CHAPTER 2

AVR ARCHITECTURE AND ASSEMBLY LANGUAGE PROGRAMMING

OBJECTIVES

Upon completion of this chapter, you will be able to:

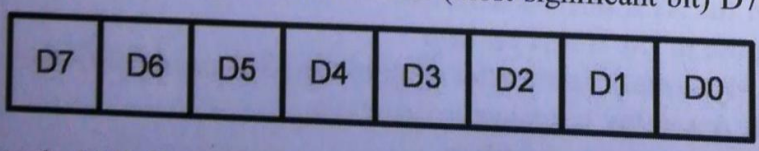
- **List the registers of the AVR microcontroller**
- **Examine the data memory of the AVR microcontroller**
- **Perform simple operations, such as ADD and load, and access internal RAM memory in the AVR microcontroller**
- **Explain the purpose of the status register**
- **Discuss data RAM memory space allocation in the AVR microcontroller**
- **Code simple AVR Assembly language instructions**
- **Describe AVR data types and directives**
- **Assemble and run a AVR program using Atmel Studio**
- **Describe the sequence of events that occur upon AVR power-up**
- **Examine programs in AVR ROM code**
- **Detail the execution of AVR Assembly language instructions**
- **Understand the RISC and Harvard architectures of the AVR microcontroller**
- **Examine the AVR's registers and data RAM using the Atmel Studio simulator**

CPUs use 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 Section 2.1 we look at the general purpose registers (GPRs) of the AVR. We demonstrate the use of GPRs with simple instructions such as LDI and ADD. Allocation of RAM memory and the direct addressing mode of the AVR are discussed in Sections 2.2 and 2.3. In Section 2.4 we discuss the status register's flag bits and how they are affected by arithmetic instructions. In Section 2.5 we look at some widely used Assembly language directives, pseudocode, and data types related to the AVR. In Section 2.6 we examine Assembly language and machine language programming and define terms such as mnemonics, opcode, operand, and so on. The process of assembling and creating a ready-to-run program for the AVR is discussed in Section 2.7. Step-by-step execution of an AVR program and the role of the program counter are examined in Section 2.8. The merits of RISC architecture are examined in Section 2.9. Section 2.10 discusses the Atmel Studio.

SECTION 2.1: 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.

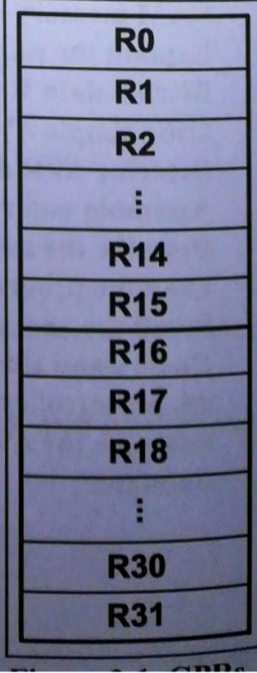
AVR microcontrollers have many registers for arithmetic and logic operations. In the CPU, registers are used to store information temporarily. That information could be a byte of data to be processed, or an address pointing to the data to be fetched. The vast majority of AVR registers are 8-bit registers. 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) D0. 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 R0–R31 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

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 R16 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 0x25 (25 in hex).

```
LDI R20,0x25              ;load R20 with 0x25 (R20 = 0x25)
```

The following instruction loads the R31 register with the value 0x87 (87 in hex).

```
LDI R31,0x87              ;load 0x87 into R31  (R31 = 0x87)
```

The following instruction loads R25 with the value 0x15 (15 in hex and 21 in decimal).

```
LDI R25,0x79              ;load 0x79 into R25 (R25 = 0x79)
```

Note: We cannot load values into registers R0 to R15 using the LDI instruction. For example, the following instruction is not valid:

```
LDI R5,0x99               ;invalid instruction
```

Notice the position of the source and destination operands. As you can see, the LDI loads the right operand into the left operand. In other words, the destination comes first.

To write a comment in Assembly language we use ';'. It is the same as '//' in C language, which causes the remainder of the line of code to be ignored. For instance, in the above examples the expressions mentioned after ';' just explain the functionality of the instructions to you, and do not have any effects on the execution of the instructions.

