ICT5307: Embedded System Design

Lecture 3
Status Register, Stack and Pointer, Subroutine, LCD, 7-Segment Display, Inputting Data

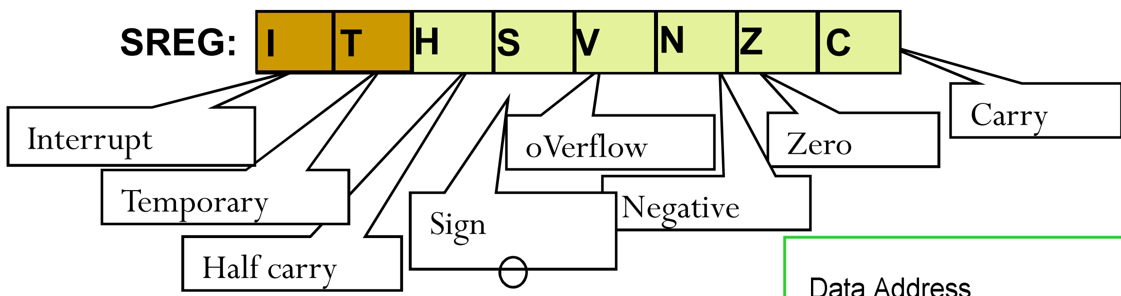
Professor S.M. Lutful Kabir
IICT, BUET

Status Register, SREG

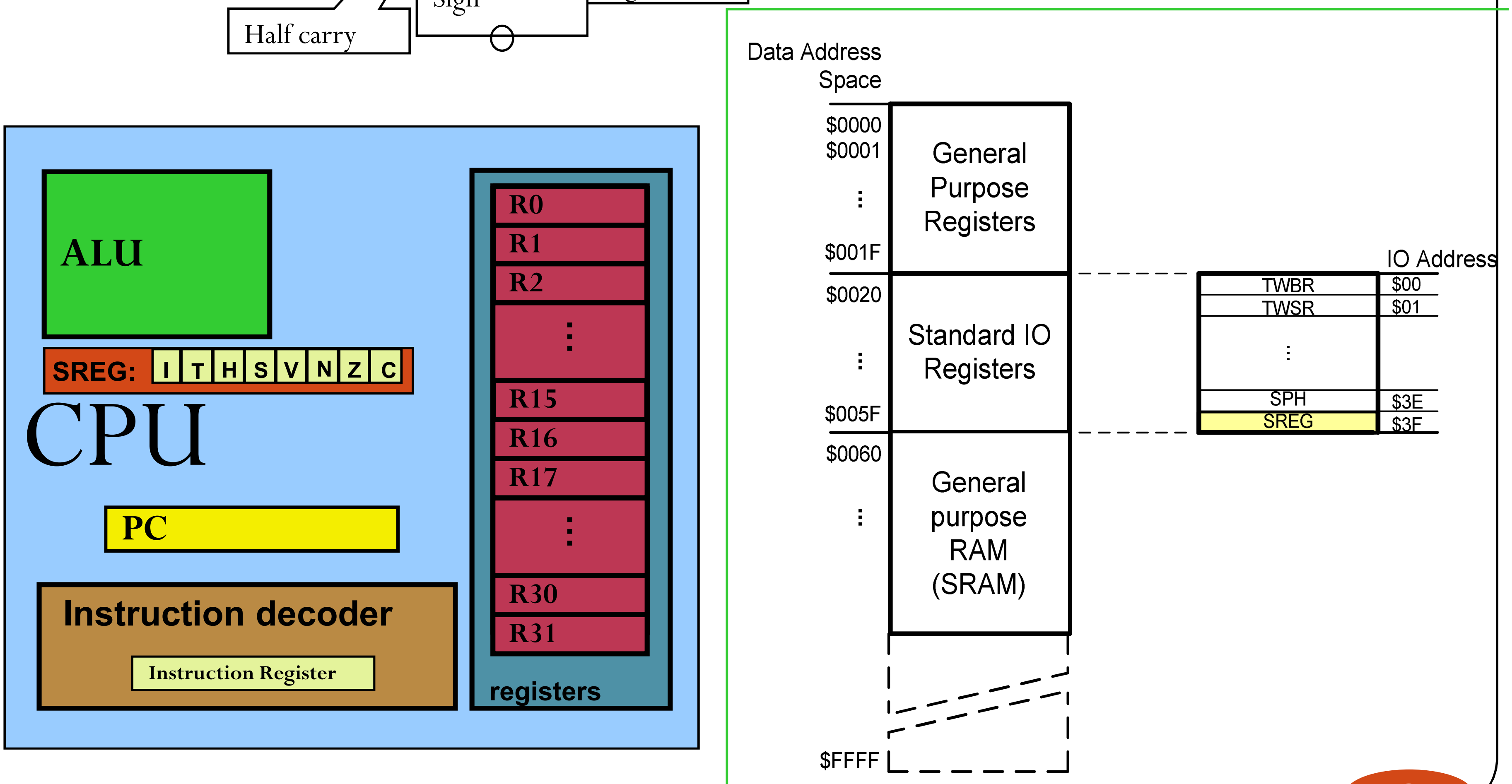
I	T	H	S	V	N	Z	C
D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0

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

'C', 'Z' and 'H' flag



SREG:
I
T
H
S
V
N
Z
C
Interrupt
Temporary
Half carry
Sign
oVerflow
Negative
Zero
Carry



ALU
SREG:
I
T
H
S
V
N
Z
C
CPU
PC
Instruction decoder
Instruction Register
R0
R1
R2
R15
R16
R17
R30
R31
registers
Data Address Space
$0000
$0001
$001F
$0020
$005F
$0060
$FFFF
General Purpose Registers
Standard IO Registers
General purpose RAM (SRAM)
IO Address
TWBR
$00
TWSR
$01
SPH
$3E
SREG
$3F

Example: Show the status of the C, H, and Z flags after the subtraction of 0x9C from 0x9C in the following instructions:

```
LDI     R20, 0x9C
LDI     R21, 0x9C
SUB     R20, R21        ;subtract R21 from R20
```

Solution:

	$9C	**1001 1100**	
-	**$9C**	**1001 1100**	
	$00	**0000 0000**	**R20 = $00**

C = 0 because R21 is not bigger than R20 and there is no borrow from D8 bit.
Z = 1 because the R20 is zero after the subtraction.
H = 0 because there is no borrow from D4 to D3.

