EE2016 Microprocessor Lab & Theory -July -November 2024

Experiment 3: Hardware Wiring and Assembly Programming using Atmel Atmega(8) AVR

Dr. R. Manivasakan, EE Dept, IITM

1 Aim

This experiment introduces assembly programming and interaction with peripherals in Atmel Atmega8 microcontroller.

1. Wire the microcontroller along with the given peripherals in a breadboard.

2. Program the microcontroller to read the DIP switch values and display it in an LED using assembly programming.

3. Program the microcontroller to perform the addition and multiplication of two four-bit numbers which are read from the DIP switches connected to a port and display the result using LED's connected to another port.

2 Equipments, Hardware Required

To perform this experiment, the following components are required.

1. Atmel AVR (Atmel8L) Chip - 1
2. A breadboard with microprocessor socket
3. 8-bit DIP switches
4. 5 LEDs
5. Capacitors, resistors and wires
6. AVR Programmer (USB-ASP)
7. A windows PC loaded with Microchip Studio 7 and AVR Burn-O-MAT (for burning_asm)

3 Concepts involved:

3.1 Memory Mapped I/O

Memory mapped I/O uses the same address space to address both memory and registers of I/O devices. This implies part of whole address (range) space corresponds to memory and remaining part corresponds to all I/O devices. In MMIO, the CPU sees everything as memory locations. An I/O device responds to an address (issued by CPU), if this address falls within the range of addresses assigned to that device. It is the most common type of I/O interfaces today's microprocessors / microcontrollers. Examples: most or all of microcontrollers, powerPC, MIPS, etc [In this course only MMIO would be discussed].

Port mapped I/O: All peripheral devices (apart from memory) are grouped and is distinguished from memory for interfacing. It works on a dedicated address space for I/O. Each physical port is assigned a code and specific CPU instructions (IN, OUT) are used to access the same. Eg. intel x86 processors.

3.2 Memory Mapped I/O in AVR

Address Space: In AVR processors, the memory address space is shared between program memory (Flash memory) and data memory (SRAM). The address space is divided into several regions, and a portion of this address space is reserved for memory-mapped I/O registers.

I/O Registers: AVR microcontrollers have a set of special registers known as I/O registers. These registers are used for controlling and communicating with peripheral devices like GPIO pins, timers, UART (serial communication), SPI, and more. Each I/O register is associated with a specific peripheral or function and is located at a specific memory address within the memory-mapped I/O space.

Direct Access: Program instructions can directly access these I/O registers by reading from or writing to the memory addresses associated with the registers. This means that to control a peripheral or read its status, you can use regular load (LDS) and store (STS) instructions. For example, to set a GPIO pin as an output in an AVR microcontroller, you would write to the appropriate I/O register, which sets the corresponding bits in that register.

Simplified I/O Handling: Memory-mapped I/O simplifies the interaction with peripheral devices. Instead of using complex I/O instructions, such as IN and OUT in AVR assembly language, you can use simple load and store instructions to read from and write to peripheral registers. This makes it more intuitive and efficient to work with hardware.

In this experiment, we would be using IN and OUT instructions.

4 Technical Details Specific to AVR Peripheral Interfacing

In practice, we have following requirements:

1. **Inside** the AVR chip:
   (a) the flexibility to configure the hardware pin or port as input or output on the fly
   (b) if hardware pin or port is configured as output (or input), a register to store the data to be sent out (or a register to store the data which is read from the hardware pins, into the inside of the microprocessor)
2. **Outside** the AVR chip: in the case of hardware pin being configured as input, and in the case of the feeding digital circuitry, being in float state, pull-up / pull down resistor is used to mitigate the system.
3. **AVR** assembly programming

4.1 Inside the AVR chip

The main technical challenge is to interface the external device through hardware pins of AVR chip. Interfacing a peripheral has **two steps**: (i) addressing, meaning picking the desired device out of n such devices and (ii) after picking up the peripheral, "hand-shaking" to either get tasks done by the (output) peripheral or to receive the data or other inputs from the (input) peripheral.