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

1. 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`", R16 is loaded with 50 in decimal, whereas in "`LDI R16,0x50`", R16 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 R16 = 0x05; that is, R16 = 00000101 in binary.
3. Moving a value larger than 255 (FF in hex) into the GPRs will cause an error.

```
LDI   R17, 0x7F2  ;ILLEGAL $7F2 > 8 bits ($FF)
```

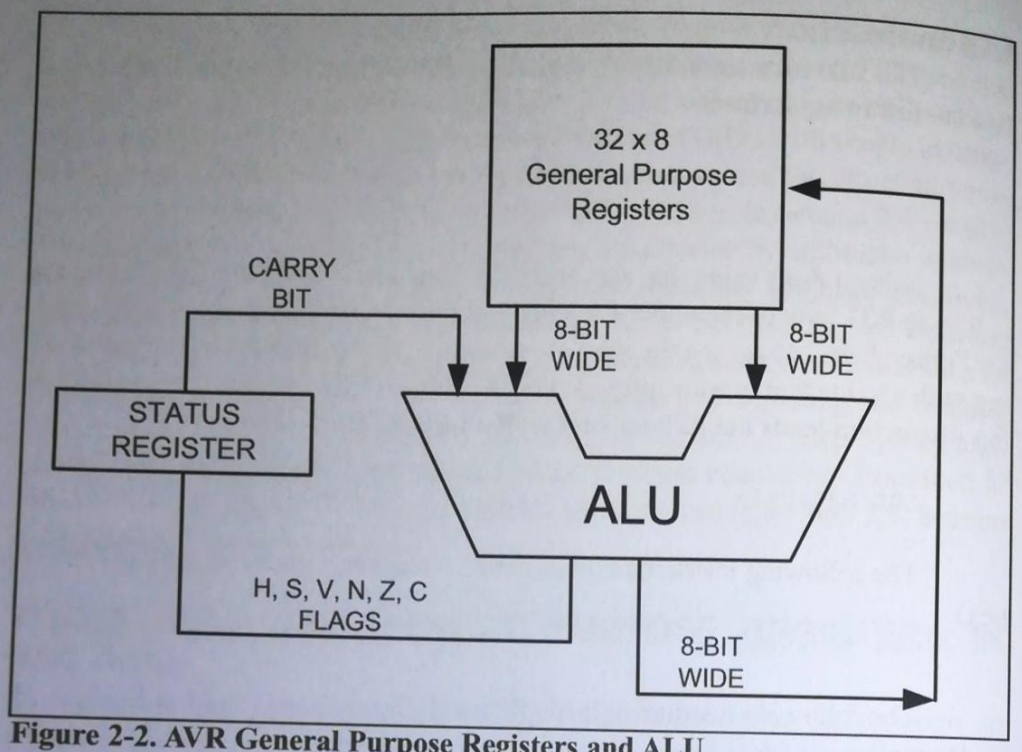


Figure 2-2. AVR General Purpose Registers and ALU

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 of Rr 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)
```

Executing the above lines results in R16 = 0x59 (0x25 + 0x34 = 0x59)

Figure 2-2 shows the general purpose registers (GPRs) and the ALU in AVR. The affect of arithmetic and logic operations on the status register will be discussed in Section 2.4.

Review Questions

1. Write an instruction to copy the value 0x34 into the R29 register.
2. Write instructions to add the values 0x16 and 0xCD. Place the result in the R19 register.
3. True or false. No value can be moved directly into the GPRs.
4. What is the largest hex value that can be moved into an 8-bit register? What is the decimal equivalent of that hex value?
5. The vast majority of registers in the AVR are _____-bit.

SECTION 2.2: THE 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. We will examine the code memory space in Section 2.8. In this section, we will discuss the data memory space. The data memory is composed of three parts: GPRs (general purpose registers), I/O memory, and internal data SRAM. See Figure 2-3.

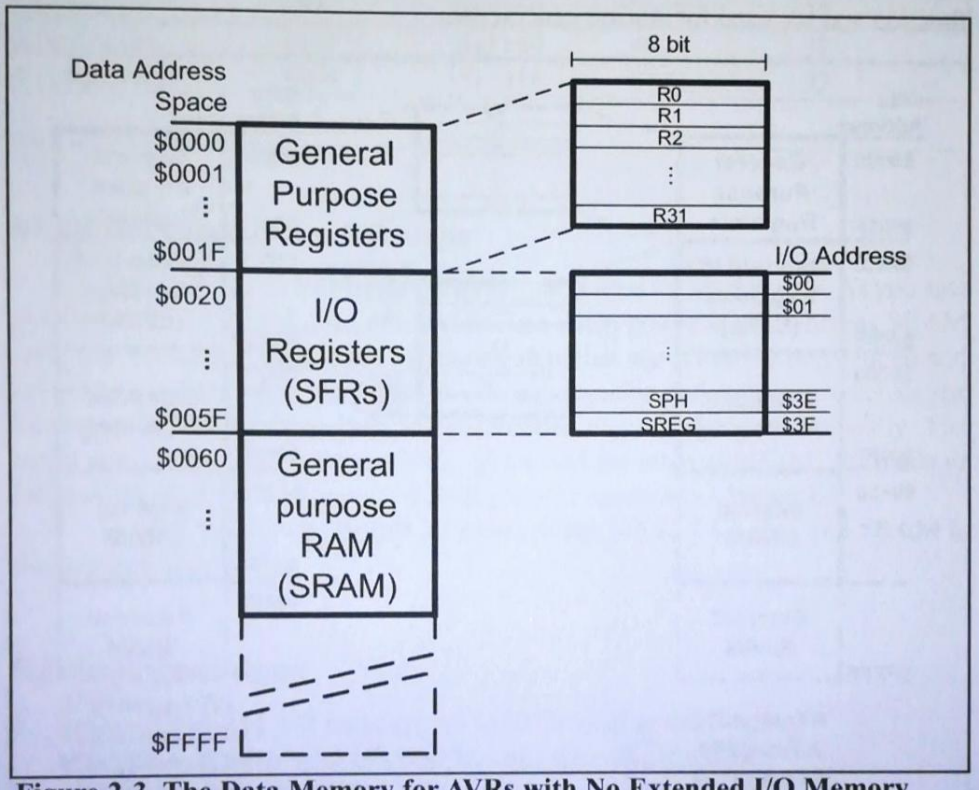


Figure 2-3. The Data Memory for AVRs with No Extended I/O Memory

GPRs (general purpose registers)

As we discussed in the last section, the GPRs use 32 bytes of data memory space. They always take the address location $00–$1F in the data memory space, regardless of the AVR chip number. See Figure 2-3.

I/O 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 I/O 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 AVRs have at least 64 bytes of I/O memory locations. This 64-byte section is called *standard I/O memory*. In AVRs with more peripherals (e.g., ATmega328, ATmega128, 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 I/O 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.