Example: Show the status of the C, H, and Z flags after the subtraction of 0x73 from 0x52 in the following instructions:

```
LDI    R20, 0x52
LDI    R21, 0x73
SUB    R20, R21       ;subtract R21 from R20
```

Solution:

	$52	***0101 0010***	
-	***$73***	***0111 0011***	
	$DF	***1101 1111***	***R20 = $DF***

C = 1 because R21 is bigger than R20 and there is a borrow from D8 bit.
Z = 0 because the R20 has a value other than zero after the subtraction.
H = 1 because there is a borrow from D4 to D3.

Example: Show the status of the C, H, and Z flags after the subtraction of 0x23 from 0xA5 in the following instructions:

```
LDI    R20, 0xA5
LDI    R21, 0x23
SUB    R20, R21        ;subtract R21 from R20
```

Solution:

	$A5	**1010 0101**	
-	**$23**	**0010 0011**	
	$82	**1000 0010**	**R20 = $82**

C = 0 because R21 is not bigger than R20 and there is no borrow from D8 bit.
Z = 0 because the R20 has a value other than 0 after the subtraction.
H = 0 because there is no borrow from D4 to D3.

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

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

Solution:

	1	
$9C	1001 1100	
+ $64	0110 0100	
$100	1 0000 0000	R20 = 00

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

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

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

Solution:

	1	
$38	0011 1000	
+ $2F	0010 1111	
$67	0110 0111	R16 = 0x67

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

Negative flag

- N, the Negative flag
  D7 represents negative bit in signed representation so if D7=0, it is a positive and if D7=1, it is negative number
  In case of negative number the magnitude is represented by 2's complement
  -128 = 1000 0000
  - 34 = 1101 1110

Overflow flag

- While using signed numbers, a serious problem sometimes arises that must be dealt with.
- If the result of an operation is too large for the register, it is called an overflow and the programmer is notified by raising the 'V' flag.
- For example,

<u>EXAMPLE 1</u>

```
+96     0110 0000
+70     0100 0110
----------------------
+166    1010 0110
```

Carry from D6 to D7,
No carry from D7 to out

N=1, SUM=-90 => V=1
(wrong)

<u>EXAMPLE 2</u>

```
-128    1000 0000
  -2    1111 1110
--------------------
-130    0111 1110
```

No carry from D6 to D7,
Carry from D7 to out

N=0, SUM=+126 => V=1
(wrong)

Sign flag

- When overflow occurs, V=1
- But 'N' flag, which should indicate whether the result is positive or negative, gives wrong sign.
- S, the Sign flag, indicates actual sign of the result when it is corrupted due to overflow.
- And S flag is a result of EX-OR between N and V flag
- In Example 1, $S=N \oplus V=0$ and in Example 2, $S=N \oplus V=1$

'T' Bit Copy Storage

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

"Bit 6 – T: Bit Copy Storage

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

‘I’ Global Interrupt Enable

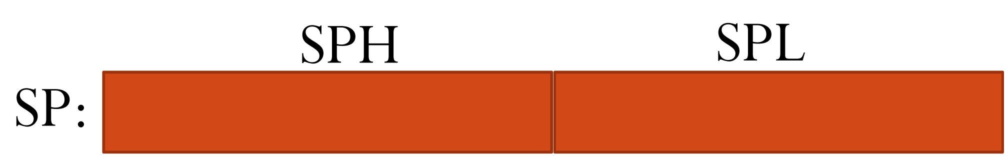
Bit	7	6	5	4	3	2	1	0	
	I	T	H	S	V	N	Z	C	SREG
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

“Bit 7 – I: Global Interrupt Enable

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

Stack and Stack Pointer (SP) in AVR

- The Stack is a section of RAM used by the CPU to store information temporarily.
- The information could be data or an address. The CPU needs this storage area because there are only a limited number of registers
- A register called Stack Pointer (SP) is used to access the Stack memory area. It contains the address at which temporary data has to be stored at present.
- SP is implemented as two registers, SPL and SPH, lower and higher byte of SP respectively.



Pushing and Popping into Stack

- The storing of information on the stack is called PUSH and loading of stack content back into a CPU register is called a POP.
- In other words, a register is pushed into stack to save it and popped off the stack to retrieve it
- The Stack pointer (SP) points to the top of the stack (TOS).
- As we push data onto the stack, the data is saved where SP points to, and Sp is decremented by one.

PUSH and POP Instructions

- To push a register onto stack we use PUSH instruction:

  PUSH Rr ; Rr can be any of the general purpose registers (R0-R31), and SP is decremented
- Popping is opposite to Pushing.
- When POP instruction is executed, the SP is incremented and the top location of the stack is copied back to the register.
- That means POP is a LIFO (Last In First Out) Memory.
- The format of POP instruction is as follows:

  POP Rr ;incremented SP and the load the top of the stack (from location indicated by the present value of SP after increment) to Rr (Rr can be from R0-R31)

Initializing the Stack Pointer (SP)

