CHAPTER 2

AVR ARCHITECTURE AND ASSEMBLY LANGUAGE PROGRAMMING

OBJECTIVES

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

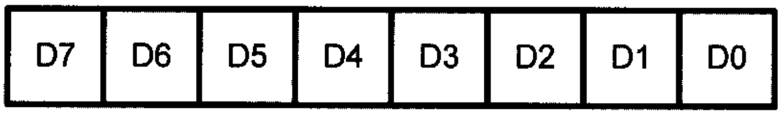
>> List the registers of the AVR microcontroller
>> Examine the data memory of the AVR microcontroller
>> Perform simple operations, such as ADD and load, and access internal RAM memory in the AVR microcontroller
>> Explain the purpose of the status register
>> Discuss data RAM memory space allocation in the AVR microcontroller
>> Code simple AVR Assembly language instructions
>> Describe AVR data types and directives
>> Assemble and run a AVR program using AVR 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 AVR 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 inside the AVR and the 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 AVR 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.

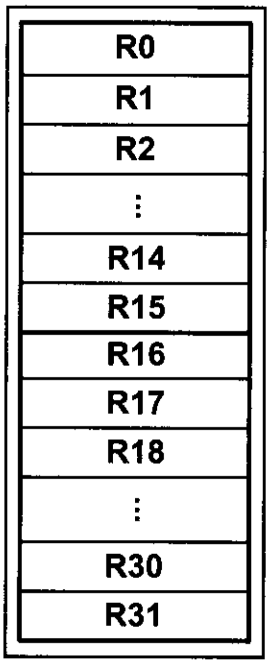


Figure 2-1. GPRs

LDI instruction

Simply stated, the LDI instruction copies 8-bit data into the general purpose registers. It has the following format:

```
LDI Rd,K     ;load Rd (destination) with Immediate value K
             ;d must be between 16 and 31
```

K is an 8-bit value that can be 0–255 in decimal, or 00–FF in hex, and Rd is 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)
```

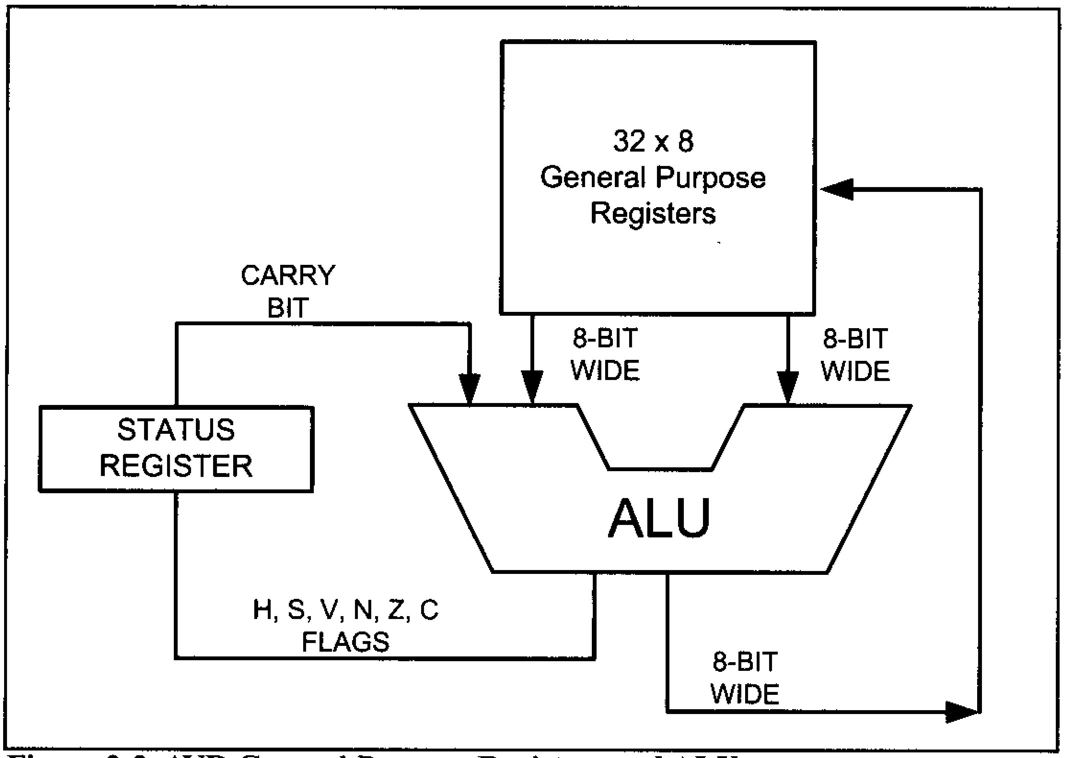


Figure 2-2. AVR General Purpose Registers and ALU

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 instructions to move 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.

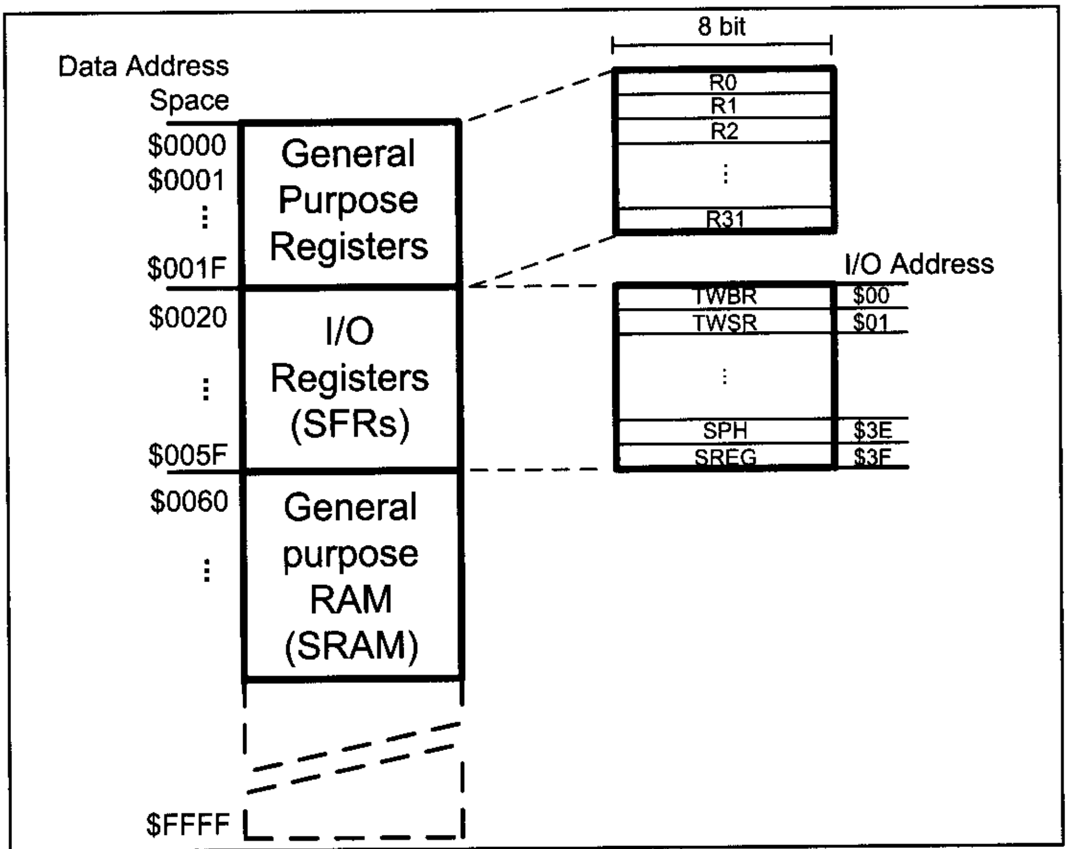


Figure 2-3. The Data Memory for AVRs with No Extended I/O Memory

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 than 32 I/O pins (e.g., ATmega64, 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.

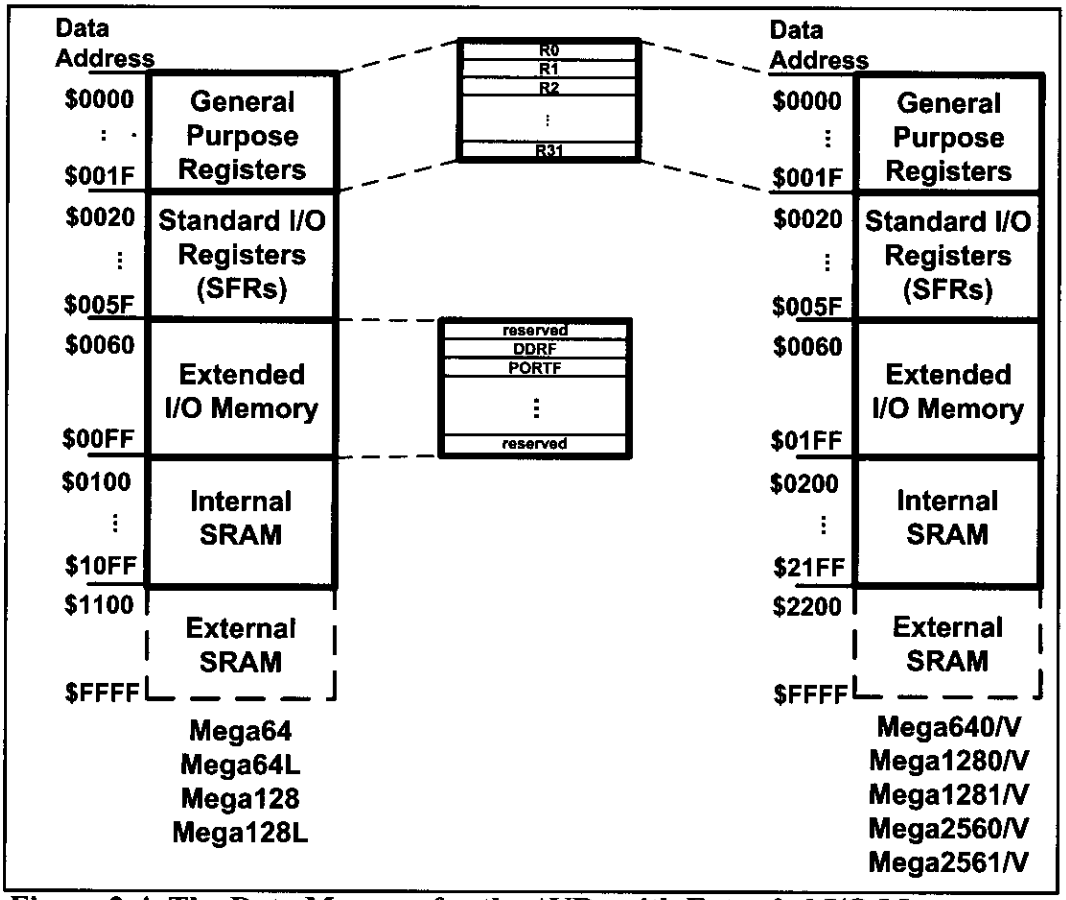


Figure 2-4. The Data Memory for the AVRs with Extended I/O Memory

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. Again, 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.

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. 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
ATmega128	4352		64+160		4096		32
ATmega2560	8704		64+416		8192		32

Extracted from http://www.atmel.com

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 2-3, 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 0x200. As you can see in Figure 2-3, 0x200 is located in the internal SRAM:

```
LDS R5,0x200 ;load R5 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
```

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.

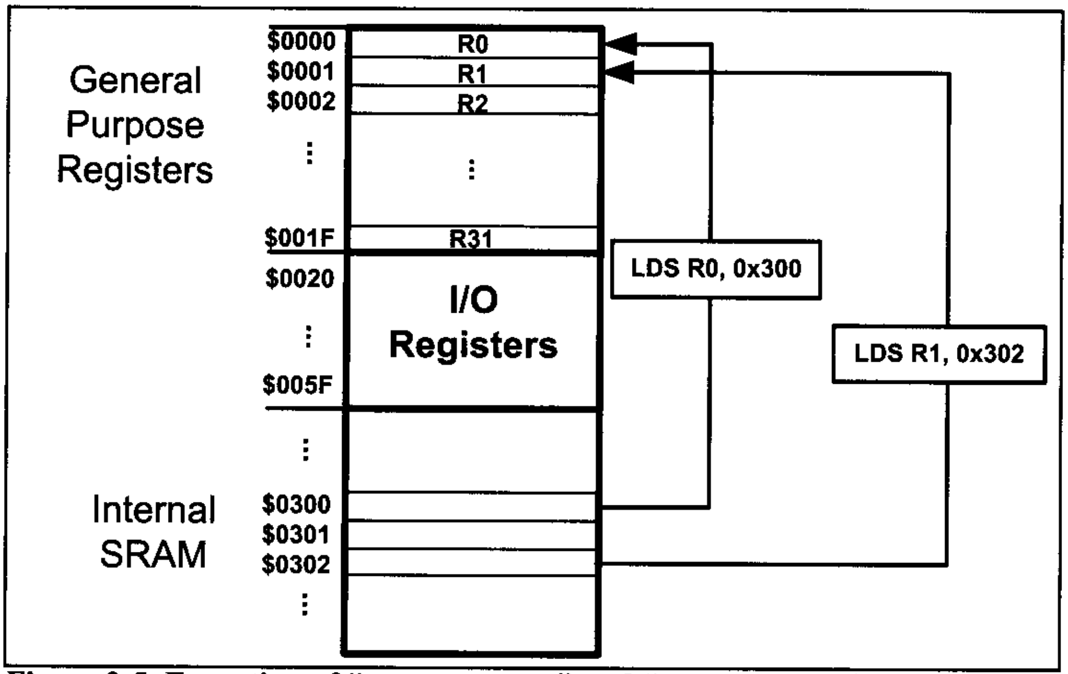


Figure 2-5. Execution of "`LDS R0,0x300`" and "`LDS R1,0x302`" Instructions

	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 2-3, 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 2-3, 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 0x38, 0x35, and 0x32, 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)
```

As we saw in Figure 2-3, PORTB, PORTC, and PORTD are part of the special function registers in 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	TWBR
$21	$01	TWSR
$22	$02	TWAR
$23	$03	TWDR
$24	$04	ADCL
$25	$05	ADCH
$26	$06	ADCSRA
$27	$07	ADMUX
$28	$08	ACSR
$29	$09	UBRRL
$2A	$0A	UCSRB
$2B	$0B	UCSRA
$2C	$0C	UDR
$2D	$0D	SPCR
$2E	$0E	SPSR
$2F	$0F	SPDR
$30	$10	PIND
$31	$11	DDRD
$32	$12	PORTD
$33	$13	PINC
$34	$14	DDRC
$35	$15	PORTC
$36	$16	PINB
$37	$17	DDRB
$38	$18	PORTB
$39	$19	PINA
$3A	$1A	DDRA
$3B	$1B	PORTA
$3C	$1C	EECR
$3D	$1D	EEDR
$3E	$1E	EEARL
$3F	$1F	EEARH
$40	$20	UBRRC
		UBRRH
$41	$21	WDTCR
$42	$22	ASSR
$43	$23	OCR2
$44	$24	TCNT2
$45	$25	TCCR2
$46	$26	ICR1L
$47	$27	ICR1H
$48	$28	OCR1BL
$49	$29	OCR1BH
$4A	$2A	OCR1AL
$4B	$2B	OCR1AH
$4C	$2C	TCNT1L
$4D	$2D	TCNT1H
$4E	$2E	TCCR1B
$4F	$2F	TCCR1A
$50	$30	SFIOR
$51	$31	OCDR
		OSCCAL
$52	$32	TCNT0
$53	$33	TCCR0
$54	$34	MCUCSR
$55	$35	MCUCR
$56	$36	TWCR
$57	$37	SPMCR
$58	$38	TIFR
$59	$39	TIMSK
$5A	$3A	GIFR
$5B	$3B	GICR
$5C	$3C	OCR0
$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 ATmega32 and Their Data Memory Address Locations

Example 2-1

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 R20,0x16`" instruction will copy the contents of location $16 of the I/O memory (whose data memory address is 0x36) into R20. As shown in Figure 2-7, I/O address 0x16 belongs to PINB, so the instruction copies the contents of PINB to R20.

The following instruction loads R19 with the contents of location 0x10 of the I/O memory:

```
IN R19,0x10         ;load R19 with location $10 (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
```

Notice that to be able to use the names of the I/O addresses instead of the I/O addresses we should include the proper header files, as discussed in Section 2.5. 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 12 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 we cannot copy an immediate value to an I/O register nor to an SRAM location.

The following program copies PIND to PORTA:

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

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 PORT C 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 the SFR location of 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 PORT B continuously forever.