Consider now three registers associated with a physical port of 8 pins in AVR. See Figure 1, wherein data bus and control bus are not shown explicitly. Out of these three, two are data registers, one for input and another for output. The third one is for control (or configuration). These registers are analogous to the registers in the PIC (Peripheral Interface Controller). Recall the memory mapped I/O concept described in the class. Hence, a small address range is reserved for picking up these registers corresponding to this particular xth peripheral. This solves the addressing (**first**) problem.

The **second part** is solved thus: To start with, we need to decide, whether we would like to configure xth port as input or output. That is achieved by DDRx register. If 'nth' pin is to set as output, then assign DDRx.n = 0b1. Also, you also would like to receive input from the same physical nth pin of xth hardware port (pin), some other time. In that case, assign DDRx.n = 0b0. Once configured, output destined data is held by PORTx (or single nth pin) and input data read from peripheral is written into PINx (or single nth PIN thereof). The schematic is given below (Read PORTx for PORTox and PINx for PORTix).

Rest of description given below, are the expanded version of the above with full technical details.

4.1.1 Configuration of Ports

The architecture of AVR microcontrollers is register-based: information in the microcontroller such as the program memory, state of input pins and state of output pins, their configuration are stored in corresponding I/O registers. There are a total of 32, 8-bit general purpose registers (GPRs). Atmega8 has 23 I/O pins. These pins are grouped under what are known as ports. A port can be visualized as a gate with a specific address (used by CU), between the CPU and the external world. The CPU uses these ports to read input from and write output into them. The Atmega8 microcontroller has 3 hardware ports: PortB, PortC, and PortD. Each of these ports is associated with 3 registers - DDRx, PORTx and PINx which set, respectively, the direction, input and output functionality of the ports. Each bit in these registers configures a pin of a port. Bit0 of a register is mapped to Pin0 of a particular port, Bit1 to Pin1, Bit2 to Pin2 and so on. Each register is explained in detail below.

4.1.1.1 Register DDRx

DDR stands for Data Direction Register and x indicates a port index. As the name suggests, this register is used to set the direction of port pins to either input or output. For input, we set to 0 while for output, we set to 1. For instance, let us consider PortB. To set this port as input or output, we need to initialize DDRB. Each bit in DDRB corresponds to the respective pin in PortB. Suppose we write DDRB = 0xFF, then all bits in PortB are congured to Output. Similarly, DDRB=0x00 configures the all the bits in the port to be Input.

4.1.1.2 Register PORTx

Given a port x (either all bits or nth bit) is set as output port (by configuring the DDRx suitably as above), the PORTx pins are loaded with bits, which are meant for output. These bits, appear on the hardware pins of the output port x. (This means in active high logic, if PORTx.n is 1, then the hardware nth pin of port x would be high - 1.8 V etc).

In the Figure 1, the buffer gates have o/p enable pin making it as a tri-state device. The inverter in the circuit, ensures whether the targetted hardware pin can be either input or output at a time. (Addressing ensures the selection of this port x, out of the 3 such ports in AVR).

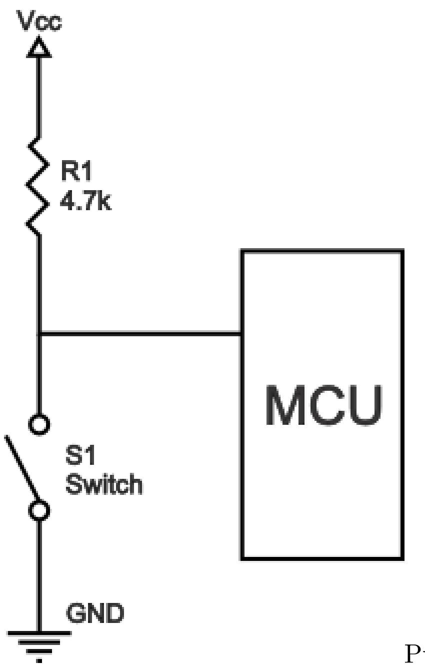
4.1.1.3 Register PINx

The functionality of a PINx register is straightforward: it reads data from the port pin. As mentioned above, the hardware pins have to be configured as input, by setting the corresponding pin of DDRx to be 0.