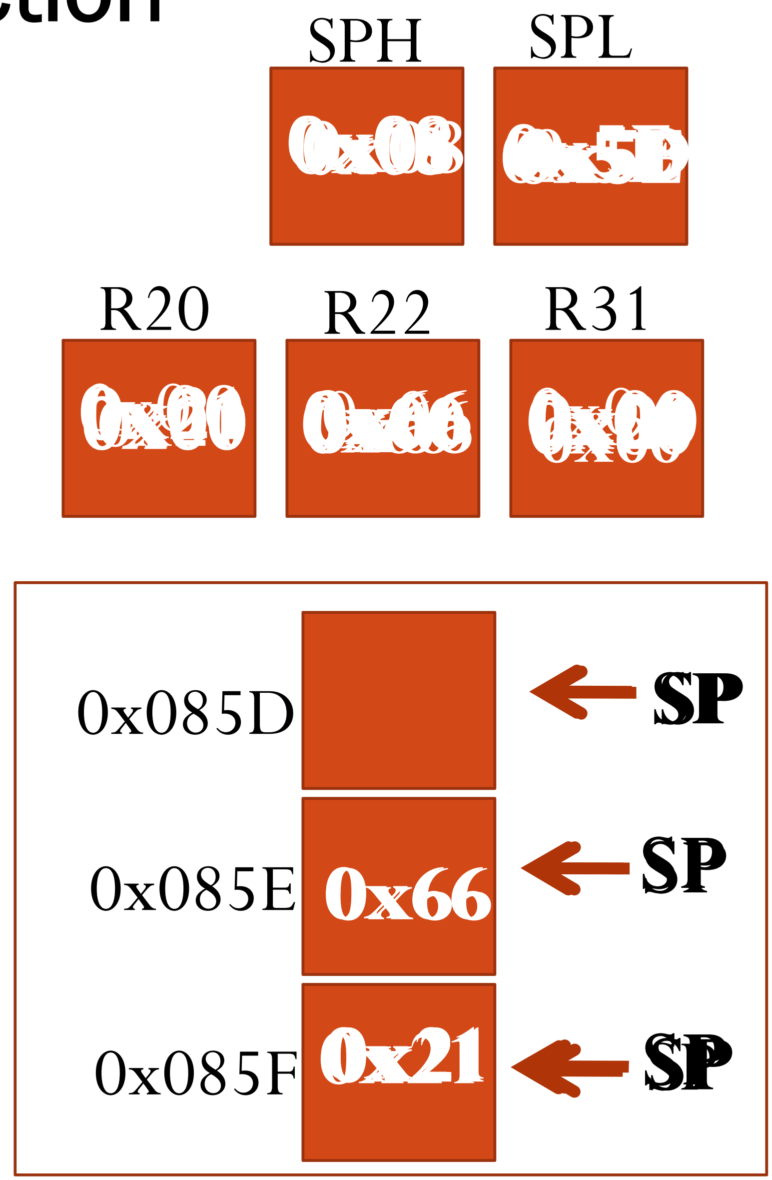
- When AVR is powered up, the SP register contains the value of 0, which is the address of R0.
- Therefore, we must initialize the SP at the beginning of the program so that it points to somewhere in the internal SRAM.
- In AVR, the stack grows from higher memory location to lower memory location.
- So, it is common to initialize the SP to uppermost memory location.
- RAMEND represents the address of the last memory location, so we can simply load RAMEND into SP. [high byte of RAMEND to SPH and low byte of RAMEND to SPL

ATMega32 Programmer Model: Memory

Type	Flash		RAM		EEPROM	
	F_END	Size, kB	RAM-END	Size, kB	E_END	Size, kB
Atmega8	$0FFF	8	$045F	1	$1FF	0.5
Atmega32	**$3FFF**	**32**	**$085F**	**2**	**$3FF**	**1**
Atmega64	$7FFF	64	$10FF	4	$7FF	2
Atmega128	$FFFF	128	$10FF	4	$FFF	4

Example of Initializing SP and PUSH and POP Instruction

```
LDI  R16, HIGH (RAMEND)
OUT  SPH, R16
LDI  R16, LOW (RAMEND)
OUT  SPL, R16
LDI  R31,0
LDI  R20, 0x21
LDI  R22, 0x66
PUSH  R20
PUSH  R22
LDI  R20, 0
LDI  R22,0
POP  R22
POP  R31
```



CALL Instruction

- CALL is a control transfer instruction used for calling a subroutine.
- Subroutines are often used to execute tasks that need to be performed frequently.
- This makes a program more structured and saves memory space.
- In AVR there are four instruction for calling subroutines.
  - CALL (Long Call)
  - RCALL (Relative Call)
  - ICALL (Indirect Call to Z)
  - EICALL (Extended Indirect Call to Z)
- The Choice of which one to use depends on target address

Returning from Subroutine

- When a subroutine is called, control is transferred to that subroutine.
- The processor saves the PC (program counter) on the stack.
- **Note: PC normally contains the address of the next instruction to be executed. So when CALL instruction is executed, PC contains the address of the instruction immediately after the CALL**
- After finishing execution of the subroutine, the RET (short of Return) instruction transfers control back to the caller
- This happens by retrieving the data from the stack

CALL Instruction and the Role of the 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 transfer controls to the subroutine.
- This is how the CPU knows where to resume when it returns from the called subroutine
- The values of the PC is broken into two bytes.
- The higher byte is pushed into the stack first and then the lower byte is pushed.

RET Instruction and Role of Stack

- When RET instruction at the end of the subroutine is executed, the memory content at the top of stack is copied to PC. (And the SP is incremented).
- Since this top content is nothing but the address next to the CALL instruction (PUSHed during CALL), so CPU will again start executing from the position from where it was branched out.

An Exercise on Stack Memory Related to Subroutine Call and Return

```
.ORG  0000
LDI  R16, HIGH (RAMEND)
OUT  SPH, R16
LDI  R16, LOW (RAMEND)
OUT  SPL, R16
BACK:
    LDI  R16, 0x55
    OUT PORTB, R16
    CALL DELAY
    LDI  R16, 0xAA
    OUT PORTB, R16
    CALL DELAY
    RJMP BACK
```