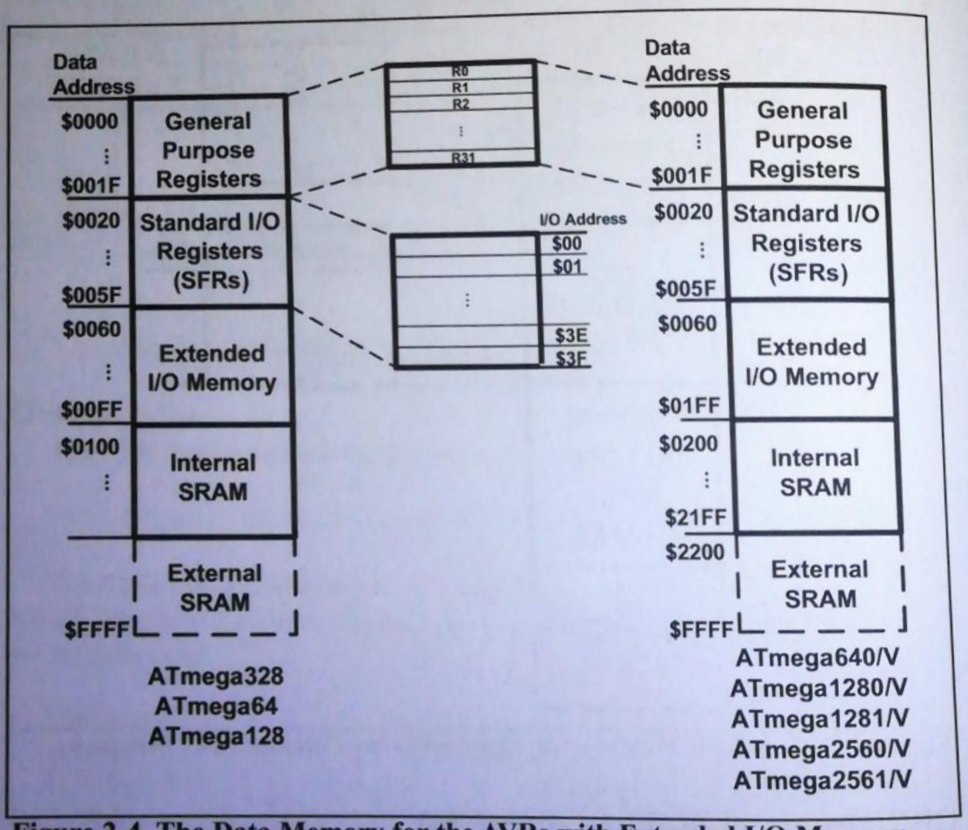


Figure 2-4. The Data Memory for the AVRs with Extended I/O Memory

Internal data SRAM

Internal data SRAM is widely used for storing data and parameters by AVR programmers and C compilers. Generally, this is called *scratch pad*. Each location of the SRAM can be accessed directly by its address. We will use these locations in future chapters to store data brought into the CPU via I/O and serial ports. Each location is 8 bits wide and can be used to store any data we want as long as it is 8-bit. The size of SRAM can vary from chip to chip, even among members of the same family. See Table 2-1 for a comparison of the data memories of various AVR chips. Also, see Figure 2-4.

By adding the sizes of GPR, SFRs (I/O registers), and SRAMs we get the data memory size. See Table 2-1.

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
ATmega328	2304		64+160		2048	32
ATmega128	4352		64+160		4096	32
ATmega2560	8704		64+416		8192	32

SRAM vs. EEPROM in AVR chips

The AVR has an EEPROM memory that is used for storing data. As you saw in Chapter 0, EEPROM does not lose its data when power is off, whereas SRAM does. So, the EEPROM is used for storing data that should rarely be changed and should not be lost when the power is off (e.g., options and settings); whereas the SRAM is used for storing data and parameters that are changed frequently. The three parts of the data memory (GPRs, SFRs, and the internal SRAM) are made of SRAM. The EEPROM memory of AVR chips is covered in Chapter 6.

In AVR datasheets, EEPROM refers to the EEPROM's size, and SRAM is the internal SRAM size.

Review Questions

1. True or false. The I/O registers are used for storing data.
2. The GPRs together with I/O registers and SRAM are called_____.
3. The I/O registers in AVR are _____-bit.
4. The data memory space in AVR is divided into _____ parts.
5. The data memory space in AVR can be a maximum of _____ bytes.
6. The standard I/O memory space in AVR is _____ bytes.

SECTION 2.3: USING INSTRUCTIONS WITH THE DATA MEMORY

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 in Section 2.1. The AVR 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 AVR Assembly language programming.

LDS instruction (LoaD direct from data Space)