Solution:

```
      LDI   R20, 0x55    ;R20 = 0x55
      OUT   PORTB, R20   ;move R20 to Port B SFR (PB = 0x55)
L1:   COM   R20          ;complement R20
      OUT   PORTB, R20   ;move R20 to Port B SFR
      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 OCR0, R23`" do?
8. What is wrong with "`STS OCR0, 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.

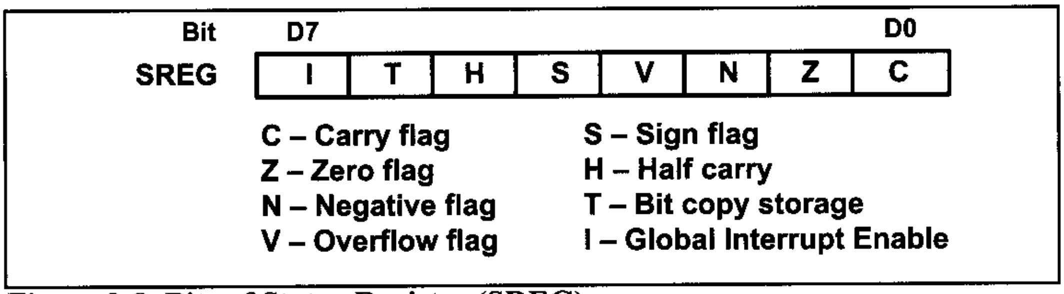


Figure 2-8. Bits of Status Register (SREG)

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

Review Questions

1. The flag register in the AVR is called the ________.
2. What is the size of the flag register in the AVR?
3. Find the C, Z, and H flag bits for the following code:

```
LDI   R20, 0x9F
LDI   R21, 0x61
ADD   R20, R21
```

4. Find the C, Z, and H flag bits for the following code:

```
LDI   R17, 0x82
LDI   R23, 0x22
ADD   R17, R23
```

5. Find the C, Z, and H flag bits for the following code:

```
LDI   R20, 0x67
LDI   R21, 0x99
ADD   R20, R21
```

SECTION 2.5: AVR DATA FORMAT AND DIRECTIVES

In this section we look at some widely used data formats and directives supported by the AVR assembler.

AVR data type

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 0xFF, or 0 to 255 in decimal) to be processed by the CPU. For examples of how to process data larger than 8 bits, see Chapter 5. The data types used by the AVR can be positive or negative. A discussion of signed numbers is given in Chapter 5 also. The bit-addressable data is discussed in Chapters 4 and 6.

Data format representation

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

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`

We use both of these methods in this book, because many application notes out there use one of them and we need to get used to them.

Here are a few lines of code that use the hex format:

```
LDI   R28,$75      ;R28 = 0x75
SUBI  R28,0x11     ;R28 = 0x75 - 0x11 = 0x64
SUBI  R28,0X20     ;R28 = 0x64 - 0x20 = 0x44
ANDI  R28,0xF      ;R28 = 0x44 - 0x0F = 0x35
```

Binary numbers

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

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

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

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

Decimal numbers

To indicate decimal numbers in an AVR assembler we simply use the decimal (e.g., 12) and nothing before or after it. Here are some examples of how to use it:

```
LDI  R17, 12  ;R17 = 00001100 or 0C in hex
SUBI R17, 2   ;R17 = 12 - 2 = 10 where 10 is equal to 0x0A
```

ASCII characters

To represent ASCII data in an AVR assembler we use single quotes as follows:

```
LDI R23,'2'  ;R23 = 00110010 or 32 in hex (See Appendix F)
```

This is the same as other assemblers such as the 8051 and x86. Here are some more examples:

```
LDI  R20,'9';R20 = 0x39, which is hex number for ASCII '9'
SUBI R20,'1';R20 = 0x39 - 0x31 = 0x8
            ;(31 hex is for ASCII '1')
```

To represent a string, double quotes are used; and for defining ASCII strings (more than one character), we use the .DB (define byte) directive. We will see .DB usage in Chapter 6.

Assembler directives

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

.EQU (equate)

This is used to define a constant value or a fixed address. The .EQU directive does not set aside storage for a data item, but associates a constant number with a data or an address label so that when the label appears in the program, its constant will be substituted for the label. The following uses .EQU for the counter constant, and then the constant is used to load the R21 register:

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

When executing the above instruction "`LDI R21, COUNT`", the register R21 will be loaded with the value 25H. What is the advantage of using .EQU? Assume that a constant (a fixed value) is used throughout the program, and the programmer wants to change its value everywhere. By the use of .EQU, the programmer can change it once and the assembler will change all of its occurrences throughout the program. This allows the programmer to avoid searching the entire program trying to find every occurrence.

We mentioned earlier that we can use the names of the I/O registers instead of their addresses (e.g., we can write "OUT PORTA,R20" instead of "OUT 0x1B,R20"). This is done with the help of the .EQU directive. In include files such as M32DEF.INC the I/O register names are associated with their addresses using the .EQU directive. For example, in M32DEF.INC the following pseudo-instruction exists, which associates 0x1B (the address of PORTB) with the PORTB.

```
      .EQU  PORTB = 0x1B
```

.SET

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

Using .EQU for fixed data assignment

To get more practice using .EQU to assign fixed data, examine the following:

```
                  ;in hexadecimal
      .EQU DATA1 = 0x39       ;one way to define hex value
      .EQU DATA2 = $39        ;another way to define hex value

                  ;in binary
```

```
.EQU DATA3 = 0b00110101   ;binary (35 in hex)

             ;in decimal
.EQU DATA4 = 39          ;decimal numbers (27 in hex)

             ;in ASCII
.EQU DATA5 = '2'         ;ASCII characters
```

We use .DB to allocate code ROM memory locations for fixed data such as ASCII strings. See Chapter 6 for more examples.

Using .EQU for SFR address assignment

.EQU is also widely used to assign SFR addresses. Examine the following code:

```
.EQU COUNTER = 0x00      ;counter value 00
.EQU PORTB = 0x18        ;SFR Port B address
LDI  R16, COUNTER        ;R16 = 0x00
OUT  PORTB, R16          ;Port B (loc 0x18) now has 00 too
```

Using .EQU for RAM address assignment

Another common usage of .EQU is for the address assignment of the internal SRAM. Examine the following rewrite of an earlier example using .EQU:

```
.EQU  SUM = 0x120        ;assign RAM loc to SUM
LDI   R20, 5             ;load R20 with 5
LDI   R21, 2             ;load R21 with 2
ADD   R20, R21           ;R20 = R20 + R21
ADD   R20, R21           ;R20 = R20 + R21
STS   SUM, R20           ;store the result in loc 0x120
```

This is especially helpful when the address needs to be changed in order to use a different AVR chip for a given project. It is much easier to refer to a name than a number when accessing RAM address locations.

.ORG (origin)

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

.INCLUDE directive

The .include directive tells the AVR assembler to add the contents of a file to our program (like the #include directive in C language). In Table 2-6, you see the files that you must include whenever you want to use any of the AVRs.

For example, when you want to use ATmega32, you must write the following instruction at the beginning of your program:

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

Table 2-6: Some of the Common AVRs and Their Include Files

ATMEGA		ATTINY		Special Purpose	
ATmega8	m8def.inc	ATtiny11	tn11def.inc	AT90CAN32	can32def.inc
ATmega16	m16def.inc	ATtiny12	tn12def.inc	AT90CAN64	can64def.inc
ATmega32	m32def.inc	ATtiny22	tn22def.inc	AT90PWM2	pwm2def.inc
ATmega64	m64def.inc	ATtiny44	tn44def.inc	AT90PWM3	pwm3def.inc
ATmega128	m128def.inc	ATtiny85	tn85def.inc	AT90USB646	usb646def.inc
ATmega256	m256def.inc				
ATmega2560	m2560def.inc				

Rules for labels in Assembly language

By choosing label names that are meaningful, a programmer can make a program much easier to read and maintain. There are several rules that names must follow. First, each label name must be unique. The names used for labels in Assembly language programming consist of alphabetic letters in both uppercase and lowercase, the digits 0 through 9, and the special characters question mark (?), period (.), at (@), underline (_), and dollar sign ($). The first character of the label must be an alphabetic character. In other words, it cannot be a number. Every assembler has some reserved words that must not be used as labels in the program. Foremost among the reserved words are the mnemonics for the instructions. For example, "`LDI`" and "`ADD`" are reserved because they are instruction mnemonics. In addition to the mnemonics there are some other reserved words. Check your assembler for the list of reserved words.

Review Questions

1. Give two ways for hex data representation in the AVR assembler.
2. Show how to represent decimal 99 in formats of (a) hex, (b) decimal, and (c) binary in the AVR assembler.
3. What is the advantage in using the .EQU directive to define a constant value?
4. Show the hex number value used by the following directives:
   (a) `.EQU ASC_DATA = '4'`   (b) `.EQU MY_DATA=0B00011111`
5. Give the value in R22 for the following:
```
.EQU  MYCOUNT = 15
LDI   R22, MYCOUNT
```
6. Give the value in data memory location 0x200 for the following:
```
.EQU  MYCOUNT = 0x95
.EQU  MYREG = 0x200
LDI   R22, MYCOUNT
STS   MYREG, R22
```
7. Give the value in data memory 0x63 for the following:
```
.EQU MYDATA = 12
.EQU MYREG = 0x63
.EQU FACTOR = 0x10
LDI  R19, MYDATA
ADD  R19, FACTOR
STS  MYREG, R19
```

SECTION 2.6: INTRODUCTION TO AVR ASSEMBLY PROGRAMMING

In this section we discuss Assembly language format and define some widely used terminology associated with Assembly language programming.

While the CPU can work only in binary, it can do so at a very high speed. It is quite tedious and slow for humans, however, to deal with 0s and 1s in order to program the computer. A program that consists of 0s and 1s is called *machine language*. In the early days of the computer, programmers coded programs in machine language. Although the hexadecimal system was used as a more efficient way to represent binary numbers, the process of working in machine code was still cumbersome for humans. Eventually, Assembly languages were developed, which provided mnemonics for the machine code instructions, plus other features that made programming faster and less prone to error. The term *mnemonic* is frequently used in computer science and engineering literature to refer to codes and abbreviations that are relatively easy to remember. Assembly language programs must be translated into machine code by a program called an *assembler*. Assembly language is referred to as a *low-level language* because it deals directly with the internal structure of the CPU. To program in Assembly language, the programmer must know all the registers of the CPU and the size of each, as well as other details.

Today, one can use many different programming languages, such as BASIC, Pascal, C, C++, Java, and numerous others. These languages are called *high-level languages* because the programmer does not have to be concerned with the internal details of the CPU. Whereas an *assembler* is used to translate an Assembly language program into machine code (sometimes also called *object code* or opcode for operation code), high-level languages are translated into machine code by a program called a *compiler*. For instance, to write a program in C, one must use a C compiler to translate the program into machine language. Next we look at AVR Assembly language format.

Structure of Assembly language

An Assembly language program consists of, among other things, a series of lines of Assembly language instructions. An Assembly language instruction consists of a mnemonic, optionally followed by one or two operands. The operands are the data items being manipulated, and the mnemonics are the commands to the CPU, telling it what to do with those items.

An Assembly language program (see Program 2-1) is a series of statements, or lines, which are either Assembly language instructions such as ADD and LDI, or statements called *directives*. While instructions tell the CPU what to do, directives (also called *pseudo-instructions*) give directions to the assembler. For example, in Program 2-1, while the LDI and ADD instructions are commands to the CPU, .ORG and .EQU are directives to the assembler. The directive .ORG tells the assembler to place the opcode at memory location 0, while .EQU introduces a new expression equal to a known expression.

An Assembly language instruction consists of four fields:

```
[ label:]    mnemonic  [ operands]   [ ;comment]
```

```
     ;AVR Assembly Language Program To Add Some Data.
     ;store SUM in SRAM location 0x300.

     .EQU  SUM   = 0x300       ;SRAM loc $300 for SUM

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

Program 2-1: Sample of an Assembly Language Program

Brackets indicate that a field is optional and not all lines have them. Brackets should not be typed in. Regarding the above format, the following points should be noted:

1. The label field allows the program to refer to a line of code by name. The label field cannot exceed a certain number of characters. Check your assembler for the rule.
2. The Assembly language mnemonic (instruction) and operand(s) fields together perform the real work of the program and accomplish the tasks for which the program was written. In Assembly language statements such as

   ```
   LDI   R23, $55
   ADD   R23, R19
   SUBI  R23, $67
   ```

   ADD and LDI are the mnemonics that produce opcodes; the "$55" and "$67" are the operands. Instead of a mnemonic and an operand, these two fields could contain assembler pseudo-instructions, or directives. Remember that directives do not generate any machine code (opcode) and are used only by the assembler, as opposed to instructions that are translated into machine code (opcode) for the CPU to execute. In Program 2-1 the commands .ORG (origin) and .EQU are examples of directives. More of these pseudo-instructions are discussed in future chapters.
3. The comment field begins with a semicolon comment indicator ";". Comments may be at the end of a line or on a line by themselves. The assembler ignores comments, but they are indispensable to programmers. Although comments are optional, it is recommended that they be used to describe the program in a way that makes it easier for someone else to read and understand.
4. Notice the label "HERE" in the label field in Program 2-1. In the JMP the AVR is told to stay in this loop indefinitely. If your system has a monitor program you do not need this line and should delete it from your program. In Section 2.7 we will see how to create a ready-to-run program.

Review Questions

1. What is the purpose of pseudo-instructions?
2. ____________ are translated by the assembler into machine code, whereas ________________ are not.
3. True or false. Assembly language is a high-level language.
4. Which of the following instructions produces opcode? List all that do.
   (a) `LDI R16,0x25` (b) `ADD R23,R19` (c) `.ORG 0x500` (d) `JMP HERE`
5. Pseudo-instructions are also called __________.
6. True or false. Assembler directives are not used by the CPU itself. They are simply a guide to the assembler.
7. In Question 4, which one is an assembler directive?

SECTION 2.7: ASSEMBLING AN AVR PROGRAM

Now that the basic form of an Assembly language program has been given, the next question is: How it is created, assembled, and made ready to run? The steps to create an executable Assembly language program (Figure 2-9) are outlined as follows:

1. First we use a text editor to type in a program similar to Program 2-1. In the case of the AVR microcontrollers, we use the AVRStudio IDE, which has a text editor, assembler, simulator, and much more all in one software package. It is an excellent development software that supports all the AVR chips and is free. Many editors or word processors are also available that can be used to create or edit the program. A widely used editor is the Notepad in Windows, which comes with all Microsoft operating systems. Notice that the editor must be able

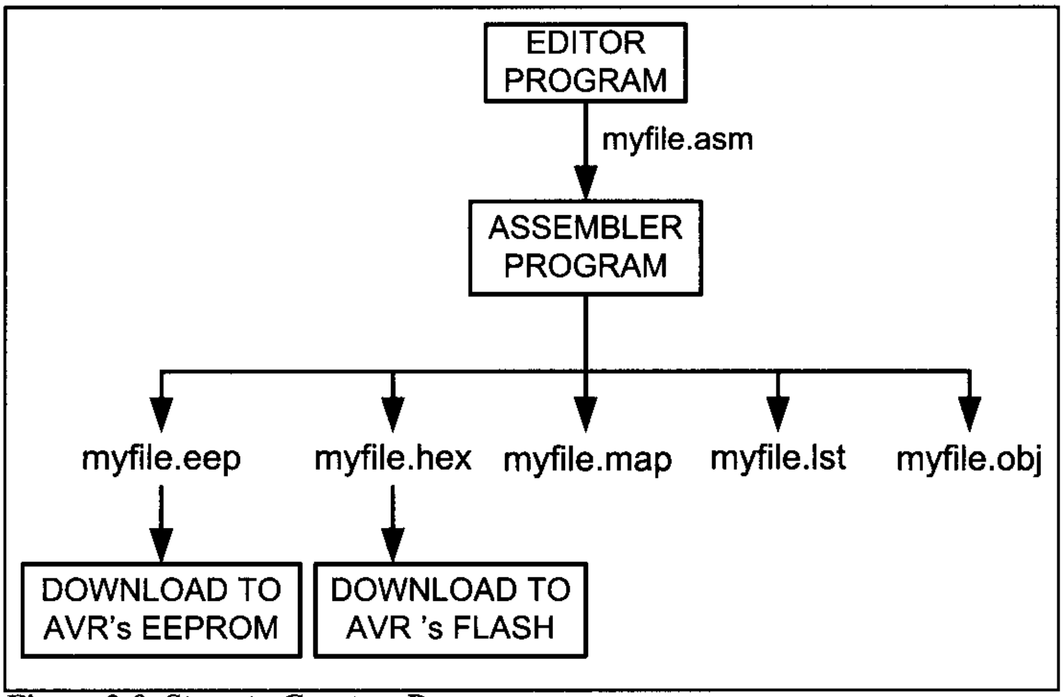


Figure 2-9. Steps to Create a Program

```
AVRASM: AVR macro assembler 2.1.2 (build 99 Nov  4 2005 09:35:05)
Copyright (C) 1995-2005 ATMEL Corporation

F:\AVR\Sample\Sample.asm(7): error: Invalid register
F:\AVR\Sample\Sample.asm(8): error: Operand(s) out of range in 'ldi r17,0x3432'
F:\AVR\Sample\Sample.asm(9): error: Undefined symbol: R38
F:\AVR\Sample\Sample.asm(9): error: Invalid register
F:\AVR\Sample\Sample.asm(16): No EEPROM data, deleting F:\AVR\Sample\Sample.eep

Assembly failed, 4 errors, 0 warnings
```

Figure 2-10. Sample of an AVR Error

to produce an ASCII file. For assemblers, the file names follow the usual DOS conventions, but the source file has the extension "asm". The "asm" extension for the source file is used by an assembler in the next step.

2. The "asm" source file containing the program code created in step 1 is fed to the AVR assembler. The assembler produces an object file, a hex file, an eeprom file, a list file, and a map file. The object file has the extension "obj", the hex file extension is "hex", the list file extension is "lst", the map file extension is "map", and the eeprom file has the extension "eep". After a successful link, the hex file is ready to be burned into the AVR's program ROM and is downloaded into the AVR Trainer. We can write the eeprom file into the AVR's EEPROM to initialize the EEPROM. See Chapter 8 for more details.

More about asm and object files

The asm file is also called the *source* file and must have the "asm" extension. As mentioned earlier, this file is created with a text editor such as Windows Notepad. Many assemblers come with a text editor. The assembler converts the asm file's Assembly language instructions into machine language and provides the obj (object) file. The object file, as mentioned earlier, has an "obj" as its extension. The object file is used as input to a simulator or an emulator.

Before we can assemble a program to create a ready-to-run program, we must make sure that it is error free. The AVR Studio IDE provides us error messages and we examine them to see the nature of syntax errors. The assembler will not assemble the program until all the syntax errors are fixed. A sample of an error message is shown in Figure 2-10.

