MODULE1 – 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 generalPurpose 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

Microcontroller

- A microcontroller has a CPU (microprocessor) with fixed amount of RAM, ROM, I/O ports and timer on a single chip.
- All the required hardware for a system is combined together on a single chip.
- In other words, RAM, ROM, I/O ports and timer are embedded on a single chip.
- Thus, no need to add external memory, I/O ports and timer to it.
- Being a built- in chip it occupies less space, draw less current and cheap. So it is ideal for many applications.
- Applications: TV Remote control, modem, printer, keyboard, cable TV terminal etc.
- Popular microcontrollers are AVR, 8051, PIC and Z8

CPU	RAM	ROM
I/O ... ADC	Timer	Serial com port

Fig 1.1 Diagram of microcontroller

Microprocessor

- Microprocessor contains no ROM, no RAM, no I/O ports and no timer on the chip itself.
- Microprocessor must add RAM, ROM, I/O ports and timer externally to make them functional.
- Although the addition of external RAM, ROM, I/O ports, and timer makes them bulkier and much more expensive.
- Example: Intel's X86 family (8086, 80286, 80386, Pentium), Motorola

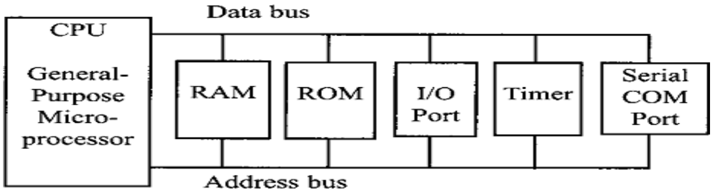


Fig 1.2 Diagram of General purpose microprocessor system

Comparison and Contrast Between Microprocessor and Microcontroller

Microcontroller	**Microprocessor**
It contains fixed amount of RAM, ROM and I/O ports and timer all on a single chip.	It contains no ROM, no RAM, no timer and no I/O ports on the chip itself.
Occupies less space, draw less power and cheap.	Needs more space consume more power less reliable and costly.
Microcontrollers have memory oriented architecture.	Most of the microprocessors have register oriented architecture.
Microcontrollers based systems are easy to troubleshoot and maintain.	Microprocessor based systems are not easy to troubleshoot and maintain.

Designer cannot add any external memory, I/O ports or timer. Eg: Z8,PIC,AVR	Designer decides the amount of RAM, ROM and I/O ports needed to fit the task. Eg: 8085,8086

Overview of AVR Microcontroller

- The basic architecture of AVR was designed by two students of Norwegian Institute of Technology (NTH) Alf-Egil Bogen and Vegard Wollan and then was bought and developed by **Atmel** in **1996**.
- AVR can have different meaning to different people. Atmel says that AVR stands for

  Advanced Virtual RISC or Alf Vegard RISC.

  RISC- Reduced Instruction Set Computer
- There are many kinds of AVR microcontrollers with different properties.
- Except of AVR32 (32 bit microcontroller), AVRs are all 8-bit microprocessors, meaning that the CPU can work on only 8 bits of data at a time. Data larger than 8 bits has to broke into pieces to be processed by the CPU.
- AVRs are not compatible in software from one family to other. To run programs written for ATtiny25 on ATmega64, we must recompile the program and possibly change some register locations.
- **AVRs are generally classified into 4 groups:**

  1.Mega 2. Tiny 3.Special Purpose 4.Classic
- **Mega family microcontrollers are widely used**.

AVR Family

AVRs are generally classified into 4 groups:

1. **Classic**
2. **Mega**
3. **Tiny**
4. **Special Purpose**

1. Classic AVR (AT90Sxxxx)

- This is the original AVR chip, which has been replaced by the newer AVR chips.
- Example: **AT90S2313, AT90S2323** etc.

2. Mega AVR (ATmegaxxxx)

- These are **powerful** microcontrollers with **more than 120 instructions** and lots of different **peripheral capabilities**, which can be used in different designs.
- Program Memory : **4k to 256k** bytes
- Package : **28 to 100 pins**
- Extensive peripheral set
- Extended instruction set: They have **rich instruction sets**.
- Example: **ATmega8, ATmega16, ATmega32**, **ATmega64** etc.

3. Tiny AVR (ATtinyxxxx)

- Have **less instructions** and smaller packages in comparison to mega family.
- Can design systems with **low costs and power consumptions** using the Tiny AVRs.
- Program Memory : **1k to 8k bytes**
- Package : **8 to 28** pins.
- Limited peripheral set.

- **Limited instruction set**: The instruction sets are limited. Some of them do not have the multiply instruction.
- Example: **ATtiny13, ATtiny25, ATtiny44** etc..

4. **Special Purpose AVR**
   - ICs of this group can be considered as a **subset of other groups**, but their special capabilities are made for **designing specific applications**.
   - Some of the special capabilities/ examples are: **USB controller, CAN controller, LCD controller**, Zigbee, FPGA, Ethernet controller and Advanced PWM.

AVR Features

1. AVR is an 8-bit RISC single chip microcontroller with Harvard architecture.
2. AVR's have some standard features such as on-chip program ROM, data RAM, data EEPROM, timers and I/O ports.
3. Most AVRs have some additional features like ADC (Analog to Digital Converter), PWM and different kinds of serial interfaces such as USART (Universal Synchronous Asynchronous Receiver Transmitter), CAN (Controller Area Network), USB(Universal Serial Bus).
4. The Program ROM size can vary from **1k to 256k**.
5. Uses on-chip **flash memory** for program storage.
6. Flash memory can be **erased easily in seconds** compared to UV-EEPROM.
7. AVR has maximum of 64k bytes of data RAM space.
8. AVR can have from **3 to 86 pins for I/O**.
9. AVR can have **up to 6 timers**.