4.2 Outside the AVR chip

The following discusses the problem, one encounters when feeding the hardware input, externally, in the specific case of hardware AVR pin is configured as input. The circuitry outside the microcontroller, feeding the input to this port x, at some times, be floating. Pull up resistor, or pull down resistors are used to handle the floating situations. This is described next.

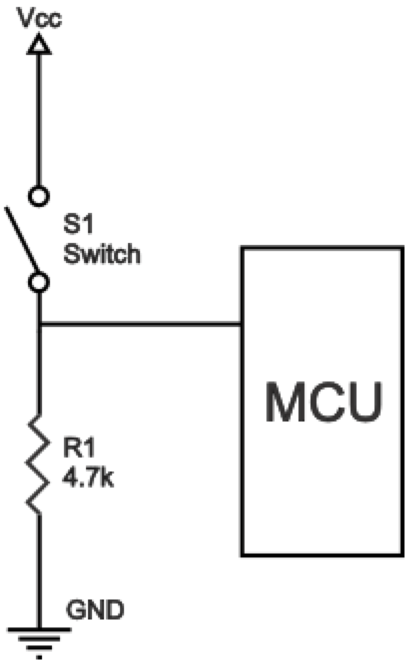
Pull-up resistors: Pull-up resistors are resistors used in logic circuits to ensure a well-defined logical level at a pin under all conditions. As a reminder, digital logic circuits have three logic states: high, low and floating (or high impedance). The high-impedance (input) state occurs, when the digital circuitry, (which feeds the input to the considered microcontroller), decides not to connect any of output of output stage gates - it rarely occurs. In that case, the input hardware pin floats, which is undesirable. Consider the case in which the other end of the external input is connected to ground. Without the pull-up resistor, the AVR MCU's input would be floating when the switch is open and pulled down to a logical low only when the switch is closed.



Pull-up resistor

Now, connect the pull-up resistor from input to supply Vcc. When the feeding external digital circuitry floats, the input hardware pin is high, which is a valid input state (other one also valid).

Pull-down resistors: Situations are similar to the above, except that the other end of the digital circuitry is connected to high (instead of ground).



Pull-down resister is used to make the input completely legal irrespective of the state of feeding digital circuitry.

4.3 Assembly Programming

AVR microarchitecture supports special instructions to interact with peripheral devices. Two such instructions are IN and OUT instruction.

4.3.1 OUT

This instruction stores data from register Rr in the Register File to I/O Space (Ports, Timers, Configuration Registers, etc.). For example, "OUT A, Rx" writes the value stored in Rx to I/O register named A. [The I/O register here could refer to the PORTx (output) register corresponding to the xth peripheral hardware port. This OP code is also used to configure the hardware port by writing into the corresponding DDRB register].

4.3.2 IN

This instruction loads data from the I/O Space (Ports, Timers, Configuration Registers, etc.) into register Rd in the Register File. For example, "IN Rd,A" loads the value stored in I/O register named A to register Rd. [The I/O register here could refer to the PINx (input) register corresponding to the xth peripheral hardware port].