"lst" and "map" files

The map file shows the labels defined in the program together with their values. Examine Figure 2-11. It shows the Map file of Program 2-1.

The lst (list) file, which is optional, is very useful to the programmer. The

```
AVRASM ver. 2.1.2  F:\AVR\Sample\Sample.asm Sun Apr 06 23:39:32 2008


EQU  SUM           00000300
CSEG HERE          00000009
```

Figure 2-11. Map File of Program 2-1

```
AVRASM ver. 2.1.2  F:\AVR\Sample\Sample.asm Tue Mar 11 11:28:34 2008

               ;store SUM in SRAM location 0x300.
                 .DEVICE ATMega32
                 .EQU  SUM   = 0x300       ;SRAM loc $300 for SUM

                 .ORG 00                   ;start at address 0
000000 e205      LDI R16, 0x25             ;R16 = 0x25
000001 e314      LDI R17, $34              ;R17 = 0x34
000002 e321      LDI R18, 0b00110001       ;R18 = 0x31
000003 0f01      ADD R16, R17              ;add R17 to R16
000004 0f02      ADD R16, R18              ;add R18 to R16
000005 e01b      LDI R17, 11               ;R17 = 0x0B
000006 0f01      ADD R16, R17              ;add R17 to R16
000007 9300 0300 STS SUM, R16              ;save the SUM in loc $300
000009 940c 0009 HERE: JMP HERE            ;stay here forever

RESOURCE USE INFORMATION
------------------------
...
Memory use summary [ bytes] :
Segment   Begin     End       Code   Data   Used    Size   Use%
---------------------------------------------------------------
[ .cseg] 0x000000 0x000016     22      0     22 unknown     -
[ .dseg] 0x000060 0x000060      0      0      0 unknown     -
[ .eseg] 0x000000 0x000000      0      0      0 unknown     -

Assembly complete, 0 errors, 0 warnings
```
Figure 2-12. List File of Program 2-1

list shows the binary and source code; it also shows which instructions are used in the source code, and the amount of memory the program uses. See Figure 2-12.

Many assemblers assume that the list file is not wanted unless you indicate that you want to produce it. These files can be accessed by a text editor such as Notepad and displayed on the monitor, or sent to the printer to get a hard copy. The programmer uses the list and map files to ensure correct system design.

There are many different AVR assemblers available for free nowadays. If you use the Windows operating system, AVR Studio can be a good choice. It has a nice environment and provides great help.

Review Questions

1. True or false. The AVR Studio IDE and Windows Notepad text editor both produce an ASCII file.
2. True or false. The extension for the source file is "asm".
3. Which of the following files can be produced by a text editor?
   (a) myprog.asm (b) myprog.obj (c) myprog.hex (d) myprog.lst
4. Which of the following files is produced by an assembler?
   (a) myprog.asm (b) myprog.obj (c) myprog.hex (d) myprog.lst

SECTION 2.8: THE PROGRAM COUNTER AND PROGRAM ROM SPACE IN THE AVR

In this section we discuss the role of the program counter (PC) in executing a program and show how the code is fetched from ROM and executed. We will also discuss the program (code) ROM space for various AVR family members. Finally, we examine the Harvard architecture of the AVR.

Program counter in the AVR

The most important register in the AVR microcontroller is the PC (program counter). The program counter is used by the CPU to point to the address of the next instruction to be executed. As the CPU fetches the opcode from the program ROM, the program counter is incremented automatically to point to the next instruction. 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.

In AVR microcontrollers each Flash memory location is 2 bytes wide. For example, in ATmega32, whose Flash is 32K bytes, the Flash is organized as 16K× 16, and its program counter is 14 bits wide (2^{14} = 16K memory locations). The ATmega64 has a 15-bit program counter, so its Flash has 32K locations (2^{15}=32K), with each location containing 2 bytes (32K × 2 bytes = 64K bytes).

In the case of a 16-bit program counter, the code space is 64K (2^{16} = 64K), which occupies the 0000–$FFFF address range. 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. Because each Flash location is 2 bytes wide, the AVR can have a maximum of 8M bytes of code. However, at the time of this writing, none of the members of the AVR family have the entire 8M bytes of on-chip ROM installed. See Table 2-7.

Table 2-7: AVR On-chip ROM Size and Address Space

	On-chip Code ROM (Bytes)	Code Address Range (Hex)	ROM Organization
ATtiny25	2K	00000–003FF	1K × 2 bytes
ATmega8	8K	00000–00FFF	4K × 2 bytes
ATmega32	32K	00000–03FFF	16K × 2 bytes
ATmega64	64K	00000–07FFF	32K × 2 bytes
ATmega128	128K	00000–0FFFF	64K × 2 bytes
ATmega256	256K	00000–1FFFF	128K × 2 bytes

ROM memory map in the AVR family

As we just discussed, some family members have only a few kilobytes of on-chip ROM and some, such as the ATmega128, have 128K of ROM. The point to remember is that 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). It must be noted that while the first

Example 2-9

Find the ROM memory address of each of the following AVR chips:
(a) ATtiny25 with 2 KB
(b) ATmega16 with 16 KB
(c) ATmega64 with 64 KB

Solution:

(a) With 2K bytes of on-chip ROM memory, we have 2048 bytes (2 × 1024 = 2048). As each address location in AVR is 2 bytes, its Flash has 1024 locations (2048 / 2 = 1024). This maps to address locations of 0000 to $03FF. Notice that 0 is always the first location.
(b) With 16K bytes of on-chip ROM memory, we have 16,384 bytes (16 × 1024 = 16,384), and 8192 locations (16384 / 2 = 8192), which gives 0000–$1FFF.
(c) With 64K we have 65,535 bytes (64 × 1024 = 65,535), and 32,768 locations. Converting 32,768 to hex, we get $8000; therefore, the memory space is 0000 to $7FFF.

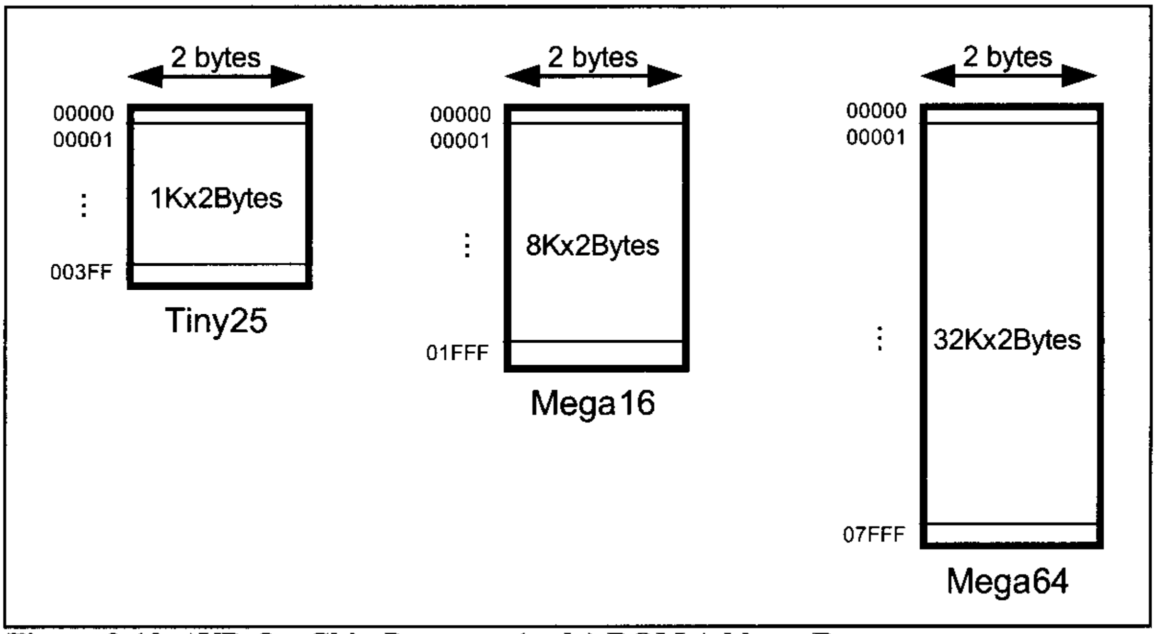


Figure 2-13. AVR On-Chip Program (code) ROM Address Range

location of program ROM inside the AVR has the address of 000000, the last location can be different depending on the size of the ROM on the chip. (See Figure 2-13.) Among the AVR family members, the ATmega8 has 8K of on-chip ROM. This 8K ROM memory is organized as 4K × 2 bytes and has memory addresses of 00000 to $00FFF. Therefore, the first location of on-chip ROM of this AVR has an address of 00000 and the last location has the address of $00FFF. Look at Example 2-9 to see how this is computed.