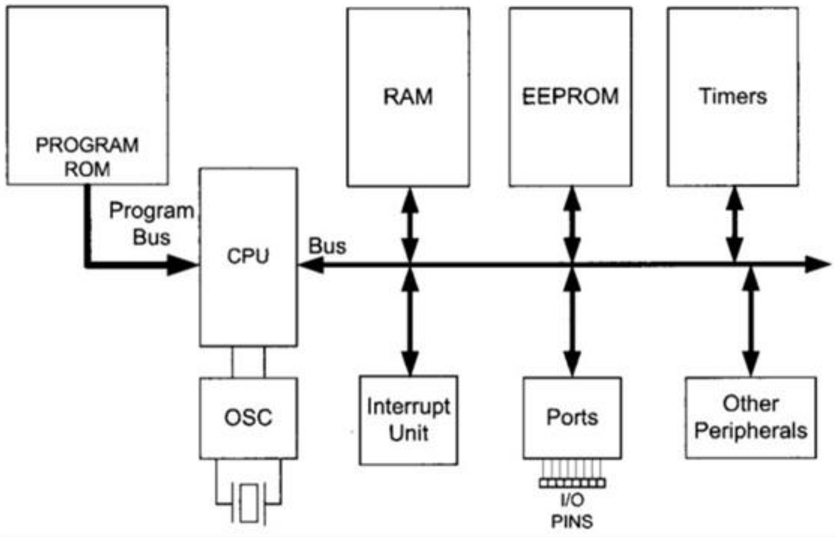


Fig. 1.3 Simplified View of an AVR Microcontroller

The General Purpose Register (GPR) in the AVR

- Registers used to store data or information temporarily.
- That information could be a byte of data to be processed, or an address pointing to the data to be fetched.
- GPR's are used for arithmetic and logic operations.
- GPR's are 8-bit registers.
- The 8 bits of register are shown in diagram below :

D7	D6	D5	D4	D3	D2	D1	D0

Fig. 1.4: 8 bits of GPRs in an AVR

- These range from MSB (Most Significant Bit) D7 to the LSB (Least Significant Bit) D0.
- In AVR there are 32 general purpose registers. All of these registers are 8 bits.
- They are R0 to R31 and located in the lowest location of memory address.
- Its shown in figure below:

R0
R1
R2
: **:** **:**
R31

Fig. 1.5: 32 General Purpose Registers in an AVR

- GPRs are same as accumulator in microprocessor; they can be used by all arithmetic and logic instructions.

AVR Data Memory

- In AVR microcontroller there are **two kinds of memory space**:
  1. **Code Memory Space**
  2. **Data Memory Space.**
- Our program is stored in code memory space, whereas data memory stores data.
- The **data memory** is composed of three parts:
  1. **GPRs(General Purpose Register)**
  2. **I/O Memory**
  3. **Internal SRAM**
- The diagram below shows Data Memory for the AVRs:

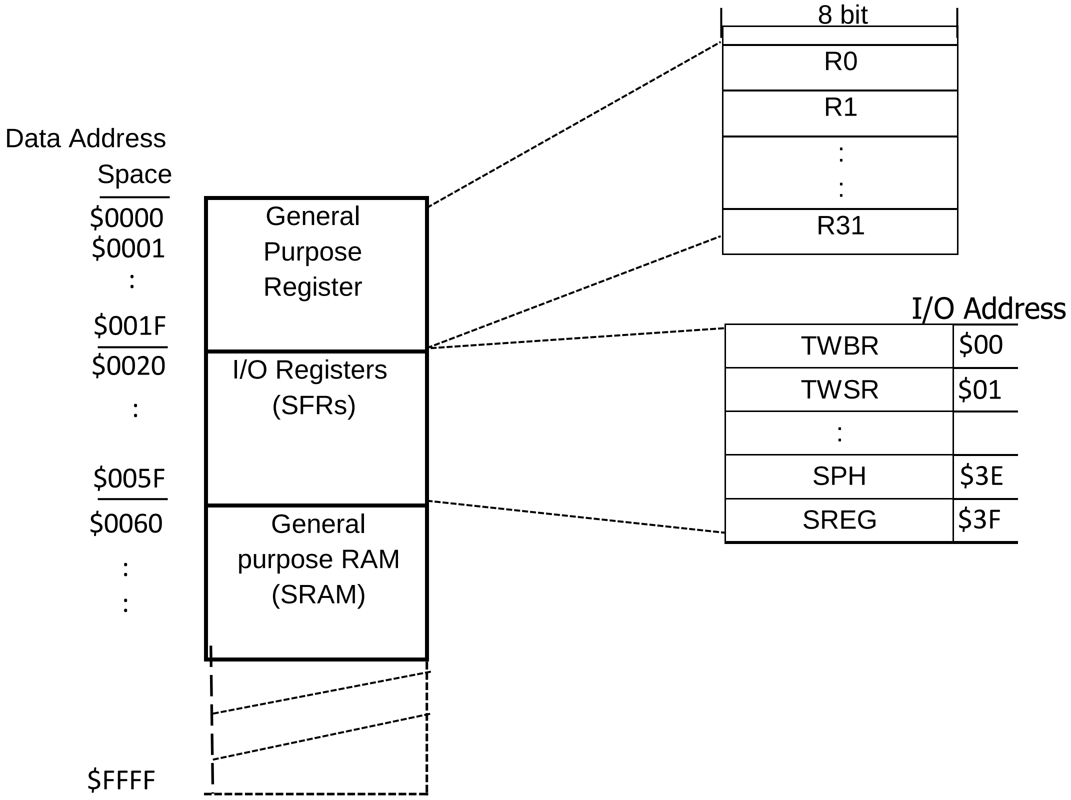


Fig. 1.6: The Data Memory for AVRs with No Extended I/O Memory

1. GPRs

- GPRs uses 32 bytes of data memory space .
- They always take the address location of $00-$1F in data memory space.