```
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 location in the data memory can be in any part of the data space; it can be one of the I/O registers, a location in the internal SRAM, or a GPR. For example, the "`LDS R20,0x1`" instruction will copy the contents of location 1 (in hex) into R20. As you can see in Figure 2-4, location 1 of the data memory is in the GPR part, and it is the address of R1. So, the instruction copies R1 to R20.

The following instruction loads R5 with the contents of location 0x300. As you can see in Figure 2-4, 0x300 is located in the internal SRAM:

```
LDS R5,0x300 ;load R5 with the contents of location $300
```

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
```

You can see the execution of "`LDS R0,0x300`" and "`LDS R1,0x302`" instructions in Figure 2-5. Figure 2-6 shows the contents of R0, R1 and locations 300 and 302 of data memory before and after the execution of each of the instructions, assuming that locations $300 and $302 contain a and β, respectively.

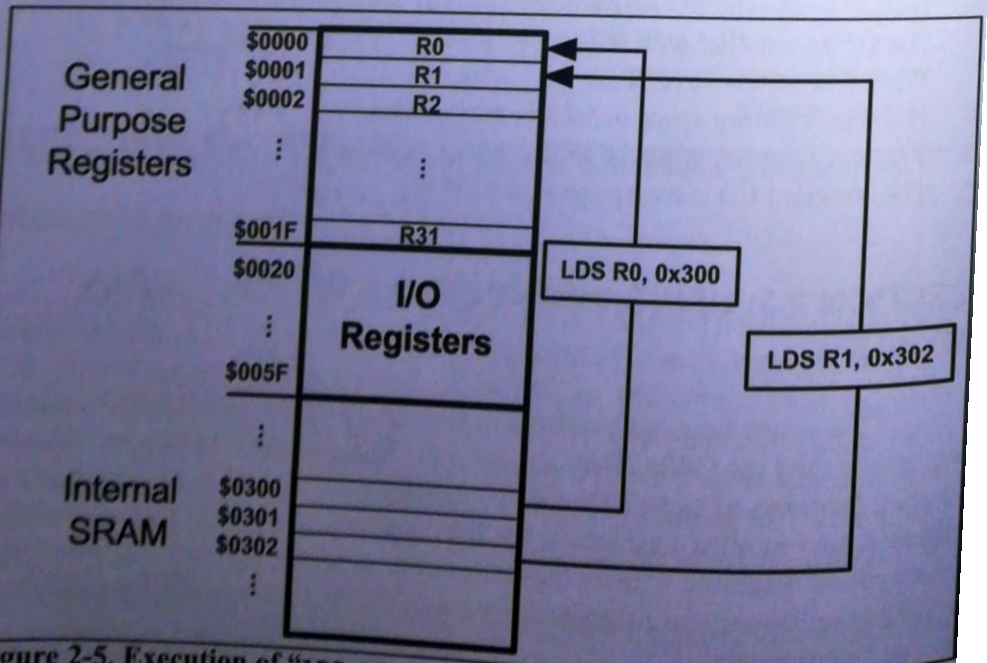


Figure 2-5. Execution of "`LDS R0,0x300`" and "`LDS R1,0x302`" Instructions

	R0	R1	Loc $300	Loc $302
Before LDS R0,0x300	?	?	α	β
After LDS R0,0x300	α	?	α	β
After LDS R1,0x302	α	β	α	β
After ADD R0, R1	α + β	β	α	β

Figure 2-6. The Contents of R0, R1, and Locations $300 and $302

STS instruction (STore direct to data Space)