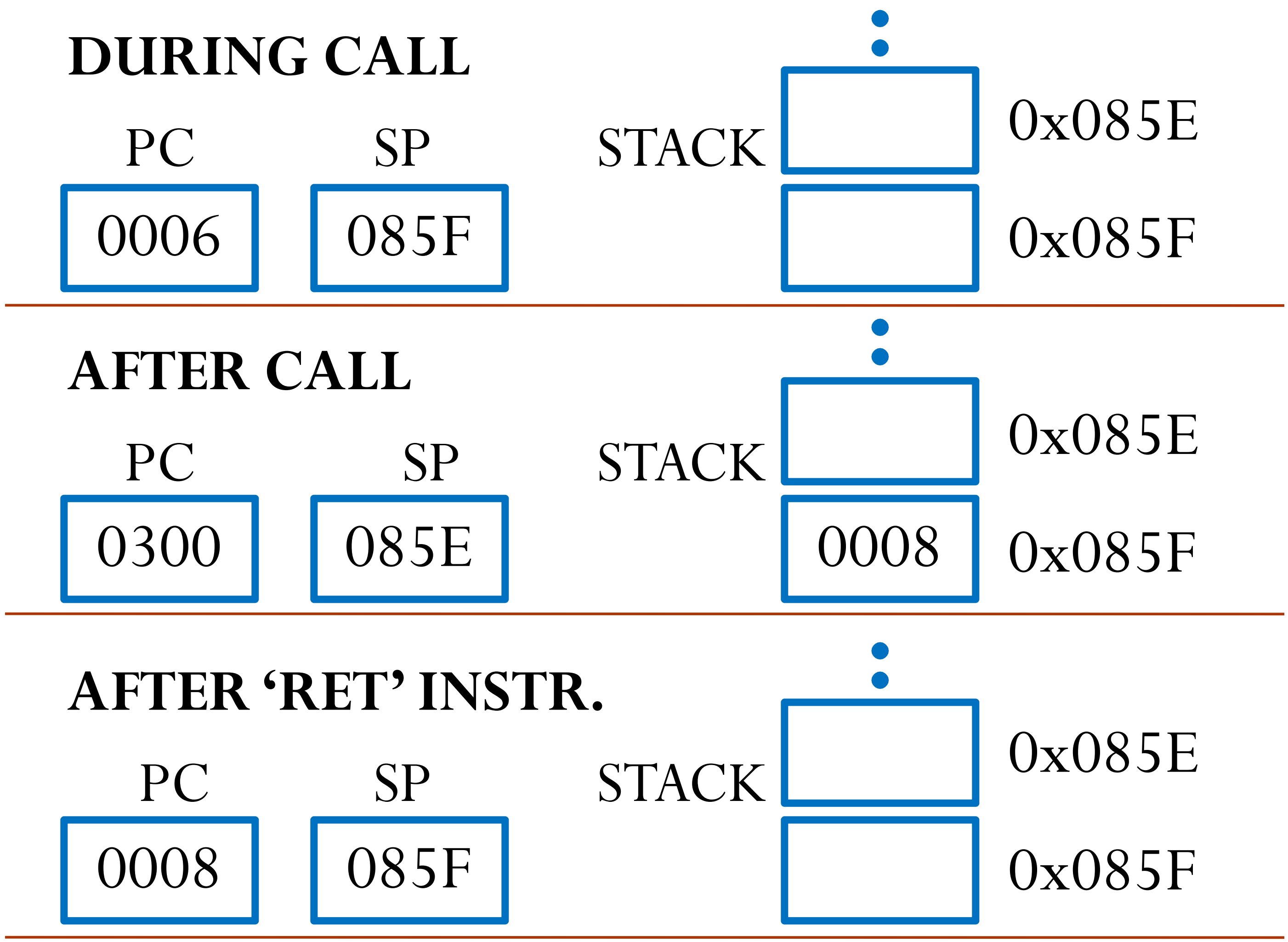
```
.ORG  0300
DELAY:
    LDI R20, 0xFF
    AGAIN:
        NOP
        NOP
        DEC R20
        BRNE AGAIN
    RET
```

An Exercise on Stack Memory Related to Subroutine Call and Return

0000	LDI R16, HIGH (RAMEND)	
0001	OUT SPH, R16	
0002	LDI R16, LOW (RAMEND)	
0003	OUT SPL, R16	
0004	LDI R16, 0x55	← BACK
0005	OUT PORTB, R16	
0006	CALL DELAY	
0008	LDI R16, 0xAA	
0009	OUT PORTB, R16	
000A	CALL DELAY	
000C	RJMP BACK	

0300	LDI R20, 0xFF	← DELAY
0301	NOP	← AGAIN
0302	NOP	
0303	DEC R20	
0304	BRNE AGAIN	
0305	RET	

Regarding first CALL



Calling a Subroutine from another Subroutine

- When a subroutine is called from the main program, the location of the program where to return is stored in the Stack.
- Say a new subroutine is called from the first subroutine the location of return for the second (inner) subroutine is stored in the stack in the same manner.
- At first the return will occur for the 2nd routine (LIFO) and then it will be for the 1st subroutine

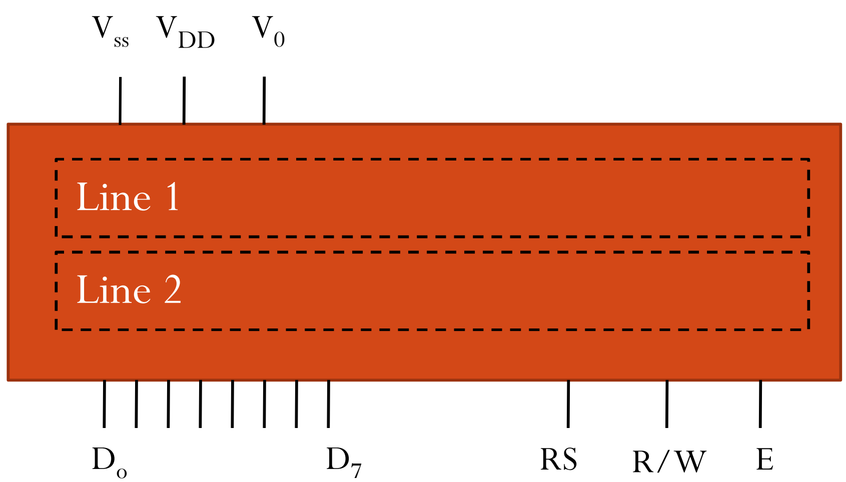
```
main() {
    ...
    CALL subroutine1()
    ...
}
void subroutine1() {
    ...
    CALL subroutine2()
    ...
    RET
}
void subroutine2() {
    .....
    .....
    RET
}
```

LCD Display : Its advantages

- LED consumes lot of power
- Advantages of LCD
  - Low power consumption
  - Easy to read in bright light
  - Declining cost
  - Ability to display both Alphanumeric and Graphics
  - In different form, 7-segment & graphics form

