table 3-1: AVR Conditional Branch (Jump) Instructions

| Instruction | Action |
|---|---|
| BRLO | Branch if C = 1 |
| BRSH | Branch if C = 0 |
| BREQ | Branch if Z = 1 |
| BRNE | Branch if Z = 0 |
| BRMI | Branch if N = 1 |
| BRPL | Branch if N = 0 |
| BRVS | Branch if V = 1 |
| BRVC | Branch if V = 0 |

**Unconditional branch instruction**

The unconditional branch is a jump in which control is transferred unconditionally to the target location. In the AYR there are three unconditional branches: JMP Gump), RJMP (relative jump), and IJMP (indirect jump).
JMP (JMP is a long jump)
JMP is an unconditional jump that can go to any memory location in the 4M (word) address space of the AYR. It is a 4-byte (32-bit) instruction in which I 0 bits are used for the opcode, and the other 22 bits represent the 22-bit address of the target location.
RJMP (relative jump)
In this 2-byte (16-bit) instruction, the first 4 bits are the opcode and the rest (lower 12 bits) is the relative address of the target location. The relative address range of 000 - $FFF is divided into forward and backward jumps; that is, within -2048 to +2047 words of memory relative to the address of the current PC (program counter). If the jump is forward, then the relative address is positive. If the jump is backward, then the relative address is negative.
IJMP (indirect jump)

IJMP is a 2-byte instruction. When the instruction executes, the PC is loaded with the contents of the Z register, so it jumps to the address pointed to by the Z register.

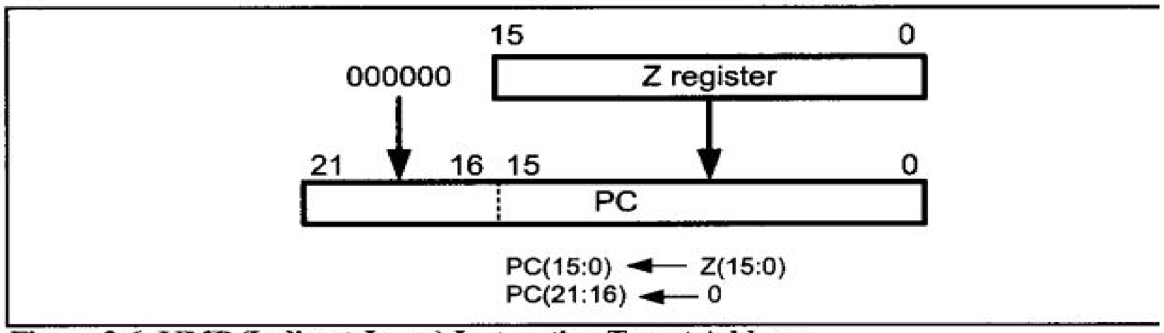


**Figure 3-6. IJMP (Indirect Jump) Instruction Target Address**

## Call instructions and stack

Another control transfer instruction is the CALL instruction, which is used to call a subroutine. Subroutines are often used to perform tasks that need to be performed frequently. This makes a program more structured in addition to saving memory space. In the AYR there are four instructions for the call subroutine: CALL (long call) RCALL (relative call), ICALL (indirect call to Z), and EICALL (~xtended indirect call to Z).

CALL

In this 4-byte (32-bit) instruction, I 0 bits are used for the opcode and the other 22 bits, k21-k0, are used for the address of the target subroutine, just as in the JMP instruction. To make sure that the AVR knows where to come back to after execution of the called subroutine, the microcontroller automatically saves on the stack the address of the instruction immediately below the CALL. When a subroutine is called, control is transferred to that subroutine, and the processor saves the PC (program counter) of the next instruction on the stack and begins to fetch instructions from the new location. After finishing execution of the subroutine, the RET instruction transfers control back to the caller. Every subroutine needs RET as the last instruction.

RCALL (relative call)

RCALL is a 2-byte instruction in contrast to CALL, which is 4 bytes. Because RCALL is a 2-byte instruction, and only 12 bits of the 2 bytes are used for the address, the target address of the subroutine must be within -2048 to +2047 words of memory relative to the address of the current PC.

ICALL (indirect call)

In this 2-byte (16-bit) instruction, the Z register specifies the target address. When the instruction is executed, the address of the next instruction is pushed into the stack (like CALL and RCALL) and the program counter is loaded with the contents of the Z register. So, the Z register should contain the address of a function when the ICALL instruction is executed. Because the Z register is 16 bits wide, the ICALL instruction can call the subroutines that are within the lowest 64K words of the program memory. (The target address calculation in ICALL is the same as for the UMP instruction.)