2. Input Output Memory

- 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.
- I/O memory is made of **8-bit registers**.
- The number locations in the data memory depends on the pin numbers and peripheral functions supported by the chip, although the number can vary from chip to chip even among members of same family.
- AVRs have at least **64 bytes of I/O memory location**.
- Also called **SFR (Special Function Register)** is called due to its specific function.

3. Internal Data SRAM

- SRAM- **Static RAM**
- Internal data SRAM is widely used for **storing data and parameters** by AVR programmers and C Compilers.
- Each location of the SRAM can be **accessed directly by its addresses**. See Fig. 1.6
- Size of SRAM can vary from chip to chip, even among members of the same family.

SRAM Vs EEPROM in AVR Chips / Comparison between SRAM and EEPROM in AVR Chips

SRAM	EEPROM
Used for storing data and parameters.	Used for storing data.
Loses its data when power is off.	Does not lose its data when power is off.
Used for storing data and parameters, those are changed frequently.	Used for storing data that should be rarely changed.

USING INSTRUCTIONS WITH THE DATA MEMORY

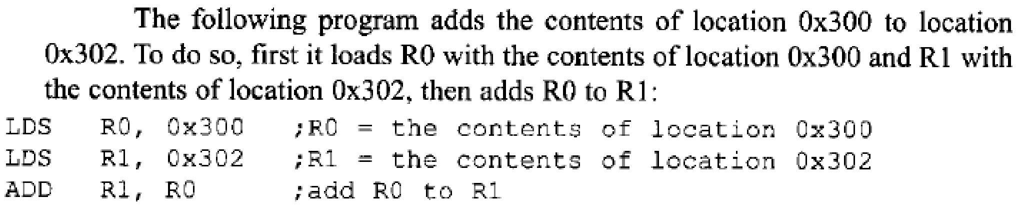
- The AVR allows direct access to locations in the data memory.

1)LDS instruction (LoaD direct from data Space)

Format:

```
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 following instruction loads R5 with the contents of location Ox200.
- LDS R5,0x200   ;load RS 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
```

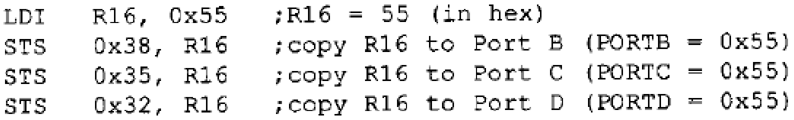
2)STS(Store direct to dataSpace)

Format:

```
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.
- The following instruction stores the contents of R25 to location Ox230.
- `STS 0x230, R25 ;store R25 to data space location 0x230`
- The following program first loads the RI 6 register with value Ox55, then moves this value around to 1/0 registers of ports B, C, and D. The addresses of PORTB, PORTC, and PORT are 0x38, Ox35, and Ox32,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)
```

3)IN (IN from I/O location)

Format:

```
IN Rd,A        ; load an I/0 location to the GPR (0 ≤ d ≥ 31),(0 ≤ A ≥ 63)
```

- The IN instruction tells the CPU to load one byte from an I/O register to the GPR.
- After this instruction is executed, the GPR will have the same value as the 1/0 register.
- For example, the "IN R20,0x16"instruction will copy the contents of location 16 (in hex) of the 1/0 memory into R20.
- Each location in I/o memory has two addresses: I/O address and data memory address.
- Each location in the data memory has a unique address called the *data memory address.* Each I/O register has a relative address in comparison to the beginning of the 1/0 memory; this address is called the *I/O address.*

IN Vs LDS

IN	LDS
Faster, it needs 1 machine cycle.	Slower, it needs 2 machine cycle.
It is 2 byte instruction.	It is 4 byte instruction.
I/O registers can be accessed by their names.	Memory can be accessed by their address only.
Available in all AVR's	Not implemented in all AVR's.

4)OUT(OUT to I/O location)

Format:

```
OUT A,Rr  ;store register to I/O location (0 ≤ r ≥ 31),(0 ≤ A ≥ 63)
```

- The OUT instruction tells the CPU to store the GPR to the I/O register.
- After the instruction is executed, the I/0 register will have the same value as the GPR.
- For example, the "OUT PORTD, R10" instruction will copy the contents of R10 into PORTD.

Instructions with GPR

1)LDI instruction

- 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."
- The following instruction loads the R20 register with a value of 0x25 (25 in hex).

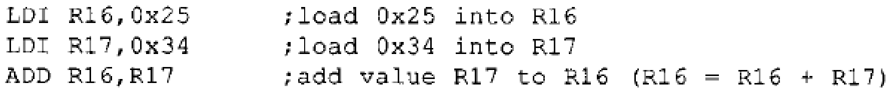
```
LOI  R20,0x25              ;load R20  with  0x25   (R20  =  0x25}
```

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

3)MOV

- Used to copy data among the GPR register of R0-R31.