```
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. For example, the "`STS 0x1,R10`" instruction will copy the contents of R10 into location 1. As you can see in Figure 2-4, location 1 of the data memory is in the GPR part, and it is the address of R1. So, the instruction copies R10 to R1.

The following instruction stores the contents of R25 to location 0x230. As you can see in Figure 2-4, 0x230 is located in the internal SRAM:

```
STS 0x230, R25    ;store R25 to data space location 0x230
```

The following program first loads the R16 register with value 0x55, then moves this value around to I/O registers of ports B, C, and D. As shown in Figure 2-7, the addresses of PORTB, PORTC, and PORTD are 0x25, 0x28, and 0x2B, respectively:

```
LDI   R16, 0x55   ;R16 = 55 (in hex)
STS   0x25, R16   ;copy R16 to Port B (PORTB = 0x55)
STS   0x28, R16   ;copy R16 to Port C (PORTC = 0x55)
STS   0x2B, R16   ;copy R16 to Port D (PORTD = 0x55)
```

PORTB, PORTC, and PORTD are part of the I/O memory. They can be connected to the I/O pins of the AVR microcontroller as we will see in Chapter 4.

We can also store the contents of a GPR into any location in the SRAM region of the data space. The following program will put 0x99 into locations 0x200–0x203 of the SRAM region in the data memory:

```
LDI   R20, 0x99   ;R20 = 0x99
STS   0x200, R20  ;store R20 in loc 0x200
STS   0x201, R20  ;store R20 in loc 0x201
STS   0x202, R20
STS   0x203, R20 ;see the Mem. contents->
```

Address	Data
$200	0x99
$201	0x99
$202	0x99
$203	0x99

Notice that you cannot copy (store) an immediate value directly into the SRAM location in the AVR. This must be done via the GPRs.

The following program adds the contents of location 0x220 to location 0x221, and stores the result in location 0x221:

```
LDS   R30, 0x220  ;load R30 with the contents of location 0x220
LDS   R31, 0x221  ;load R31 with the contents of location 0x221
ADD   R31, R30    ;add R30 to R31
STS   0x221, R31  ;store R31 to data space location 0x221
```

See Examples 2-1 and 2-2.