Where the AVR wakes up when it is powered up

One question that we must ask about any microcontroller (or microprocessor) is: At what address does the CPU wake up when power is applied? Each microprocessor is different. In the case of the AVR microcontrollers (that is, all members regardless of the family and variation), the microcontroller wakes up at memory address 0000 when it is powered up. By powering up we mean applying V_{CC} to the RESET pin as discussed in Chapter 8. In other words, when the AVR is powered up, the PC (program counter) has the value of 00000 in it. This means that it expects the first opcode to be stored at ROM address \$00000. For this reason, in the AVR system, the first opcode must be burned into memory location \$00000 of program ROM because this is where it looks for the first instruction when it is booted. We achieve this by using the .ORG statement in the source program as shown earlier. Next we discuss the step-by-step action of the program counter in fetching and executing a sample program.

Placing code in program ROM

To get a better understanding of the role of the program counter in fetching and executing a program, we examine the action of the program counter as each instruction is fetched and executed. First, we examine once more the list file of the sample program and show how the code is placed into the Flash ROM of the AVR chip. As we can see, the opcode and operand for each instruction are listed on the left side of the list file.

After the program is burned into ROM of an AVR family member such as ATmega32 or ATtiny11, the opcode and operand are placed in ROM memory locations starting at 0000 as shown in the Program 2-1 list file.

The list shows that address 0000 contains E205, which is the opcode for moving a value into R16, and the operand (in this case 0x25) to be moved to R16. Therefore, the instruction "`LDI R16,0x25`" has a machine code of "E205", where E is the opcode and 205 is the operand. See Figures 2-14 and 2-15. Similarly, the machine code "E314" is located in ROM memory location 0001 and represents the opcode and the operands for the instruction "`LDI R17,$34`". In

1 1 1 0	k k k k	d d d d	k k k k

LDI Rd, k **$16 \le d \le 31$, $0 \le K \le 255$**

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

E k_1 | d k_0 LDI Rd, k_1k_0	E 2 | 0 5 LDI R16, 0x25
E 3 | 1 4 LDI R17, 0x34	E 3 | 2 1 LDI R18, 0x31

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

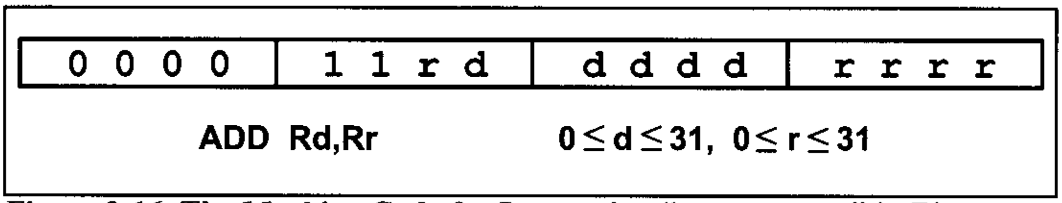


Figure 2-16. The Machine Code for Instruction "ADD Rd,Rr" in Binary

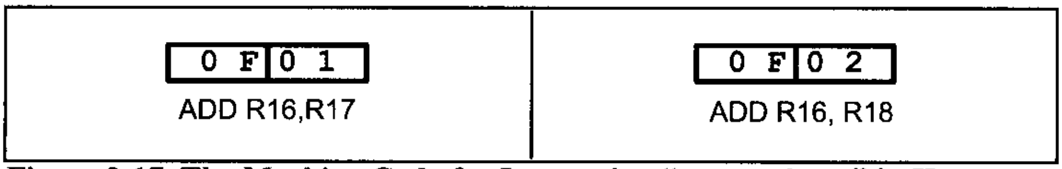


Figure 2-17. The Machine Code for Instruction "ADD Rd,Rr" in Hex

the same way, machine code "E321" is located in memory location 0002 and represents the opcode and the operand for the instruction "`LDI R18,0b00110001`". The memory location 0003 has the machine code of 0F01, which is the opcode and the operands for the instruction "`ADD R16,R17`". Similarly, the machine code "0F02" is located in memory location 0004 and represents the opcode and the operands for the instruction "`ADD R16, R18`". See Figures 2-16 and 2-17. The memory location 0005 has the opcode and operand for the "`LDI R17,11`" instruction. The memory location 0006 has the opcode and operand for the "`ADD R16,R17`" instruction. The opcode for instruction "`STS SUM,R16`" is located at address 00007 and its address of 0x300 at address 00008. The opcode for "`JMP HERE`" and its target address are located in locations 00009 and 0000A. While all the instructions in this program are 2-byte instructions, the JMP and STS instructions are 4-byte instructions. The reasons are explained at the end of this section.

Executing a program instruction by instruction

Assuming that the above program is burned into the ROM of an AVR chip, 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. In the case of the above program the first code is E205, which is the code for moving operand 0x25 to R16. Upon executing the code, the CPU places the value of 25 in R16. Now one instruction is finished. Then the program counter is incremented to point to 00001 (PC = 00001), which contains code E314, the machine code for the instruction "`LDI R17,0x34`".
2. Upon executing the machine code E314, the value 0x34 is loaded to R17. Then the program counter is incremented to 0002.
3. ROM location 0002 has the machine code for instruction "`LDI R18, 0x31`". This instruction is executed and now PC = 0003.
4. This process goes on until all the instructions up to "`ADD R16,R17`" are fetched and executed. Notice that all the above instructions are 2-byte instructions; that is, each one takes two bytes of ROM (one word).
5. Now PC = 0007 points to the next instruction, which is "`STS SUM,R16`". This is a 2-word (4-byte) instruction. It takes addresses of 07 and 08. When the

instruction is executed, the content of R16 is stored into memory location 0x300. After the execution of this instruction, PC = 0009.

6. Now PC = 0009 points to the next instruction, which is "`JMP HERE`". This is a 2-word (4-byte) instruction. It takes addresses of 09 and 0A. After the execution of this instruction, PC = 0009. This keeps the program in an infinite loop. The fact that the program counter points at the next instruction to be executed explains why some microprocessors (notably the x86) call the program counter the *instruction pointer.*

ROM width in the AVR

As we have seen so far in this section, each location of the address space holds two bytes (a word). If we have 16 address lines, this will give us 2^{16} locations, which is 64K of memory location with an address map of 0000–FFFFH. To bring in more information (code or data) into the CPU, AVR increased the width of the data bus to 16 bits. In other words, the AVR is word-addressable. In contrast, the 8051 CPU is byte addressable. In a sense, the data bus is like traffic lanes on the highway where each lane is 8 bits wide. The more lanes, the more information we can bring into the CPU for processing. For the AVR, the internal data bus between the code ROM and the CPU is 16 bits wide, as shown in Figure 2-18. Therefore, the 64K ROM space is shown as 32K × 16 using a 16-bit word data size. The same rule applies to the entire program address space of AVR, which is 8M, organized as 4M × 16. The widening of the data path between the program ROM and the CPU is another way in which the AVR designers increased the processing power of the AVR family. Another reason to make the code ROM 16 bits wide is to match it with the instruction width of the AVR because the vast majority of the instructions are 2-byte instructions. This way, the CPU brings in an instruction from ROM every time it makes a trip to the program ROM. That will make instruction fetch a single cycle, as we will see in the next chapter when instruction timing is discussed.

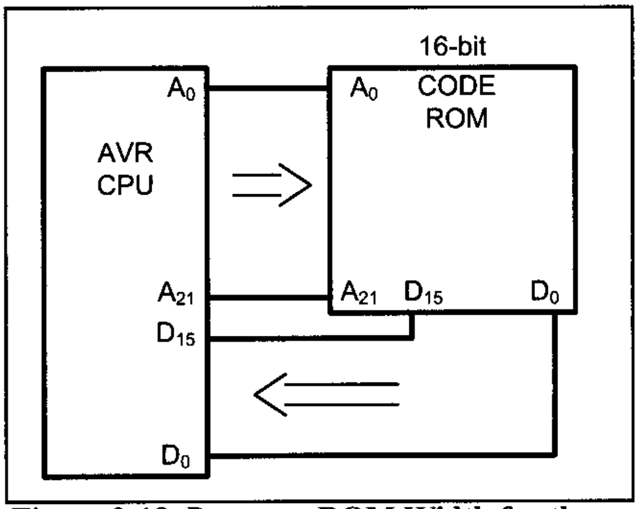


Figure 2-18. Program ROM Width for the AVR

The AVR designers have made all instructions either 2-byte or 4-byte; there are no 1-byte or 3-byte instructions, as is the case with the x86 and 8051 chips. This is part of the RISC architectural philosophy, which we will study in the next section. It must also be noted that the data memory SRAM in the AVR microcontroller is still 8-bit, and it is byte-addressable.

Harvard architecture in the AVR

As we mentioned in Chapter 0, AVR uses Harvard architecture, which means that there are separate buses for the code and the data memory. See Figure 2-19. The Program Bus provides access to the Program Flash ROM whereas the Data Bus is used for bringing data to the CPU.

As we can see in Figure 2-19, in the Program Bus, the data bus is 16 bits wide and the address bus is as wide as the PC register to enable the CPU to address the entire Program Flash ROM.