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 Rl 0:

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

4)INC

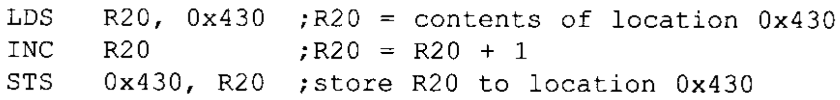
Format:

- `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 Ox430 by 1:

The following program increments the contents of data memory location Ox430 by 1:



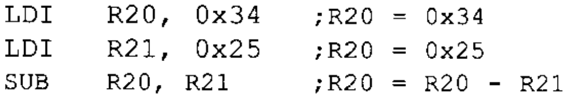
```
LDS   R20, 0x430  ;R20 = contents of location 0x430
INC   R20         ;R20 = R20 + 1
STS   0x430, R20  ;store R20 to location 0x430
```

5)SUB

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
```

6)DEC

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 RIO:

```
DEC   R10        ;R1O  = R1O - 1
```

9)COM

format:

- The "COM   Rd" instruction complements (inverts) the contents of Rd and places the result back into the Rd register.
- In the following program, we put Ox55 into Rl6 and then send it to the SFR location of PORTB.
- In the following program, we put Ox55 into Rl6 and then send it to the SFR location of PORTB.

```
LDI   R16,0x55    ;R16 = 0x55
OUT   PORTB, R16  ;copy R16 to Port B SFR (PB = 0x55)
COM   R16         ;complement R16        (R16 = 0xAA)
OUT   PORTB, R16  ;copy R16 to Port B SFR (PB = 0xAA)
```

<u>AVR data format representation</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.
- 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.

➢ ***Hex numbers***

✓ There are two ways to show hex numbers:

1. Put 0x (or 0X) in front of the number like this: LDI   R16,   0x99
2. Put$ in front of the number, like this: LDI   R22, $ 99

➢ ***Binary numbers***

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

LDI Rl6,0bl0011001   ;R16 = 10011001 or 99 in hex

✓ Can use 0b or 0B.

➢ ***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’

Assembler directives

- Directives (also called *pseudo instructions)* give directions to the assembler.
- The directives help us to make our program more readable and understand.
- eg:.EQU,.SET,.ORG

.EQU(equate)

- This is used to define a constant value or a fixed address.

```
.EQU   COUNT = 0x25
       ...    ....
       LDI    R21, COUNT         ;R21 = 0x25
```

.SET

- It is used to define a constant value or a fixed address.
- Compared to .EQU,the assigned value by .SET directive may be reassigned later.

.ORG

- 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.

```
.INCLUDE   "M32DEF.INC"
```

AVR Status Register

- AVR has a flag register to indicate arithmetic conditions such as carry bit.
- The **flag register in AVR is called as Status registe**r.
- Status register is an **8-bit** register.

D7	D6	D5	D4	D3	D2	D1	D0

Fig. 1.7: 8 bits of Status register in an AVR

- The Figure below fig 1.8 shows bits of Status Register

SREG

D7							D0
I	T	H	S	V	N	Z	C

Fig. 1.8: Each Bits of Status Register

C - Carry flag	**N - Negative flag**	**S - Sign flag**	**T - Bit Copy Storage**
Z - Zero flag	**V - Overflow flag**	**H - Half flag**	**I - Global Interrupt Enable**

- The bits **C, Z, N, V, S and H are called** ***conditional flags***, meaning that they indicate some conditions that result after an instruction is executed.

1. **Carry flag (C)**
   - This flag is set whenever there is a **carry out from the D7 bit**.
   - This flag bit is affected by 8 bit addition or subtraction
2. **Zero flag (Z)**
   - Shows the result of an arithmetic or logic operation.
   - If the result is **zero, then Z=1** and If the result is **not zero, then Z=0**.
3. **Negative flag (N)**
   - 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 of the result is **one, then N=1** and the result is negative.
4. **Overflow flag (V)**
   - Is set whenever the **result of a signed number is too large**, causing the high order bit to

overflow into sign bit.

5. **Sign bit (S)**
   - Is the result of **Exclusive OR** ing of N flag and V flag.
6. **Half carry flag (H)**
   - 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 arithmetic.
   - In some microprocessor this is called **AC flag** (Auxiliary Carry flag).
7. **Bit Copy Storage (T)**
   - Acts as **source and destination** for the Bit Copy Storage instructions.
8. **Global Interrupt Enable (I)**
   - **Activates or deactivates the interrupts**.

Difference between Carry flag and Overflow flag

Carry flag	Overflow flag
Used to detect errors in unsigned arithmetic operation.	Used to detect errors in signed arithmetic operation.

Example 1:1

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:

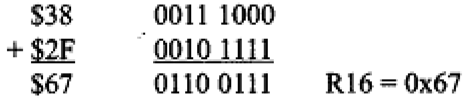
After	Value of R20	The Z flag
LDI R20, 4	4	0
DEC R20	3	0
DEC R20	2	0
DEC R20	1	0
DEC R20	0	1

Example 1:2

Show the status of the C, H, and Z flags after the addition of Ox38and Ox2Fin the following instructions:

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

Solution:

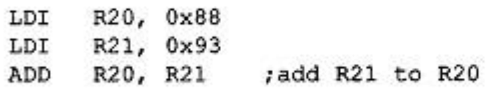