4.3.3 RJMP

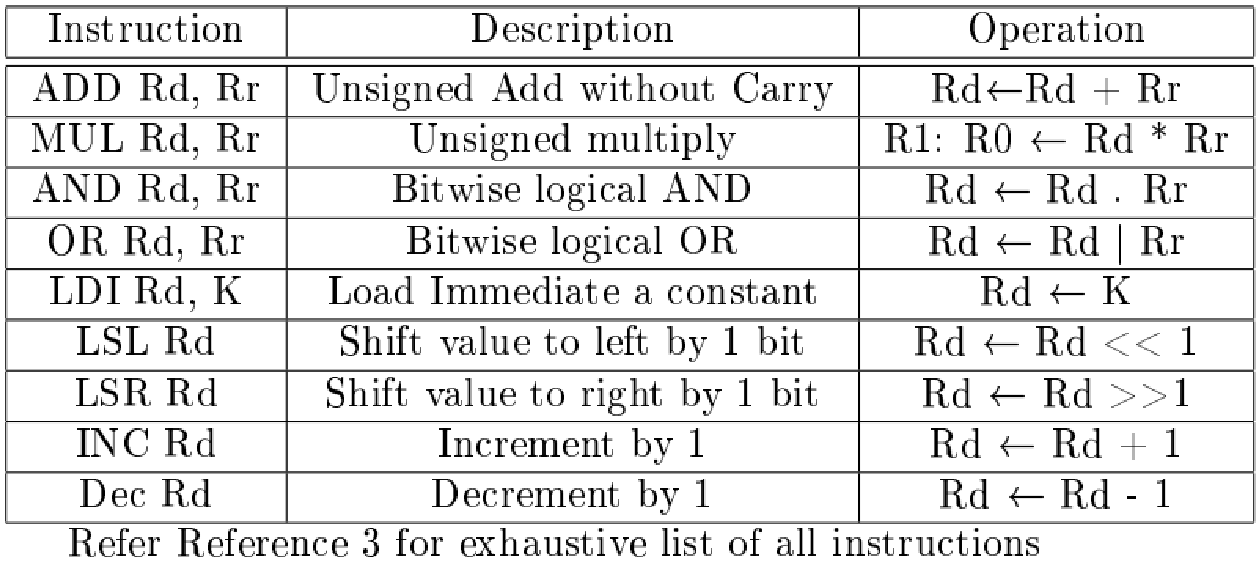
This instruction is used to jump the control of the program to a particular instruction. For example, the following snippet moves the program counter to the instruction LDI after execution of MOV.

```
again:   LDI R16, 0x23

        MOV R0,R1

        RJMP again
```

4.3.4 Other instructions



Instruction	Description	Operation
ADD Rd, Rr	Unsigned Add without Carry	Rd←Rd + Rr
MUL Rd, Rr	Unsigned multiply	R1: R0 ← Rd * Rr
AND Rd, Rr	Bitwise logical AND	Rd ← Rd . Rr
OR Rd, Rr	Bitwise logical OR	Rd ← Rd | Rr
LDI Rd, K	Load Immediate a constant	Rd ← K
LSL Rd	Shift value to left by 1 bit	Rd ← Rd << 1
LSR Rd	Shift value to right by 1 bit	Rd ← Rd >>1
INC Rd	Increment by 1	Rd ← Rd + 1
Dec Rd	Decrement by 1	Rd ← Rd - 1

Refer Reference 3 for exhaustive list of all instructions

4.4 Sample Program

The following program can be used to read from 8 input pins in PORTB and write to the corresponding numbered pins in PORTD.

```
#include " m8def.inc "
LDI R16 , 0xFF ; All bits set as 1 to make port as output
OUT DDRD, R16 ;DDRD=Data Direction Register for PORT D

LDI R16 , 0 x00 ; All bits set as 0 to make port as input
OUT DDRB, R16 ;DDRB=Data Direction Register for PORT B
```

`IN R16 , PINB` ; Store the input of PORT B to register R16 OUT PORTD, R16 ; Set LED connected to PDx as input of PBx

5 Procedure

AVR-USP programmer's purpose is to (read and) write data to the AVR chip's memory and flash. In particular, the programmer does this using concept of ISP (in-System Programming) by which serially both the .HEX file containing the instructions plus address (in the program ROM) are sent to Flash of the target device, thus burning the flash with program. The software on the PC which implements this, is Burn-o-Mat.

5.1 Programming Environment

1. Open Microchip Studio 7 software installed in windows.

2. Create a new project with template as Assembler and select AVR Assembler Project

3. Set device family as megaAVR, 8-bit and select the device ATmega8. You should find a new assembler project being created.

4. Write the assembly program for reading from DIP switches and writing to LEDs

5. Under the Build menu, select Build Solution to compile and build your project. If the program is correct, you will find a .HEX file being created in Debug folder.

6. Connect the AVR-USP programmer to the target AVR chip as given in next subsection 5.2. Connect a DIP switch and a pull-up resistor.