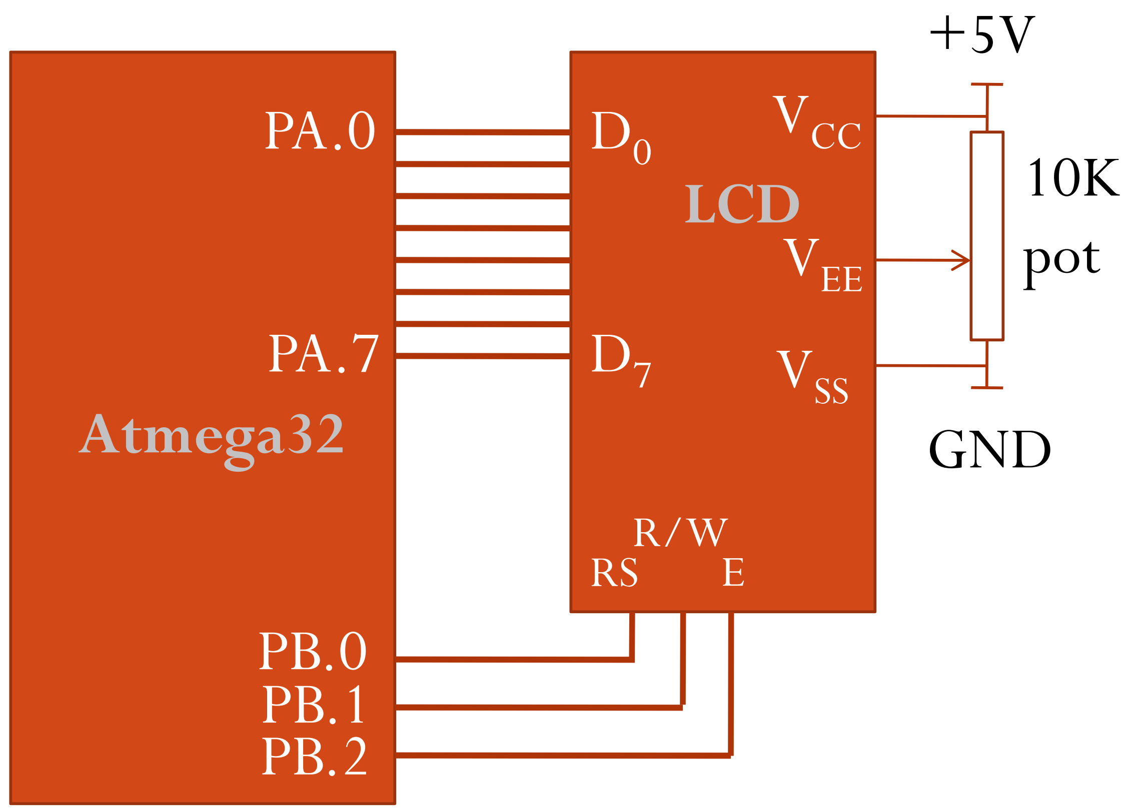
Intelligent Controller and LCD Display Panel

- An LCD panel and a small circuit board containing the controller chip
- 14 pin connection
- 2 rows, 20/40 characters in each row
- Easy to program
- Each character is displayed on a 5X7 or 5X11 dot matrix display

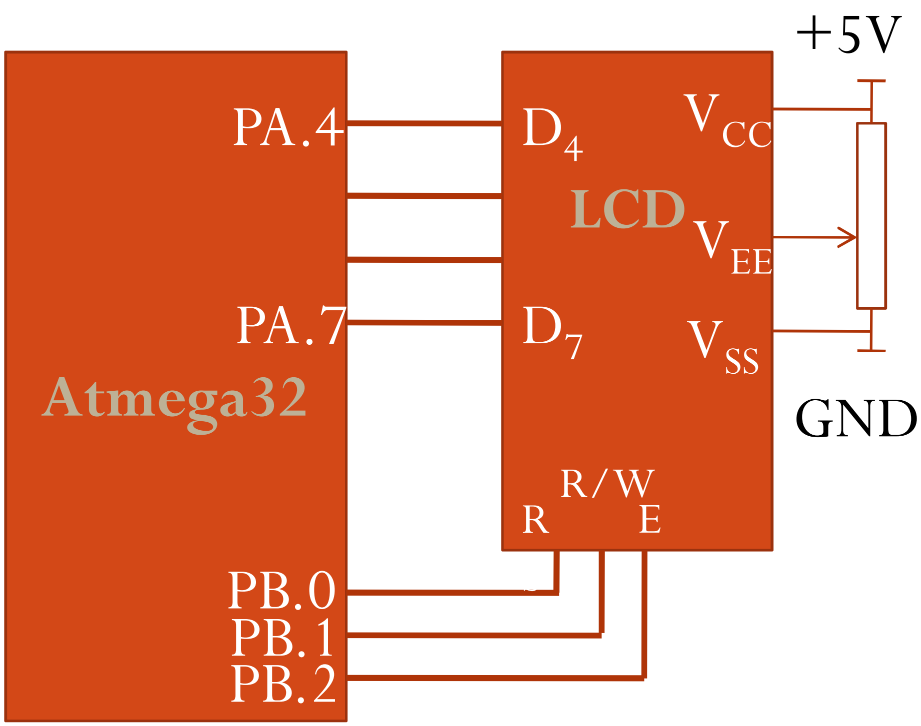


Hitachi's HD 44780 LCD module

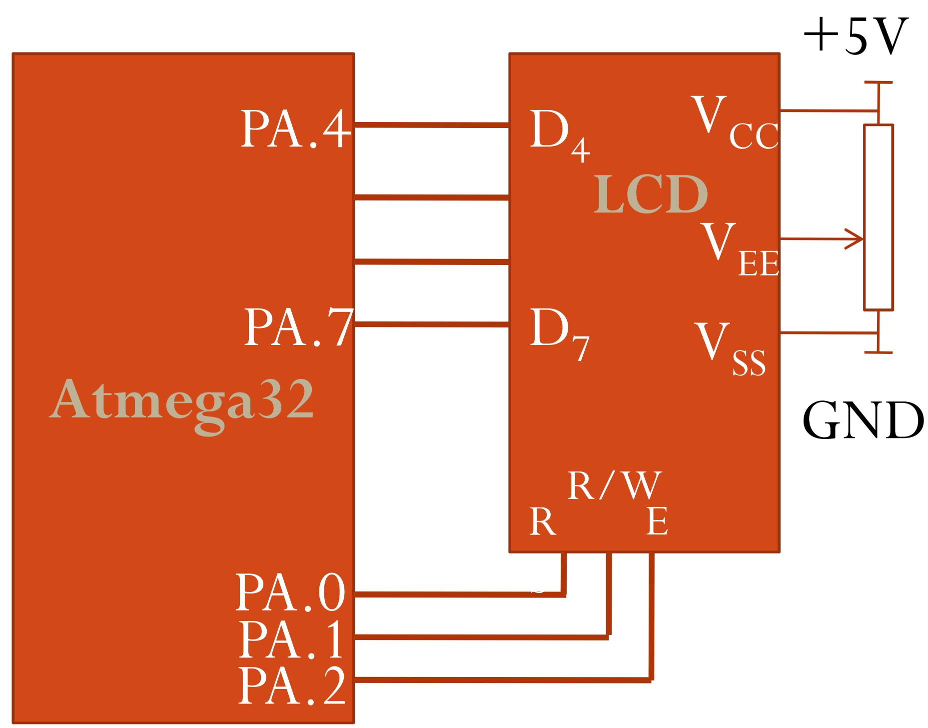
8-bit Connection



4-bit Connection



Data and Control from Different Ports



Data and Control from a single port

Inside the Display Controller

- CG ROM stores segment patterns of 192 char
- CG RAM stores segment patterns for 16-user designed characters
- An 8-bit instruction register
- An 8-bit data register
- DD RAM stores 80 numbers of 8-bit character codes
- 11 instructions
  - To clear display
  - To write a character
  - To select a position
  - To read information from the display, etc.