IN instruction (IN from I/O location)

```
IN    Rd, A ;load an I/O 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 I/O register. For example, the "`IN R20,0x16`" instruction will copy the contents of location 16 (in hex) of the I/O memory into R20. As you can see in Figure 2-7, 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 I/O memory; this address is called the *I/O address*. See Figure 2-3. You see the list of I/O registers in Figure 2-7.

Address		Name
Mem.	I/O	
$20	$00	-
$21	$01	-
$22	$02	-
$23	$03	PINB
$24	$04	DDRB
$25	$05	PORTB
$26	$06	PINC
$27	$07	DDRC
$28	$08	PORTC
$29	$09	PIND
$2A	$0A	DDRD
$2B	$0B	PORTD
$2C	$0C	-
$2D	$0D	-
$2E	$0E	-
$2F	$0F	-
$30	$10	-
$31	$11	-
$32	$12	-
$33	$13	-
$34	$14	-
$35	$15	TIFR0

Address		Name
Mem.	I/O	
$36	$16	TIFR1
$37	$17	TIFR2
$38	$18	-
$39	$19	-
$3A	$1A	-
$3B	$1B	PCIFR
$3C	$1C	EIFR
$3D	$1D	EIMSK
$3E	$1E	GPIOR0
$3F	$1F	EECR
$40	$20	EEDR
$41	$21	EEARL
$42	$22	EEARH
$43	$23	GTCCR
$44	$24	TCCR0A
$45	$25	TCCR0B
$46	$26	TCNT0
$47	$27	OCR0A
$48	$28	OCR0B
$49	$29	-
$4A	$2A	GPIOR1
$4A	$2A	GPIOR2

Address		Name
Mem.	I/O	
$4C	$2C	SPCR0
$4D	$2D	SPSR0
$4E	$2E	SPDR0
$4F	$2F	-
$50	$30	ACSR
$51	$31	DWDR
$52	$32	-
$53	$33	SMCR
$54	$34	MCUSR
$55	$35	MCUCR
$56	$36	-
$57	$37	SPMCSR
$58	$38	-
$59	$39	-
$5A	$3A	-
$5B	$3B	-
$5C	$3C	-
$5D	$3D	SPL
$5E	$3E	SPH
$5F	$3F	SREG

Note: Although Memory Address $20-$5F is set aside for I/O Registers (SFR) we can access them as I/O locations with addresses starting at $00.

Figure 2-7. I/O Registers of the ATmega328 and Their Data Memory Address Locations

State the contents of RAM locations $212 to $216 after the following program is executed:

```
LDI   R16, 0x99   ;load R16 with value 0x99
STS   0x212, R16
LDI   R16, 0x85   ;load R16 with value 0x85
STS   0x213, R16
LDI   R16, 0x3F   ;load R16 with value 0x3F
STS   0x214, R16
LDI   R16, 0x63   ;load R16 with value 0x63
STS   0x215, R16
LDI   R16, 0x12   ;load R16 with value 0x12
STS   0x216, R16
```

Solution:

After the execution of `STS 0x212,R16` data memory location $212 has value 0x99;
after the execution of `STS 0x213,R16` data memory location $213 has value 0x85;
after the execution of `STS 0x214,R16` data memory location $214 has value 0x3F;
after the execution of `STS 0x215,R16` data memory location $215 has value 0x63;
and so on, as shown in the chart.

Address	Data
$212	0x99
$213	0x85
$214	0x3F
$215	0x63
$216	0x12

Example 2-2

State the contents of R20, R21, and data memory location 0x120 after the following program:

```
LDI   R20, 5      ;load R20 with 5
LDI   R21, 2      ;load R21 with 2
ADD   R20, R21    ;add R21 to R20
ADD   R20, R21    ;add R21 to R20
STS   0x120, R20  ;store in location 0x120 the contents of R20
```

Solution:

The program loads R20 with value 5. Then it loads R21 with value 2. Then it adds the R21 register to R20 twice. At the end, it stores the result in location 0x120 of data memory.

Location	Data
R20	5
R21	
0x120	

After LDI R20, 5

Location	Data
R20	5
R21	2
0x120	

After LDI R21, 2

Location	Data
R20	7
R21	2
0x120	

After ADD R20, R21

Location	Data
R20	9
R21	2
0x120	

After ADD R20, R21

Location	Data
R20	9
R21	2
0x120	9

After STS 0x120, R20

In the IN instruction, the I/O registers are referred to by their I/O addresses. For example, the "`IN R20,0x03`" instruction will copy the contents of location $03 of the I/O memory (whose data memory address is 0x23) into R20. As shown in Figure 2-7, I/O address 0x03 belongs to PINB, so the instruction copies the contents of PINB to R20.

The following instruction loads R19 with the contents of location 0x09 of the I/O memory:

```
IN R19,0x09          ;load R19 with location $9 (R19 = PIND)
```

To work with the I/O registers more easily, we can use their names instead of their I/O addresses. For example, the following instruction loads R19 with the contents of PIND:

```
IN R19,PIND          ;load R19 with PIND
```

The details of I/O ports are discussed in Chapter 4.

The following program adds the contents of PIND to PINB, and stores the result in location 0x300 of the data memory:

```
IN    R1,PIND      ;load R1 with PIND
IN    R2,PINB      ;load R2 with PINB
ADD   R1, R2       ;R1 = R1 + R2
STS   0x300, R1    ;store R1 to data space location $300
```

IN vs. LDS

As we mentioned earlier, we can use the LDS instruction to copy the contents of a memory location to a GPR. This means that we can load an I/O register into a GPR, using the LDS instruction. So, what is the advantage of using the IN instruction for reading the contents of I/O registers over using the LDS instruction? The IN instruction has the following advantages:

1. The CPU executes the IN instruction faster than LDS. As you will see in Chapter 3, the IN instruction lasts 1 machine cycle, whereas LDS lasts 2 machine cycles.
2. The IN is a 2-byte instruction, whereas LDS is a 4-byte instruction. This means that the IN instruction occupies less code memory.
3. When we use the IN instruction, we can use the names of the I/O registers instead of their addresses.
4. The IN instruction is available in all of the AVRs, whereas LDS is not implemented in some of the AVRs.

Notice that in using the IN instruction we can access only the standard I/O memory, while we can access all parts of the data memory using the LDS instruction.

OUT instruction (OUT to I/O location)

```
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/O register will have the same value as the GPR. For example, the "`OUT PORTD,R10`" instruction will copy the contents of R10 into PORTD (location 0x0B of the I/O memory).

Notice that in the OUT instruction, the I/O registers are referred to by their I/O addresses (like the IN instruction).

The following program copies 0xE6 to the SPL register:

```
LDI   R20,0xE6     ;load R20 with 0xE6
OUT   SPL, R20     ;out R20 to SPL
```

We must remember that there is no instruction to copy an immediate value to an I/O register nor to an SRAM location.

The following program copies PIND to PORTB:

```
IN    R0, PIND     ;load R20 with the contents of I/O reg PIND
OUT   PORTB, R0    ;out R20 to PORTB
```

In Example 2-3 we use JMP to repeat an action indefinitely. JMP is similar to "goto" in the C language. We will study looping in Chapter 3.

Example 2-3

Write a program to get data from the PINB and send it to the I/O register of PORTC continuously.

Solution:

```
AGAIN:IN    R16, PINB   ;bring data from PortB into R16
      OUT   PORTC,R16   ;send it to Port C
      JMP   AGAIN       ;keep doing it forever
```

MOV instruction

The MOV instruction is used to copy data among the GPR registers of R0–R31. 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 R10:

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

For instance, if R20 contains 60, after execution of the above instruction both R20 and R10 will contain 60.

More ALU instructions involving the GPRs

The following program adds 0x19 to the contents of location 0x220 and stores the result in location 0x221:

```
LDI   R20, 0x19   ;load R20 with 0x19
LDS   R21, 0x220  ;load R21 with the contents of location 0x220
ADD   R21, R20    ;R21 = R21 + R20
STS   0x221, R21  ;store R21 to location 0x221
```

INC instruction

```
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 0x430 by 1:

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

SUB instruction

The SUB instruction has the following 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
```

The following program subtracts 5 from the contents of location 0x300 and stores the result in location 0x320:

```
LDS   R0, 0x300   ;R0 = contents of location 0x300
LDI   R16, 0x5    ;R16 = 0x5
SUB   R0, R16     ;R0 = R0 - R16
STS   0x320,R0    ;store the contents of R0 to location 0x320
```

The following program decrements the contents of R10, by 1:

```
LDI   R16, 0x1    ;load 1 to R16
SUB   R10, R16    ;R10 = R10 - R16
```

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.

DEC instruction

The DEC instruction has the following 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 R10:

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

In the following program, we put the value 3 into R30. Then the value in R30 is decremented.

```
LDI   R30, 3          ;R30 = 3
DEC   R30             ;R30 has 2
DEC   R30             ;R30 has 1
DEC   R30             ;R30 has 0
```

In the next chapter we will use the DEC instruction for looping.

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.

COM instruction

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 0x55 into R16 and then send it to PORTB. Then the content of R16 is complemented, which becomes AA in hex. The 01010101 (0x55) is inverted and becomes 10101010 (0xAA).

```
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)
```

Examine Example 2-4.

Example 2-4

Write a simple program to toggle the I/O register of PORTB continuously forever.

Solution:

```
      LDI   R20, 0x55   ;R20 = 0x55
      OUT   PORTB, R20  ;move R20 to Port B (PB = 0x55)
L1:   COM   R20         ;complement R20
      OUT   PORTB, R20  ;move R20 to Port B
      JMP   L1          ;repeat forever (see Chapter 3 for JMP)
```

The above concepts are important and must be understood since there are a large number of instructions with these formats.

Regarding Tables 2-2 and 2-3 the following points must be noted:

1. The instructions in Table 2-2 operate on two GPR registers of source (Rr) and destination (Rd) and then place the result in the destination register (Rd)
2. The instructions in Table 2-3 operate on a single GPR register and place the result in the same register.