**Stack and stack pointer in AVR**

The stack is a section of RAM used by the CPU to store information temporarily. This information could be data or an address. The CPU needs this storage area because there are only a limited number of registers.

How stacks are accessed in the AVR

If the stack is a section of RAM, there must be a register inside the CPU to point to it. The register used to access the stack is called the SP (stack pointer) register. In I/O memory space, there are two registers named SPL (the low byte of the SP) and SPH (the high byte of the SP). The SP is implemented as two registers. The SPH register presents the high byte of the SP while the SPL register presents the lower byte.

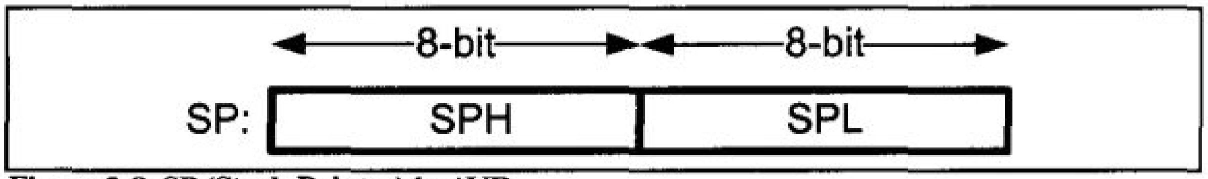


**Figure 3-8. SP (Stack Pointer) in AVR**

**AVR TIME DELAY AND INSTRUCTION PIPELINE**

Delay calculation for the AVR

In creating a time delay using Assembly language instructions, one must be mindful of two factors that can affect the accuracy of the delay:

1. The crystal frequency: The frequency of the crystal oscillator connected to the XTAL 1 and XTAL2 input pins is one factor in the time delay calculation. The duration of the clock period for the instruction cycle is a function of this crystal frequency.

2. The AVR design: Since the 1970s, both the field of IC technology and the architectural design of microprocessors have seen great advancements. Due to the limitations of IC technology and limited CPU design experience for many years, the instruction cycle duration was longer. Advances in both IC technology and CPU design in the 1980s and 1990s have made the single instruction cycle a common feature of many microcontrollers. Indeed, one way to increase performance without losing code compatibility with the older generation of a given family is to reduce the number of instruction cycles it takes to execute an instruction. One might wonder how microprocessors such as AVR are able to execute an instruction in one cycle. There are three

ways to do that: (a) Use Harvard architecture to get the maximum amount of code and data into the CPU, (b) use RISC architecture features such as fixed-size instructions, and finally ( c) use pipelining to overlap fetching and execution of instructions.

Pipelining

In early microprocessors such as the 8085, the CPU could either fetch or execute at a given time. In other words, the CPU had to fetch an instruction from memory, then execute it; and then fetch the next instruction, execute it, and so on. The idea of pipelining in its simplest form is to allow the CPU to fetch and execute at the same time, as shown in Figure 3-12. (An instruction fetches while the previous instruction executes.)