```
  $38      0011 1000
+ $2F      0010 1111
  $67      0110 0111     R16 = 0x67
```

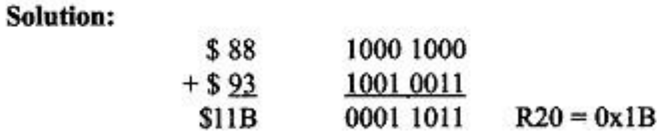
C = 0 because there is no carry beyond the D7 bit.
H = 1 because there is a carry from the D3 to the 04 bit.
Z = 0 because the R16 (the result) has a value other than 0 after the addition.

Example 1:3

Show the status of the C, H, and Z flags after the addition of Ox88 and Ox93in 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 (the result) has a value 0x1B in it after the addition.

Program counter (PC) in the AVR

- 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.
- The wider the program counter, the more memory locations a CPU can access.
- That means that a 14-bit program counter can access a maximum of 16K (2^{14} = 16K) program memory locations.
- The ATmega64 has a 15-bit program counter, so it has 32K locations.
- The program counter in the AVR family can be up to 22 bits wide.
- This means that the AVR family can access program addresses 000000 to $3FFFFF, a total of 4M locations.

ROM memory map in the AVR family

- No member of the AVR family can access more than 4M words of opcode because the program counter in the AVR can be a maximum of 22 bits wide (000000 to $3FFFFF address range).
- The AVR microcontroller wakes up at memory address 0000 when it is powered up., when the AVR is powered up, the PC (program counter) has the value of 00000 in it.
- The first opcode to be stored at ROM address $00000.

Placing code in program ROM : Executing a program instruction by instruction

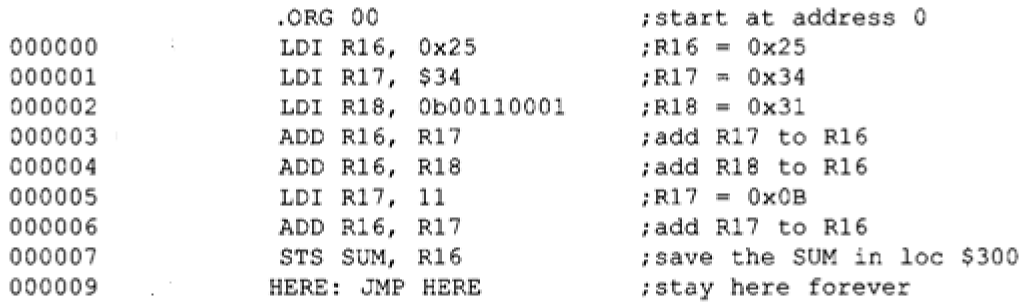
The following is a step-by-step description of the action of the AVR upon applying power to it:

1. When the AVR is powered up, the PC (program counter) has 00000 and starts to fetch the first instruction from location 00000 of the program ROM. The code for moving operand Ox25 to R16. Upon executing the code, the CPU places the value of 25 in R16 and the program counter is incremented to point to 00001 (PC = 00001).

2. Upon executing the next code, the value Ox34 is loaded to Rl 7. Then the program counter is incremented to 0002.

3. This process goes on until all the instructions up to "ADD R16, R7" are fetched and executed.
   All the below instructions are 2-byte instructions; that is, each one takes two bytes of ROM (one word).
4. Now pc=0007 points to the instruction STS SUM,R16. This is a 4 byte instruction. ie, pc=pc+2 after the execution of this instruction pc=0009.
5. Now pc=0009 , which is JMP here. It is a 4byte instruction it takes the address of 09 and this keeps the program in infinite loop.

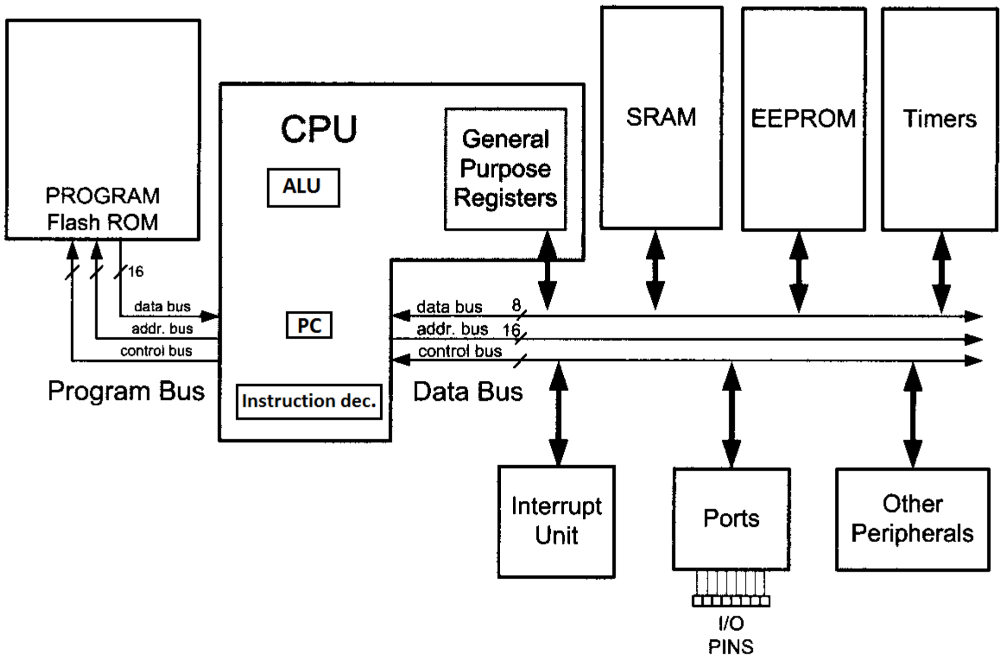
Example program:



```
                    .ORG 00                  ;start at address 0
000000              LDI R16, 0x25            ;R16 = 0x25
000001              LDI R17, $34             ;R17 = 0x34
000002              LDI R18, 0b00110001      ;R18 = 0x31
000003              ADD R16, R17             ;add R17 to R16
000004              ADD R16, R18             ;add R18 to R16
000005              LDI R17, 11              ;R17 = 0x0B
000006              ADD R16, R17             ;add R17 to R16
000007              STS SUM, R16             ;save the SUM in loc $300
000009             HERE: JMP HERE            ;stay here forever
```

Harvard architecture in the AVR

- In harvard architecture, there are separate buses for the code and data.
- Program bus provides access to the program flash ROM and data bus is used for bringing data to the CPU.
- In the program bus - data bus is 16 bit wide and address bus is as wide as the PC register.
- In the data bus - address bus is 16 bits wide and data bus is 8 bit wide.
- The memory locations are divided into lower byte and higher byte.
- The harvard architecture is shown below:



Little endian and big endian concept:

Little endian	Big endian
Low bytes go to the low memory location and high bytes goes to high memory location.	High bytes go to the low addresses and low bytes goes to high addresses.
Eg: Intel microprocessors and many microcontrollers.	Eg: Free scale microprocessors and some mainframe processors.

RISC ARCHITECTURE IN THE AVR

- There are three ways available to microprocessor designers to increase the processing power of the CPU:

1. Increase the clock frequency of the chip: One drawback of this method is that the higher the frequency, the more power and heat dissipation. Power and heat dissipation is especially a problem for hand-held devices.

2. Use Harvard architecture by increasing the number of buses to bring more information (code and data) into the CPU to be processed.

3. Change the internal architecture of the CPU and use what is called RISC architecture.

Features of RISC

1. RISC processor have a fixed instruction size : therefore the cpu can decode the instruction quickly.
2. It has 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. It avoids the need for a large stack to store parameters.
3. RISC processors have a small instruction set. This leads the program to large, use more memory and it is used in high level language.
4. More than 95% of instructions are executed with only one clock cycle.
5. RISC processors have separate buses for data and code.
6. Implemented using the hardwire method.Hardwiring of RISC instructions takes no more than 10% of the transistors.
7. RISC uses load/store architecture.
   - ✓ In RISC, instructions can only load from external memory into registers or store registers into external memory locations. There is no direct way of doing arithmetic and logic operations between a register and the contents of external memory locations.

Branch instructions and looping

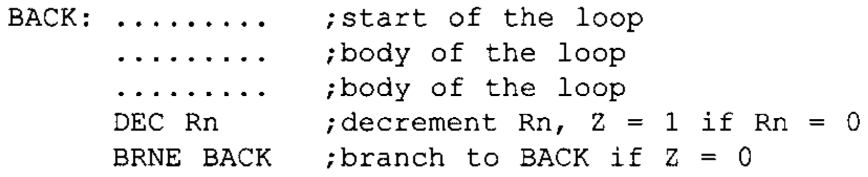
- Repeating a sequence of instructions or an operation a certain number of times is called a *loop*.
- One way is to repeat the operation over and over until it is finished
- Other way is to use a loop.
- The maximum count is 255 if we need count more than 255 use 2 counters with nested loop.
- A loop inside a loop, which is called a *nested loop*.
- In a nested loop, we use two registers to hold the count.

- There are 2 types of branch instructions:
  1)Conditional branch instructions
  2)Unconditional branch instructions

1)Conditional branch (jump) instructions

Using BRNE instructionfor 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
```

- In the last two instructions, the Rn (e.g., R16 or RI7) is decremented; if it is not zero, it branches (jumps) back to the target address referred to by the label.

AVR conditional branch(jump) instructions: Some conditional branch instruction as follows:

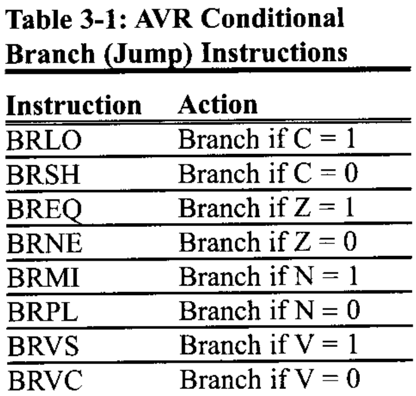


Table 3-1: AVR Conditional Branch (Jump) Instructions

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

- Conditional branch instructions were 2-byte instructions.

2) Unconditional branch(jump) instruction

- The unconditional branch is a jump in which control is transferred unconditionally to the target location.
- In the AVR there are three unconditional branches:

1. JMP(Jump) 2. RJMP (relative jump) 3. IJMP (indirect jump).

JMP (JMP is a long jump)

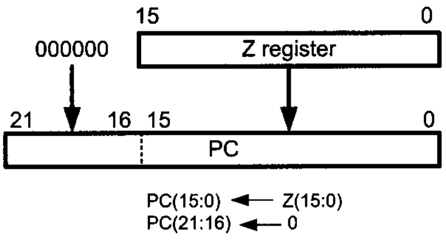
- JMP is an unconditional jump that can go to any memory location in the 4M (word) address space of the AVR.
- It is a 4-byte (32-bit) instruction.
- 10 bits are used for the opcode, and the other 22 bits represent the 22-bit address of the target location.
- The 22-bit target address allows a jump to 4M (words) of memory locations from 000000 to $3FFFFF.

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.
- 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.
- In the other jump instructions, the target address is static, which means that in a specific condition they jump to a fixed point.
- But IJMP has a dynamic target point, and we can dynamically change the target address by changing the Z register.