Review Questions

1. True or false. No value can be loaded directly into internal SRAM.
2. Write instructions to load value 0x95 into the *SPL* I/O register.
3. Write instructions to add 2 to the contents of R18.
4. Write instructions to add the values 0x16 and 0xCD. Place the result in location 0x400 of the data memory.
5. What is the largest hex value that can be moved into a location in the data memory? What is the decimal equivalent of the hex value?
6. "ADD R16, R3" puts the result in _____ .
7. What does "OUT OCR0A, R23" do?
8. What is wrong with "STS OCR0A, R23"? What does it do?

SECTION 2.4: AVR STATUS REGISTER

Like all other microprocessors, the AVR has a flag register to indicate arithmetic conditions such as the carry bit. The flag register in the AVR is called the *status register* (SREG). In this section, we discuss various bits of this register and provide some examples of how it is altered. Chapters 3 and 5 show how the flag bits of the status register are used.

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, V, S, and H are called *conditional flags*, meaning that they indicate some conditions that result after an instruction is executed. Each of the conditional flags can be used to perform a conditional branch (jump), as we will see in Chapters 3 and 5.

Bit	D7							D0
SREG	I	T	H	S	V	N	Z	C

C – Carry flag
Z – Zero flag
N – Negative flag
V – Overflow flag
S – Sign flag
H – Half carry
T – Bit copy storage
I – Global Interrupt Enable

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

The following is a brief explanation of the flag bits of the status register. The impact of instructions on this register is then discussed.

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. Chapter 5 shows how the carry flag is used.

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. See Chapter 3 to see how we use the Z flag for looping.

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. The negative and V flag bits are used for the signed number arithmetic operations and are discussed in Chapter 5.

V, the overflow flag

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. The V and N flag bits are used for signed number arithmetic operations and are discussed in Chapter 5.

S, the Sign bit

This flag is the result of Exclusive-ORing of N and V flags. See Chapter 5 for more information.

H, Half carry flag

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). See Chapter 5 for more information.

The T flag bit is discussed in Chapter 6 while Chapter 10 covers the I flag.

ADD instruction and the status register

Next we examine the impact of the ADD instruction on the flag bits C, H, and Z of the status register. Some examples should clarify their meanings. Although all the flag bits C, Z, H, V, S, and N are affected by the ADD instruction, we will focus on flags C, H, and Z for now. The other flag bits are discussed in Chapter 5, because they relate only to signed number operations. Examine Example 2-5 to see the impact of the DEC instruction on selected flag bits. See also Examples 2-6 through 2-8 to see the impact of the ADD instruction on selected flag bits.

Example 2-5

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 2-6

Show the status of the C, H, and Z flags after the addition of 0x38 and 0x2F in 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 D4 bit.
Z = 0 because the R16 (the result) has a value other than 0 after the addition.

Example 2-7

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:

$9C	1001 1100	
+ $64	0110 0100	
$100	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 value 0 in it after the addition.

Example 2-8

Show the status of the C, H, and Z flags after the addition of 0x88 and 0x93 in 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 has a value other than 0 after the addition.

Not all instructions affect the flags

Some instructions affect all the six flag bits C, H, Z, S, V, and N (e.g., ADD). But some instructions affect no flag bits at all. The load instructions are in this category. And some instructions affect only some of the flag bits. The logic instructions (e.g., AND) are in this category.

Table 2-4 shows the instructions and the flag bits affected by them. Appendix A provides a complete list of all the instructions and their associated flag bits.

Table 2-4: Instructions That Affect Flag Bits

Instruction	C	Z	N	V	S	H
ADD	X	X	X	X	X	X
ADC	X	X	X	X	X	X
ADIW	X	X	X	X	X	
AND		X	X	X	X	
ANDI		X	X	X	X	
CBR		X	X	X	X	
CLR		X	X	X	X	
COM	X	X	X	X	X	
DEC		X	X	X	X	
EOR		X	X	X	X	
FMUL	X	X				
INC		X	X	X	X	
LSL	X	X	X	X		X
LSR	X	X	X	X		
OR		X	X	X	X	
ORI		X	X	X	X	
ROL	X	X	X	X		X
ROR	X	X	X	X		
SEN			1			
SEZ		1				
SUB	X	X	X	X	X	X
SUBI	X	X	X	X	X	X
TST		X	X	X	X	

Note: X can be 0 or 1. (See Chapter 5 for how to use these instructions.)

Flag bits and decision making

There are instructions that will make a conditional jump (branch) based on the status of the flag bits. Table 2-5 provides some of these instructions. Chapter 3 discusses the conditional branch instructions and how they are used.

Table 2-5: AVR Branch (Jump) Instructions Using Flag Bits

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