In the Data Bus, the data bus is 8 bits wide. As a result, the CPU can access one byte of data at a time. The address bus is 16 bits wide. Thus the data memory space can be up to 64K bytes.

In Sections 2-2 and 2-3, you learned about data memory space and how to use the STS and LDS instructions. When the CPU wants to execute the "LDS Rn,k" instruction, it puts k on the address bus of the Data Bus, and receives data through the data bus. For example, to execute "LDS R20, 0x90", the CPU puts 0x90 on the address bus. The location $90 is in the SRAM (see Figure 2-4). Thus, the SRAM puts the contents of location $90 on the data bus. The CPU gets the contents of location $90 through the data bus and puts it in R20.

The "STS k,Rn" instruction is executed similarly. The CPU puts k on the address bus and the contents of Rn on the data bus. The unit whose address is on the address bus receives the contents of data bus. For example, to execute the "STS $100,R30" instruction the CPU puts the contents of R30 on the data bus and $100 on the address bus. Because $100 is bigger than $60, the address belongs to SRAM; thus SRAM gets the contents of the data bus and puts it in location $100 of the SRAM.

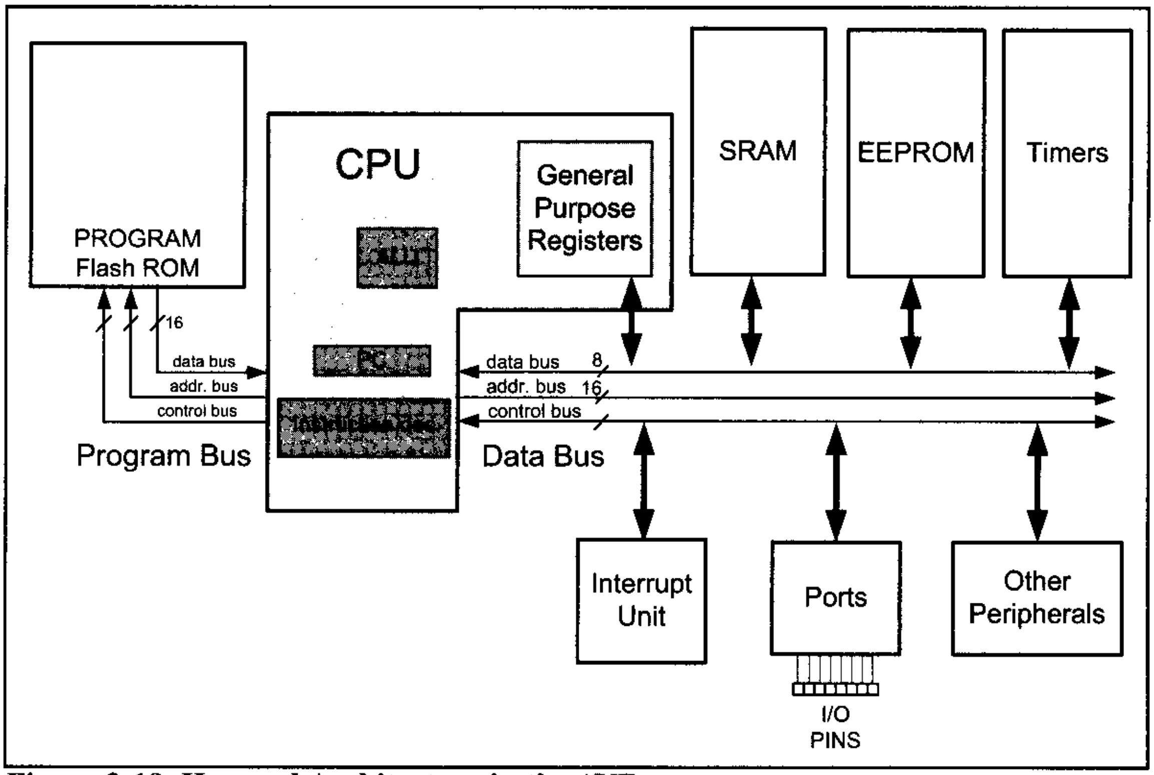


Figure 2-19. Harvard Architecture in the AVR

Little endian vs. big endian war

Examine the placing of the code in the AVR ROM, shown in Figure 2-20. The low byte goes to the low memory location, and the high byte goes to the high memory address. This convention is called little endian to contrast it with big endian. The origin of the terms *big endian* and *little endian* is from an argument in a Gulliver's Travels story over how an egg should be opened: from the big end or the little end. In the big endian method, the high byte goes to the low address, whereas in the little endian method, the high byte goes to the high address and the low byte to the low address. All Intel microprocessors and many microcontrollers use the little endian convention. Freescale (formerly Motorola) microprocessors, along with some mainframes, use big endian. The difference might seem as trivial as whether to break an egg from the big end or the little end, but it is a nuisance in converting software from one camp to be run on a computer of the other camp. Some microprocessors, such as the PowerPC from IBM/Freescale, let the software designer choose little endian or big endian convention.

Address	High byte	Low byte
00000	E2	05
00001	E3	14
00002	E3	21
00003	0F	01
00004	0F	02
00005	E0	1B
00006	0F	01
00007	93	00
00008	03	00
00009	94	0C
0000A	00	09

Figure 2-20. AVR Program ROM Contents for Program 2-1 List File

Instruction size of the AVR

Recall that the AVR instructions are either 2-byte or 4-byte. Almost all the instructions in the AVR are 2-byte instructions. The exceptions are STS, JMP, and a few others. Next we explore the instruction size and formation for a few of the instructions we have used in this chapter. This should give you some insight into the instructions of the AVR.

LDI instruction formation

The LDI is a 2-byte (16-bit) instruction. Of the 16 bits, the first 4 bits are set aside for the opcode, the second and the fourth 4 bits are used for the value of 00 to $FF, and the third 4 bits present the destination register. This is shown below.

```
LDI Rd, K   ;load register Rd with value K
```

1110	KKKK	dddd	KKKK

$16 \leq d \leq 31, 0 \leq K \leq 255$

ADD instruction formation

The ADD is a 2-byte (16-bit) instruction. Of the 16 bits, the first 6 bits are set aside for the opcode, and the other 10 bits represent the source and the destination registers. This is shown below.

```
ADD Rd, Rr    ;Add Rd to Rr
```

0000	11rd	dddd	rrrr

$0 \le d \le 31,\ 0 \le r \le 31$

STS instruction formation

The STS is a 4-byte (32-bit) instruction. Of the 32 bits, the first 16 bits are set aside for the opcode and the address of the source, and the other 16 bits are used for the address of the destination. This is shown below.

```
STS k, Rr    ;Store register Rr in memory location k
```

1001	001r	rrrr	0000
kkkk	kkkk	kkkk	kkkk

$0 \le r \le 31,\ 0 \le k \le 65535$

LDS instruction formation

The LDS is a 4-byte (32-bit) instruction. Of the 32 bits, the first 16 bits are set aside for the opcode and the destination register, and the other 16 bits are used for the address of the source memory location. This is shown below.

```
LDS Rd, k    ;Load from memory location k to register Rd
```

1001	000d	dddd	0000
kkkk	kkkk	kkkk	kkkk

$0 \le d \le 31,\ 0 \le k \le 65535$

IN instruction formation

The IN is a 2-byte (16-bit) instruction. Of the 16 bits, the first 5 bits are set aside for the opcode, and the other 11 bits are used for the address of the source memory location, and destination register. This is shown below.

```
IN Rd, A  ;load from Address A of I/O memory into register Rd
```

1011	0AAd	dddd	AAAA

$0 \le d \le 31,\ 0 \le A \le 63$

OUT instruction formation

The OUT is a 2-byte (16-bit) instruction. Of the 16 bits, the first 5 bits are set aside for the opcode, and the other 11 bits are used for the address of the source memory location and destination register. This is shown below.

```
OUT A, Rr  ;Store register Rr in I/O memory location A
```

1011	1AAr	rrrr	AAAA

$0 \leq d \leq 31, 0 \leq A \leq 63$

JMP instruction formation

The JMP is a 4-byte (32-bit) instruction. Of the 32 bits, only 10 bits are set aside for the opcode, and the rest (22 bits) are used for the target address of the JMP. This is shown below.

The 22-bit address gives us 4M of address space; so, it can address all of the ROM space.

```
JMP k   ;Jump to address k
```

1001	010k	kkkk	110k
kkkk	kkkk	kkkk	kkkk

$0 \leq k \leq 4M$

Review Questions

1. In the AVR, the program counter is, at most, ______ bits wide.
2. True or false. Every member of the AVR family, regardless of the program ROM size, wakes up at memory $0000 when it is powered up.
3. At what ROM location do we store the first opcode of an AVR program?
4. The instruction "`LDI R20,0x44`" is a ____-byte instruction.
5. The instruction "`JMP label`" is a ____-byte instruction.
6. True or false. All the instructions in the AVR are 2- or 4-byte instructions.

SECTION 2.9: 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. While in the case of

x86 and other general purpose microprocessors this architecture is very expensive and unrealistic, in today's microcontrollers this is not a problem. As we saw in the last section, the AVR has Harvard architecture.