LCD Command Codes

Code (Hex)	Command	Code (Hex)	Command
1	Clear screen	F	Display on, cursor blink
2	Return home	10	Shift cursor to left
4	Shift cursor left	14	Shift cursor to right
6	Shift cursor right	18	Shift entire display to left
5	Shift display right	1C	Shift entire display to right
7	Shift display left	80	Cursor to the beginning of Line 1
8	Display off, cursor off	C0	Cursor to the beginning of Line 2
A	Display on, cursor off	28	2 lines, 5X7 matrix, 4 bit
C	Display off, cursor on	38	2 lines, 5X7 matrix, 8 bit
E	Display off, cursor blink		

Sending Data to LCD

- To send the data or command you should go through the following steps

  1. Initialize the LCD
  2. Send any of the commands
  3. Send Character to be Shown in the LCD

Initializing the LCD for 8-bit operation

- To initialize for 2 lines, 5X7 matrix and 8 bit operation the sequence of commands 0x38, 0x0E and 0x01 should be executed.
- If initialization is the first command in your code wait for 15 msec just after power up, if not, it is not necessary

Sending Commands to the LCD

- To send any of the commands, make pin RS and R/W both '0' and the command code in pin D0-D7
- Then send a high-to-low pulse at pin E to enable the internal latch of the LCD
- For each of command you should wait at least 100 usec
- But for clearing LCD (code=0x01) and Return home (code=0x02) you should wait for 2 msec

Sending Data to the LCD

- To send any of the commands, make pin RS '1' and R/W '0' and then put data in pin D0-D7
- Then send a high-to-low pulse at pin E to enable the internal latch of the LCD
- For each of command you should wait at least 100 usec

Sending Code or Data in 4-bit mode

- In most of cases, it is preferred to use 4-bit data to save pins.
- In this case initialization is different.
- In 4-bit, we initialize LCD with 0x33, 0x32 and 0x28
- The nibble 3, 3, 3 and 2 tells the LCD to go to 4-bit mode and 0x28 initialize the LCD for 5X7 matrix and 4-bit operation

The Connection of Pins for LCD

- The LCD module must be connected to the port bits as follows:

  [LCD] [AVR Port]

  RS (pin4) ------ PD.4

  RD (pin 5) ------ PD.5

  EN (pin 6) ------ PD.6

  DB4 (pin 11) --- PC.4

  DB5 (pin 12) --- PC.5

  DB6 (pin 13) --- PC.6

  DB7 (pin 14) --- PC.7

- You must also connect the LCD power supply and contrast control voltage, according to the data sheet.

Some of the functions used for LCD

- **unsigned char lcd_init(unsigned char lcd_columns)**

  initializes the LCD module, clears the display and sets the printing character position at row 0 and column 0. The numbers of columns of the LCD must be specified (e.g. 16). No cursor is displayed.

- **void lcd_clear(void)**

  clears the LCD and sets the printing character position at row 0 and column 0.

- **void lcd_gotoxy(unsigned char x, unsigned char y)**

  sets the current display position at column x and row y. The row and column numbering starts from 0.

- **void lcd_putsf(char flash *str)**

  displays at the current display position the string str, located in FLASH

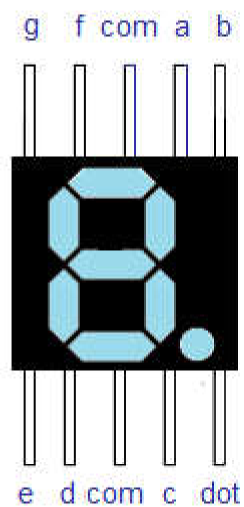
Main function

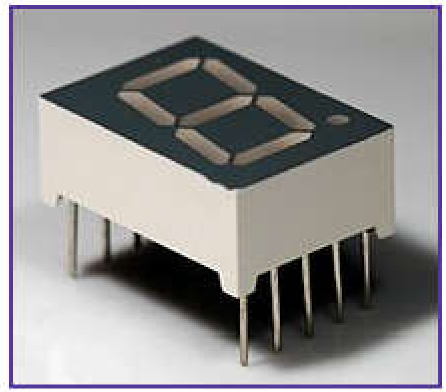
```
void main(void) {
lcd_init(16);
while (1) {
	lcd_clear();
	lcd_gotoxy(0,0);
	lcd_putsf("AVR Devp. Board");
	delay_ms(1000);
	lcd_gotoxy(0,1);
	lcd_putsf(“LCD Test Program");
	delay_ms(2000);
	lcd_clear();
	lcd_gotoxy(0,0);
	lcd_putsf("ATmega 32");
	delay_ms(1000);
	lcd_gotoxy(0,1);
	lcd_putsf("Microcontroller");
	delay_ms(2000);

	};
}
```

7-segment display: Introduction

- Seven segment displays are very common for electronic product to display numerical output.
- Many common devices like calculators, lift, watches, electronic weighing scales, ovens etc use them.
- A seven-segment display is so named because it is divided into seven different segments that can be switched on or off.
- It can display digits from 0 to 9 and quite a few characters like A, b, C, ., H, E, e, F, n, o, t, u, y, etc.
- Knowledge about how to interface a seven segment display to a micro controller is very essential in designing embedded systems.