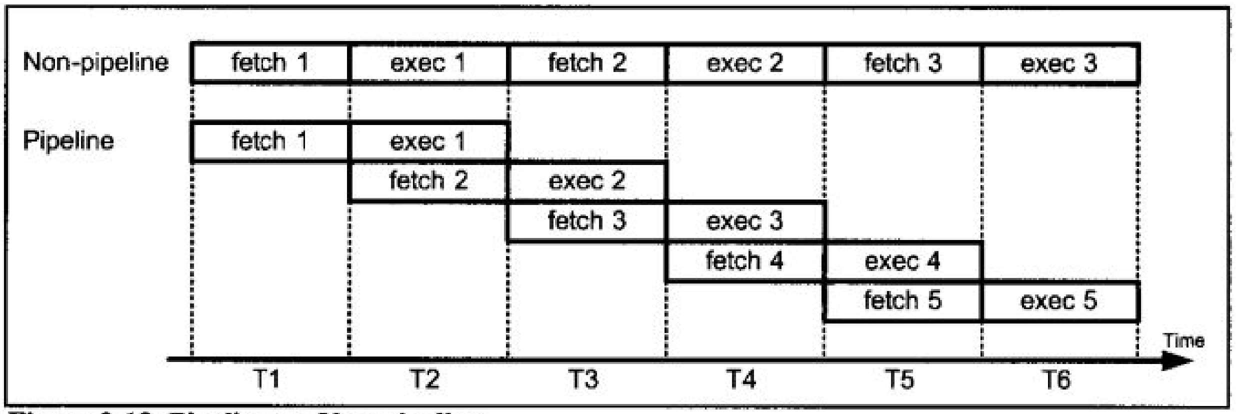


**Figure 3-12. Pipeline vs. Non-pipeline**

We can use a pipeline to speed up execution of instructions. In pipelining, the process of executing instructions is split into small steps that are all executed in parallel. In this way, the execution of many instructions is overlapped. One limitation of pipelining is that the speed of execution is limited to the slowest stage of the pipeline. Compare this to making pizza. You can split the process of making pizza into many stages, such as flattening the dough, putting on the toppings, and baking, but the process is limited to the slowest stage, baking, no matter how fast the rest of the stages are performed. What happens if we use two or three ovens for baking pizzas to speed up the process? This may work for making pizza but not for executing programs, because in the execution of instructions we must make sure that the sequence of instructions is kept intact and that there is no out-of-step execution.

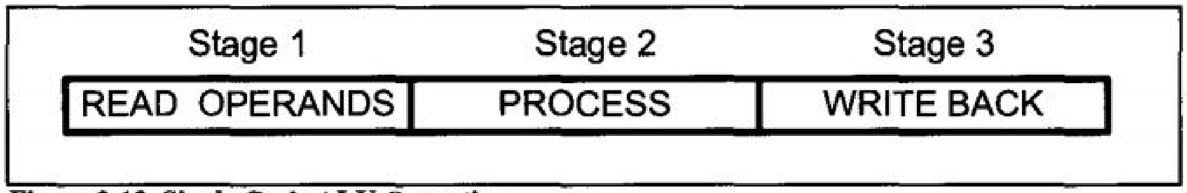


**Figure 3-13. Single Cycle ALU Operation**

AVR multistage execution pipeline

As shown in Figure 3-13, in the AVR, each instruction is executed in 3 stages: operand fetch, ALU operation execution, and result write back. In step 1, the operand is fetched. In step 2, the operation is performed; for example, the adding of the two numbers is done. In step 3, the result is written into
the destination register.

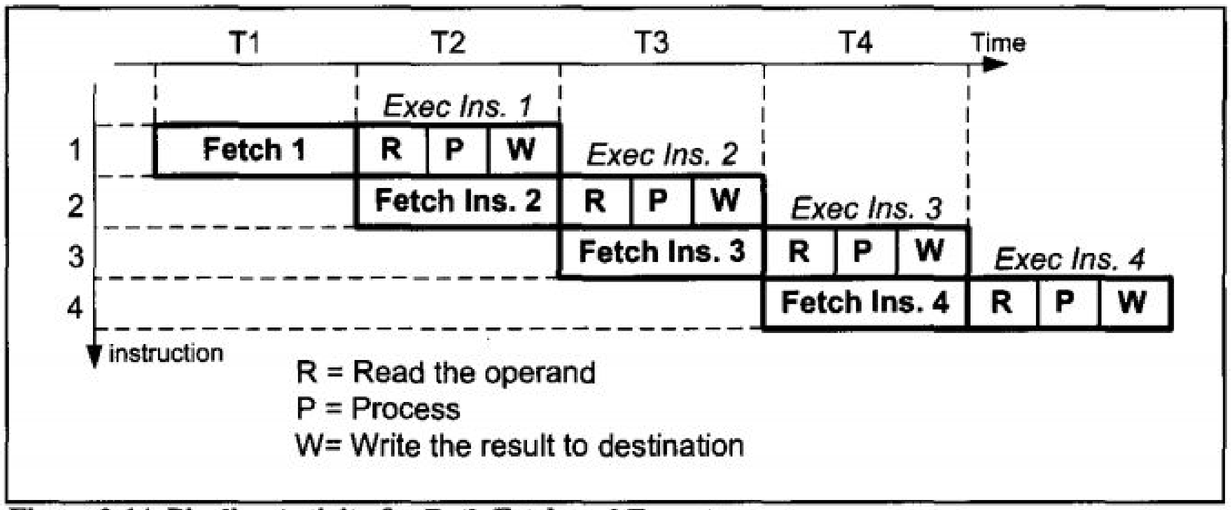


**Figure 3-14. Pipeline Activity for Both Fetch and Execute**