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.
- There are 4 CALL instructions:

  1) CALL 2)RCALL 3)ICALL 4)EICALL

CALL

- In this 4-byte (32-bit) instruction.
- 10 bits are used for the opcode and the other 22 bits, are used for the address of the target subroutine.
- CALL can be used to call subroutines located anywhere within the 4M address space of 000000–$3FFFFF.
- 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.

ICALL (indirectcall)

- 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.

RCALL (relative call)

- RCALL is a 2-byte instruction in contrast to CALL, which is 4 bytes.
- 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.
- The Z register(16 bits) should contain the address of a function when the ICALL instruction is executed.

EICALL(Extended indirect call)

- In the AVRs with more than 64K words of program memory, the EICALL (extended indirect call) instruction is available.
- The EICALL loads the Z register into the lower 16 bits of the PC and the EIND register(part of I/O memory) into the upper 6 bits of the PC.

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

- The register used to access the stack is called the SP (stack pointer) register.
- 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.
- The stack pointer must be wide enough to address all the RAM. So, in the AVRs with more than 256 bytes of memory the SP is made of two 8-bit registers (SPL and SPH),
- While in the AVRs with less than 256 bytes the SP is made of only SPL, as an 8-bit register can address 256 bytes of memory.
- The storing of CPU information such as the program counter on the stack is called a **PUSH**.
- The loading of stack contents back into a CPU register is called a **POP**.

Pushing onto the stack

- The stack pointer (SP) points to the top of the stack (TOS).
- As we push data onto the stack, the data are saved where the SP points to, and the SP is decremented by one.
- To push a register onto stack we use the PUSH instruction.

```
PUSH Rr          ;Rr can be any of the general purpose registers (R0-R31)
```

- For example, to store the value of R10 we can write the following instruction:

```
PUSH R10
```

Popping from the stack

- Popping the contents of the stack back into a given register is the opposite process of pushing.
- When the POP instruction is executed, the SP is incremented and the top location of the stack is copied back to the register.
- Stack is LIFO (Last-In-First-Out).
- To retrieve a byte of data from stack we can use the POP instruction.

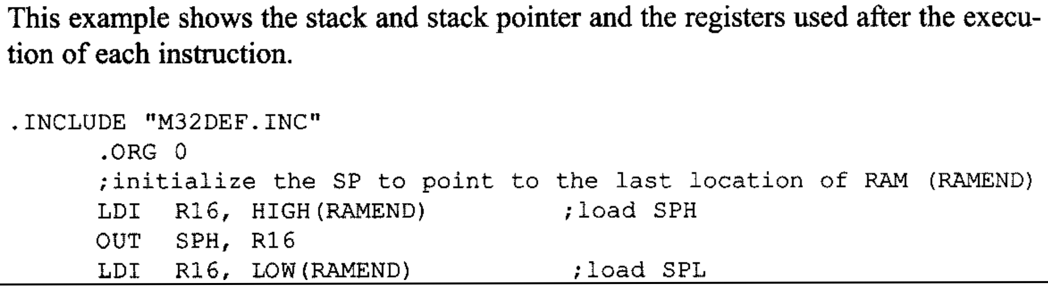
```
POP Rr          ;Rr can be any of the general purpose registers (R0-R31).
```

- For example, the following instruction pops from the top of stack and copies to R1O:

```
POP RI6          ;increment SP, and then load the top of stack to R10
```

Initializing the stack pointer

- When the AVR is powered up, the SP register contains the value 0..
- We must initialize the SP at the beginning of the program so that it points to somewhere in the internal SRAM.
- In AVR, the stack grows from higher memory location to lower memory location (when we push onto the stack, the SP decrements).
- Different AVRs have different amounts of RAM.
- In the AVR assembler RAMEND represents the address of the last RAM location.
- To initialize the SP so that it points to the last memory location, we can simply load RAMEND into the SP.
- SP is made of two registers, SPH and SPL. So, we load the high byte of RAMEND into SPH, and the low byte of RAMEND into the SPL.



This example shows the stack and stack pointer and the registers used after the execution of each instruction.

```
.INCLUDE "M32DEF.INC"
      .ORG 0
      ;initialize the SP to point to the last location of RAM (RAMEND)
      LDI   R16, HIGH(RAMEND)         ;load SPH
      OUT   SPH, R16
      LDI   R16, LOW(RAMEND)          ;load SPL
```

CALL instruction and the role of stack

- When a subroutine is called, the processor first saves the address of the instruction just below the CALL instruction on the stack, and then transfers control to that subroutine.
- This is how the CPU knows where to resume when it returns from the called subroutine.

RET instruction and the role of stack

- When the RET instruction at the end of the subroutine is executed, the top location of the stack is copied back to the program counter and the stack pointer is incremented.
- When the CALL instruction is executed, the address of the instruction below the CALL instruction is pushed onto the stack; so, when the execution of the function finishes and RET is executed, the address of the instruction below the CALL is loaded into the PC, and the instruction below the CALL instruction is executed.

Delay calculation for the AVR

- In creating a time delay using Assembly language instructions, we consider *two factors that can affect the accuracy of the delay*:
  1. **The crystal frequency**: The frequency of the crystal oscillator connected to the XTAL1 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**: 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,
     (c) use pipelining to overlap fetching and execution of instructions.

Pipelining

- Pipelining allow the CPU to fetch and execute at the same time.
- 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.
- The execution of many instructions is overlapped.
- The speed of execution is limited to the slowest stage of the pipeline.
- In the AVR, each instruction is executed in 3 stages:
  1. Operand fetch
  2. ALU operation execution
  3. Result write back.
- Single cycle ALU operation is shown below:

Stage 1	Stage 2	Stage 3
READ OPERANDS	PROCESS	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.