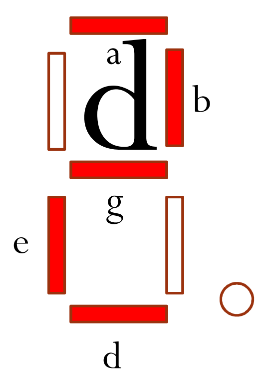
The Pin Out and Picture of a 7-segment Display

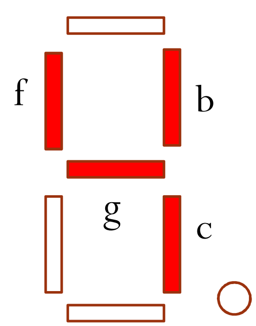


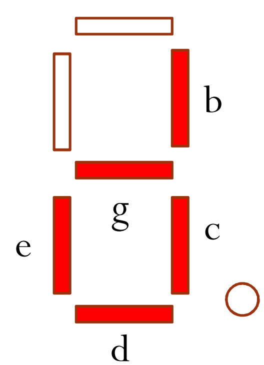


2

4



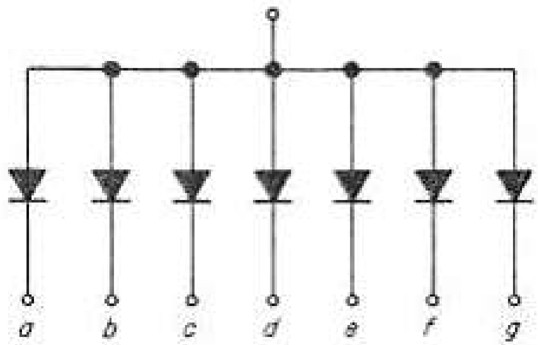




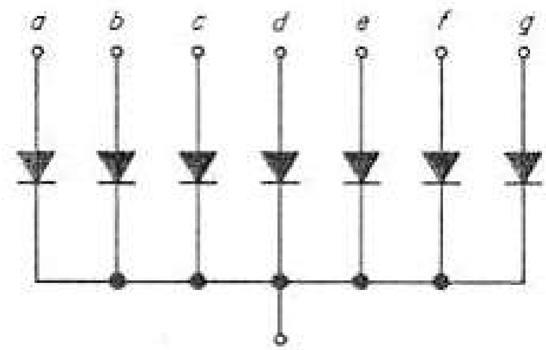
Two types of 7-segment display

- Seven segment displays are of two types, ***common cathode and common anode.***
- In common cathode type , the cathode of all LEDs are tied together to a single terminal which is usually labeled as '**com**'   and the anode of all LEDs are left alone as individual pins labeled as a, b, c, d, e, f, g & h (or dot) .
- In common anode type, the anode of all LEDs are tied together as a single terminal and cathodes are left alone as individual pins.

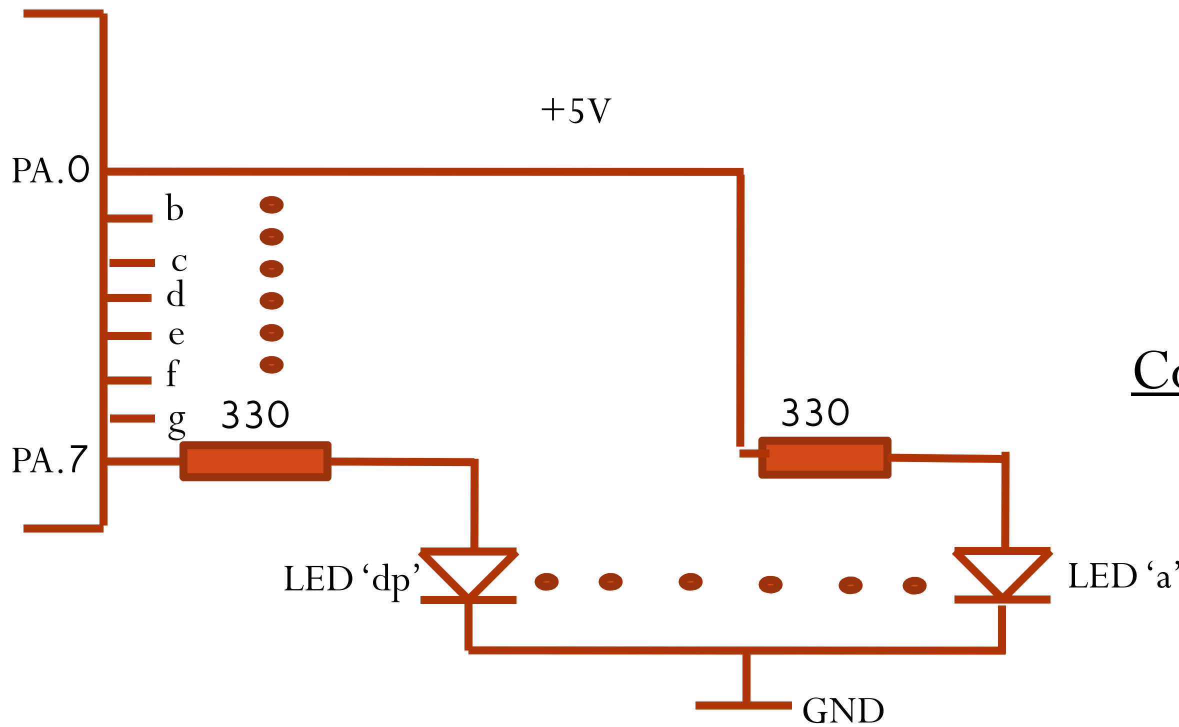
Interfacing 7 segment display



Common Anode

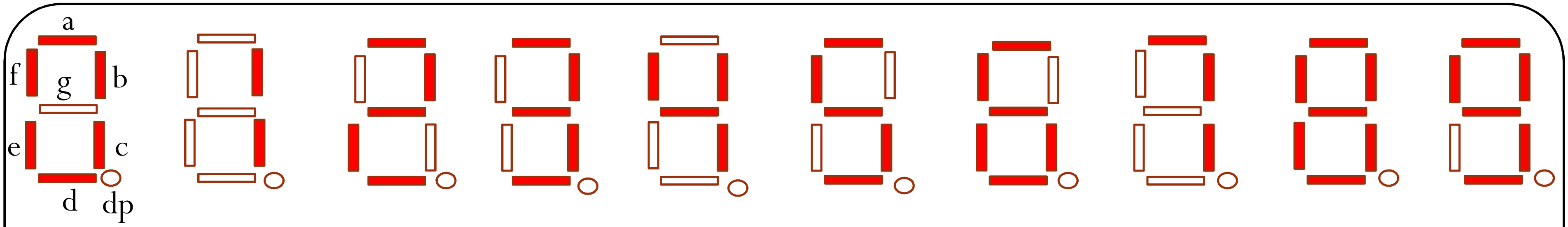


Common Cathode



<u>Common Cathode</u>

7-Segment Display



0 1 2 3 4 5 6 7 8 9

Dp	g	f	e	d	c	b	a	Hex	dig
0	0	1	1	1	1	1	1	3F	0
0	0	0	0	0	1	1	0	06	1
0	1	0	1	1	0	1	1	5B	2
0	1	0	0	1	1	1	1	4F	3
0	1	1	0	0	1	1	0	66	4
0	1	1	0	1	1	0	1	6D	5
0	1	1	1	1	1	0	1	7D	6
0	0	0	0	0	1	1	1	07	7
0	1	1	1	1	1	1	1	7F	8
0	1	1	0	1	1	1	1	6F	9