7. Open the Burn-o-MAT software, select the AVR-USP programmer, select the target AVR chip (on the breadboard), select the target .HEX file and hit “Burn” as explained in subsection 5.3

8. Target AVR would start executing your program

5.2 Connections

1. Wire the Atmega8 processor in the breadboard with AVR Programmer as shown in Figure 1.

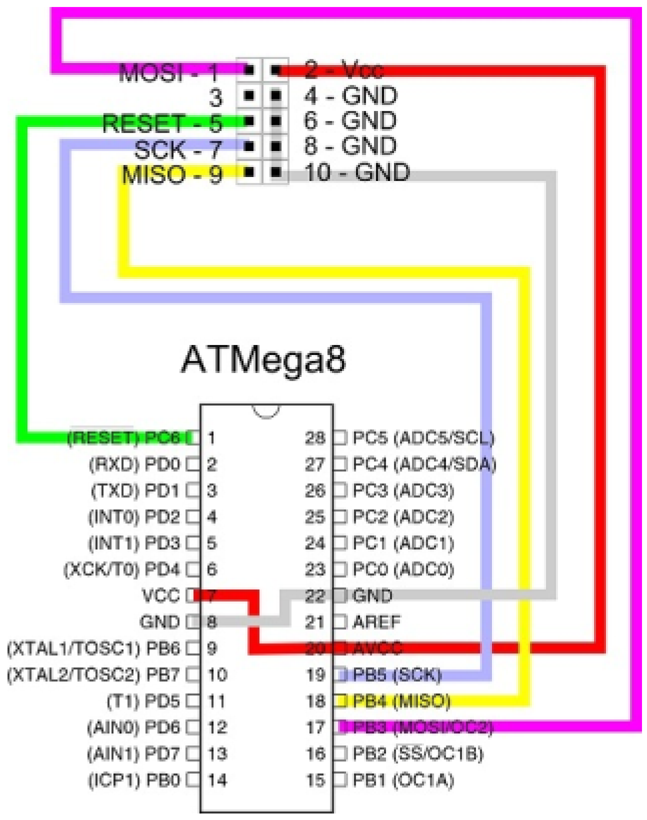


Figure 1: Microcontroller to In-System Programmer Connections

2. Connect any input port, say PB0 to one end of DIP switch and other end to ground. Connect a 10 $k\Omega$ pull-up resistor to PB0 as shown in Figure 2.

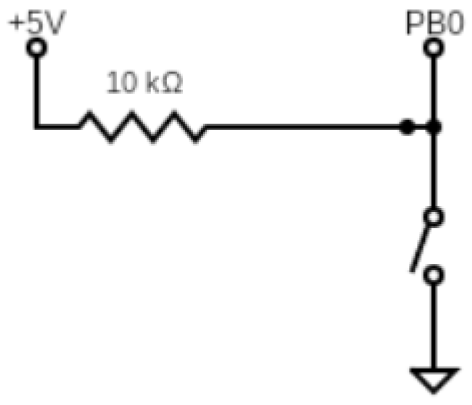


Figure 2: Connection to input port 5

3. Connect an LED between output port, say PD0 and ground preferably through a 300 Ω resistor.

5.3 Burning the program

1. Connect the USB cable to the CPU.

2. Open AVR Burn-O-MAT software and select AVR type as ATmega8

3. Under the ash section, select the previously generated .HEX le. Click on write to write the program to the microcontroller.

4. Demonstrate the working model to the TAs and verify the results for the different programs.

6 Problems:

1. Write an AVR assembly program to blink an LED with pulse width (during ON) of 0.5 sec and pulse repetition rate of 1 cycle /sec

2. Write an AVR assembly program to control an LED using a push button switch (as long as button is pushed, LED should be 'ON').

3. 4-bit addition of two unsigned nibbles from an 8-bit DIP input switch (set by TAs) and display the result obtained in LEDs.

7 References

1. ATmega8(L) - Summary Datasheet
2. ATmega8(L) - Complete Datasheet
3. AVR instruction set manual
4. Muhammad Ali Mazidi, Sarmad Naimi and Sepehr Naimi, Chapter4, AVR I/O Port Programming, the AVR microcontroller and embedded system using assembly and 'C'