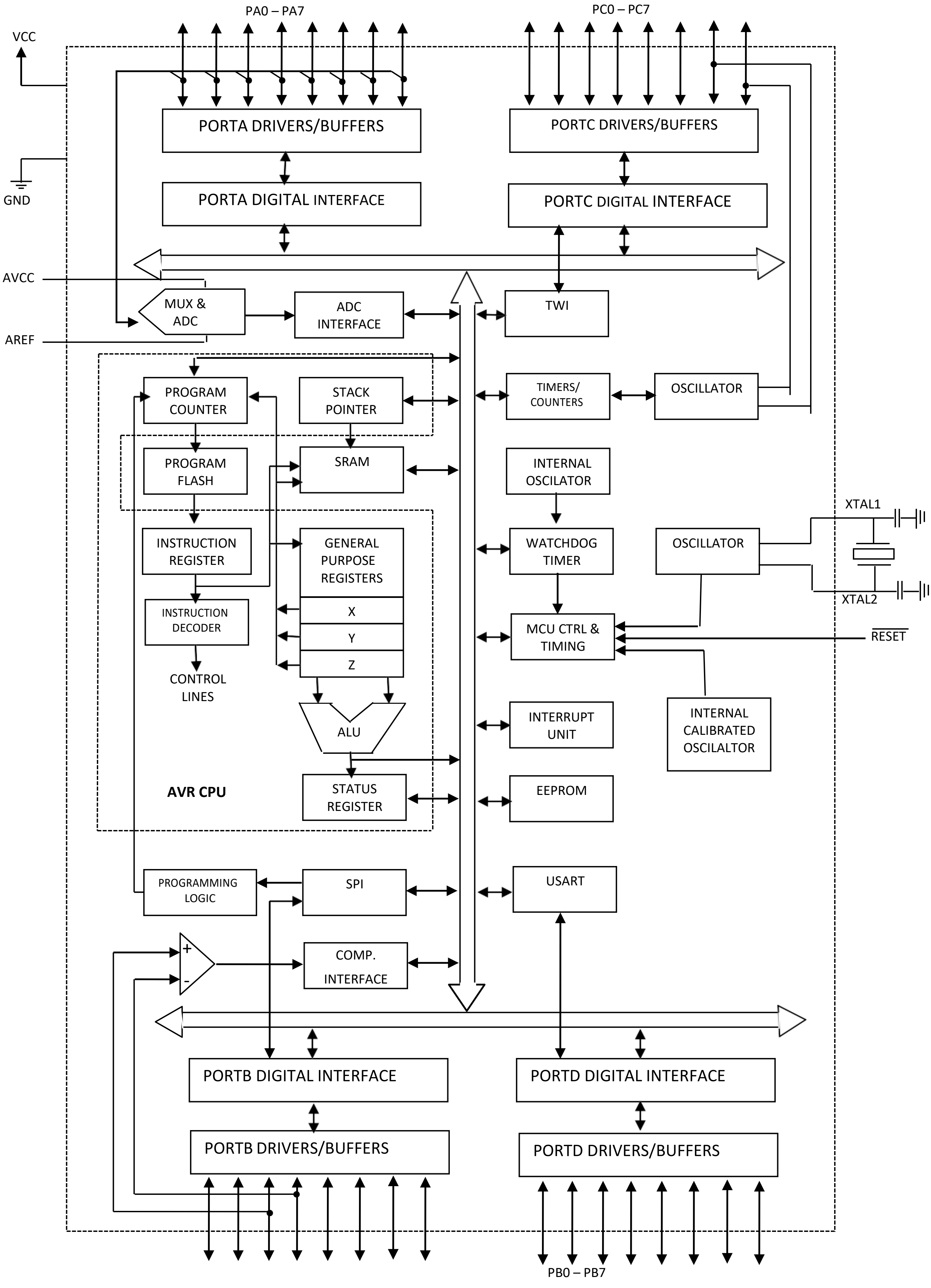
Delay calculation for AVR

✓ A delay subroutine consists of two parts: (1) setting a counter, and (2) a loop

- We calculate the time delay based on the instructions inside the loop and ignore the clock cycles associated with instructions outside the loop.

- One way to increase delay is to use NOP.
- NOP stands for 'no operation', simply waste time, but takes 2 bytes of program ROM space.
- So better way to get a large delay is to use a loop inside a loop, which is also called a *nested loop*.

ATmega32 Block Diagram


PA0 – PA7
PC0 – PC7
VCC
GND
AVCC
AREF
PORTA DRIVERS/BUFFERS
PORTA DIGITAL INTERFACE
PORTC DRIVERS/BUFFERS
PORTC DIGITAL INTERFACE
MUX & ADC
ADC INTERFACE
TWI
PROGRAM COUNTER
STACK POINTER
PROGRAM FLASH
SRAM
INSTRUCTION REGISTER
GENERAL PURPOSE REGISTERS
X
Y
Z
INSTRUCTION DECODER
CONTROL LINES
ALU
AVR CPU
STATUS REGISTER
TIMERS/ COUNTERS
OSCILLATOR
INTERNAL OSCILATOR
WATCHDOG TIMER
OSCILLATOR
XTAL1
XTAL2
MCU CTRL & TIMING
RESET
INTERRUPT UNIT
INTERNAL CALIBRATED OSCILALTOR
EEPROM
PROGRAMMING LOGIC
SPI
USART
+
-
COMP. INTERFACE
PORTB DIGITAL INTERFACE
PORTD DIGITAL INTERFACE
PORTB DRIVERS/BUFFERS
PORTD DRIVERS/BUFFERS
PB0 – PB7