7-segment display program (Part-I)

```
DDRB=(1<<DDB7) | (1<<DDB6) | (1<<DDB5) | (1<<DDB4) |
   (1<<DDB3) | (1<<DDB2) | (1<<DDB1) | (1<<DDB0);
unsigned int
   cathode[10]={0x3F,0x06,0x5B,0x4F,0x66,0x6D,0x7D,0x07,0x7F,0x6F};
unsigned int i=0;
unsigned int k=0;
while (1)
    {
    if (k==0)
       for(i=1;i<10;i++) {
         PORTB=cathode[i];
         delay_ms(1000);
         if (i==9)
            k=1;
       }
```

7-segment display program (Part-II)

```
else
     for(i=9;i>0;i--) {
      PORTB=cathode[i-1];
      delay_ms(1000);
      if (i==1)
          k=0;
     }

   }
}
```

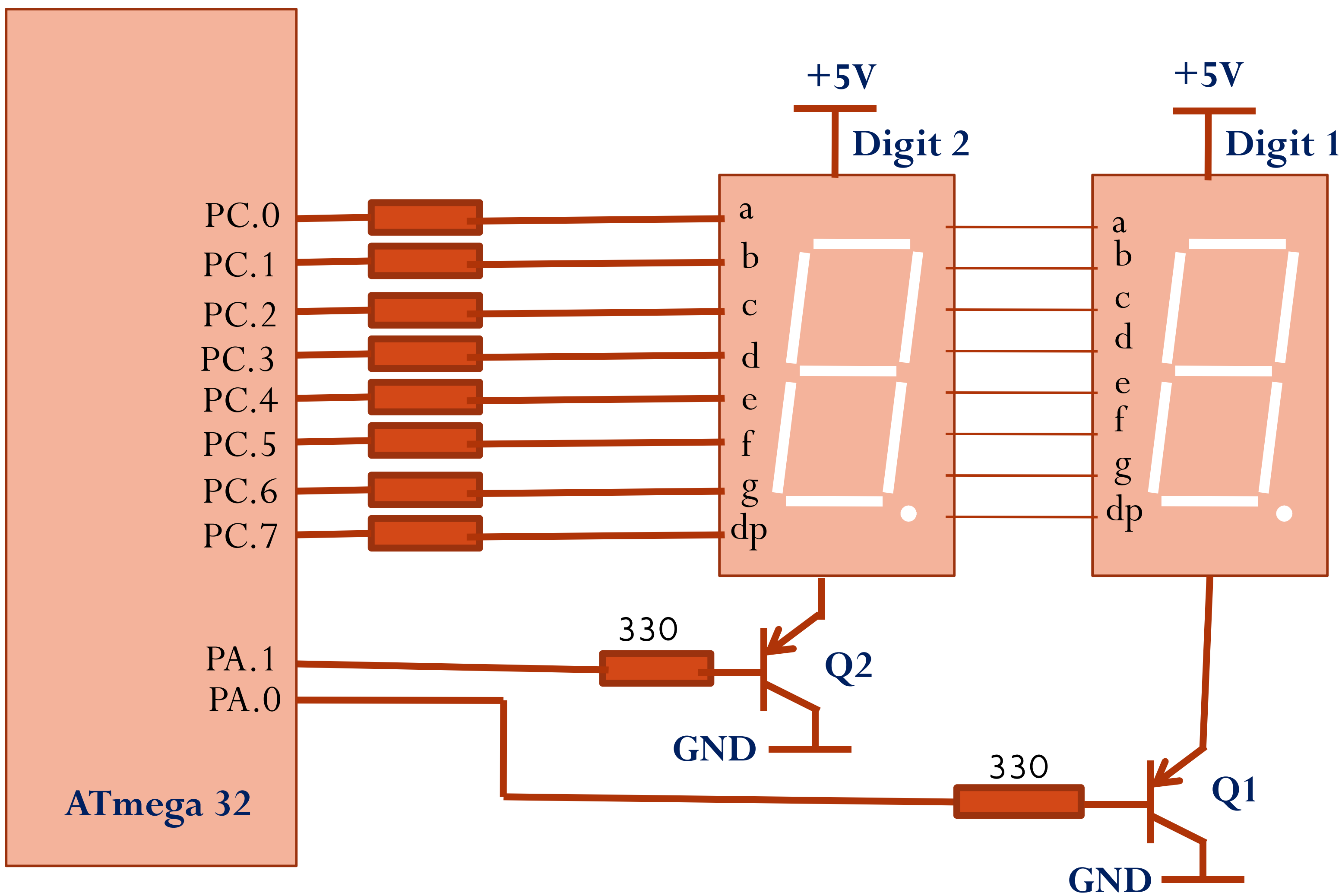
Multiple 7-Segment Display

- Suppose you need a four digit display connected to the AVR ATmega32.
- Each 7 segment display have 8 pins and so a total amount of32 pins are to the connected to the microcontroller and there will be no pin left with the microcontroller for other input output applications.
- More over four displays will be ON always and this consumes a considerable amount of power.
- All these problems associated with the straight forward method can be solved by multiplexing .

Multiplexing 7-Segment Display

- In multiplexing all displays are connected in parallel to one port and only one display is allowed to turn ON at a time, for a short period.
- This cycle is repeated for at a fast rate and due to the persistence of vision of human eye, all digits seems to glow.
- The main advantages of this method are
  - Fewer number of port pins are required .
  - Consumes less power.
  - More number of display units can be interfaced (maximum 24).
- The circuit diagram for multiplexing 4 seven segment displays to the AVR ATmega32 is shown in the next slide.

Connection Diagram of Multiple 7-segment Display with AVR ATmega32



How it works

- Let us see how '16' will be displayed in 2 digit display.
- Initially the first display is only activated by making PA.0 low and then digit drive pattern for "1" is loaded to the Port C.
- This condition is maintained for around 1ms and then PA.0 is made high.
- Then the second display is activated by making PA.1 low and then the digit drive pattern for "6" is loaded to the port C. This will make the second display to show "6". This condition is maintained for another 1ms.
- This cycle is repeated and due to the persistence of vision you will feel it as "16".

Thanks