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

Atmel used all three methods to increase the processing power of the AVR microcontrollers. In this section we discuss the merits of RISC architecture and examine how the AVR microcontrollers have adapted it.

RISC architecture

In the early 1980s, a controversy broke out in the computer design community, but unlike most controversies, it did not go away. Since the 1960s, in all mainframes and minicomputers, designers put as many instructions as they could think of into the CPU. Some of these instructions performed complex tasks. An example is adding data memory locations and storing the sum into memory. Naturally, microprocessor designers followed the lead of minicomputer and mainframe designers. Because these microprocessors used such a large number of instructions, many of which performed highly complex activities, they came to be known as CISC (complex instruction set computer) processors. According to several studies in the 1970s, many of these complex instructions etched into CPUs were never used by programmers and compilers. The huge cost of implementing a large number of instructions (some of them complex) into the microprocessor, plus the fact that a good portion of the transistors on the chip are used by the instruction decoder, made some designers think of simplifying and reducing the number of instructions. As this concept developed, the resulting processors came to be known as RISC (reduced instruction set computer).

Features of RISC

The following are some of the features of RISC as implemented by the AVR microcontroller.

Feature 1

RISC processors have a fixed instruction size. In a CISC microcontroller such as the 8051, instructions can be 1, 2, or even 3 bytes. For example, look at the following instructions in the 8051:

```
CLR   C                          ;clear Carry flag, a 1-byte instruction
ADD   Accumulator, #mybyte       ;a 2-byte instruction
LJMP  target_address             ;a 3-byte instruction
```

This variable instruction size makes the task of the instruction decoder very difficult because the size of the incoming instruction is never known. In a RISC architecture, the size of all instructions is fixed. Therefore, the CPU can decode the instructions quickly. This is like a bricklayer working with bricks of the same size as opposed to using bricks of variable sizes. Of course, it is much more efficient to use bricks of the same size. In the last section we saw how the AVR uses 2-byte instructions with very few 4-byte instructions.

Feature 2

One of the major characteristics of RISC architecture is a large number of registers. All RISC architectures have at least 32 registers. Of these 32 registers, only a few are assigned to a dedicated function. One advantage of a large number of registers is that it avoids the need for a large stack to store parameters. Although a stack can be implemented on a RISC processor, it is not as essential as in CISC because so many registers are available. In the AVR microcontrollers the use of 32 general purpose registers satisfies this RISC feature. The stack for the AVR is covered in the next chapter.

Feature 3

RISC processors have a small instruction set. RISC processors have only basic instructions such as ADD, SUB, MUL, LOAD, STORE, AND, OR, EOR, CALL, JUMP, and so on. The limited number of instructions is one of the criticisms leveled at the RISC processor because it makes the job of Assembly language programmers much more tedious and difficult compared to CISC Assembly language programming. This is one reason that RISC is used more commonly in high-level language environments such as the C programming language rather than Assembly language environments. It is interesting to note that some defenders of CISC have called it "complete instruction set computer" instead of "complex instruction set computer" because it has a complete set of every kind of instruction. How many of these instructions are used and how often is another matter. The limited number of instructions in RISC leads to programs that are large. Although these programs can use more memory, this is not a problem because memory is cheap. Before the advent of semiconductor memory in the 1960s, however, CISC designers had to pack as much action as possible into a single instruction to get the maximum bang for their buck. In the ATmega we have around 130 instructions. We will examine more of the instruction set for the AVR in future chapters.

Feature 4

At this point, one might ask, with all the difficulties associated with RISC programming, what is the gain? The most important characteristic of the RISC processor is that more than 95% of instructions are executed with only one clock cycle, in contrast to CISC instructions. Even some of the 5% of the RISC instructions that are executed with two clock cycles can be executed with one clock cycle by juggling instructions around (code scheduling). Code scheduling is most often the job of the compiler. We will examine the instruction cycle time and pipelining of the AVR in Chapter 3.

Feature 5

RISC processors have separate buses for data and code. In all the x86 processors, like all other CISC computers, there is one set of buses for the address (e.g., A0–A24 in the 80286) and another set of buses for data (e.g., D0–D15 in the 80286) carrying opcodes and operands in and out of the CPU. To access any section of memory, regardless of whether it contains code or data operands, the same address bus and data bus are used. In RISC processors, there are four sets of buses: (1) a set of data buses for carrying data (operands) in and out of the CPU, (2) a set of address buses for accessing the data, (3) a set of buses to carry the opcodes, and (4) a set of address buses to access the opcodes. The use of separate buses for code and data operands is commonly referred to as Harvard architecture. We examined the Harvard architecture of the AVR in the previous section.

Feature 6

Because CISC has such a large number of instructions, each with so many different addressing modes, microinstructions (microcode) are used to implement them. The implementation of microinstructions inside the CPU employs more than 40–60% of transistors in many CISC processors. RISC instructions, however, due to the small set of instructions, are implemented using the hardwire method. Hardwiring of RISC instructions takes no more than 10% of the transistors.

Feature 7

RISC uses load/store architecture. In CISC microprocessors, data can be manipulated while it is still in memory. For example, in instructions such as "ADD Reg, Memory", the microprocessor must bring the contents of the external memory location into the CPU, add it to the contents of the register, then move the result back to the external memory location. The problem is there might be a delay in accessing the data from external memory. Then the whole process would be stalled, preventing other instructions from proceeding in the pipeline. In RISC, designers did away with these kinds of instructions. 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. All these instructions must be performed by first bringing both operands into the registers inside the CPU, then performing the arithmetic or logic operation, and then sending the result back to memory. This idea was first implemented by the Cray 1 supercomputer in 1976 and is commonly referred to as load/store architecture. In the last section, we saw that the arithmetic and logic operations are between the data memory (internal) locations, but none involves a ROM location. For example, there is no "ADD ROM-Loc" instruction in AVR.

In concluding this discussion of RISC processors, it is interesting to note that RISC technology was explored by the scientists at IBM in the mid-1970s, but it was David Patterson of the University of California at Berkeley who in 1980 brought the merits of RISC concepts to the attention of computer scientists. It must also be noted that in recent years CISC processors such as the Pentium have used some RISC features in their design. This was the only way they could enhance the processing power of the x86 processors and stay competitive. Of course, they had to use lots of transistors to do the job, because they had to deal with all the CISC instructions of the x86 processors and the legacy software of DOS/Windows.

Review Questions

1. What do RISC and CISC stand for?
2. True or false. The CISC architecture executes the vast majority of its instructions in 2, 3, or more clock cycles, while RISC executes them in one clock.
3. RISC processors normally have a _____ (large, small) number of general-purpose registers.
4. True or false. Instructions such as "ADD R16, ROMmemory" do not exist in RISC microcontrollers such as the AVR.
5. How many instructions does the ATmega have?
6. True or false. While CISC instructions are of variable sizes, RISC instructions are all the same size.
7. Which of the following operations do not exist for the ADD instruction in RISC?
   (a) register to register (b) immediate to register (c) memory to memory
8. True or false. Harvard architecture uses the same address and data buses to fetch both code and data.

SECTION 2.10: VIEWING REGISTERS AND MEMORY WITH AVR STUDIO IDE

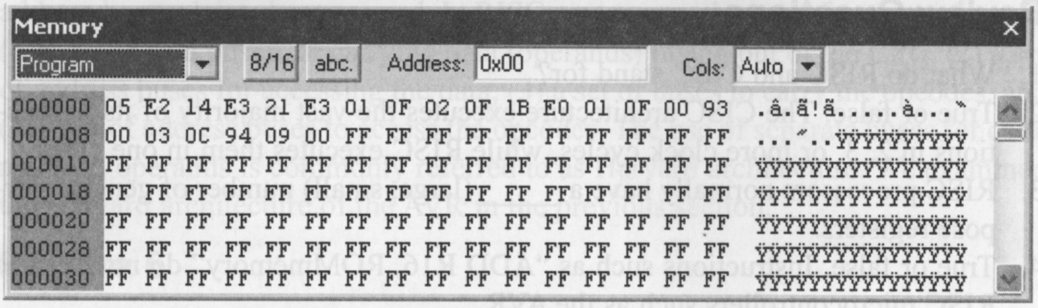
The AVR microcontroller has great tools and support systems, many of them free or inexpensive. AVR Studio is an assembler and simulator provided for free by Atmel Corporation and can be downloaded from the www.atmel.com website. See http://www.MicroDigitalEd.com for tutorials on how to use the AVR Studio assembler and simulator.

Many assemblers and C compilers come with a simulator. Simulators allow us to view the contents of registers and memory after executing each instruction (single-stepping). It is strongly recommended to use a simulator to single-step some of the programs in this chapter and future chapters. Single-stepping a program with a simulator gives us a deeper understanding of microcontroller architecture, in addition to the fact that we can use it to find the errors in our programs.

Figures 2-21 through 2-23 show screenshots for AVR simulators from AVR Studio.

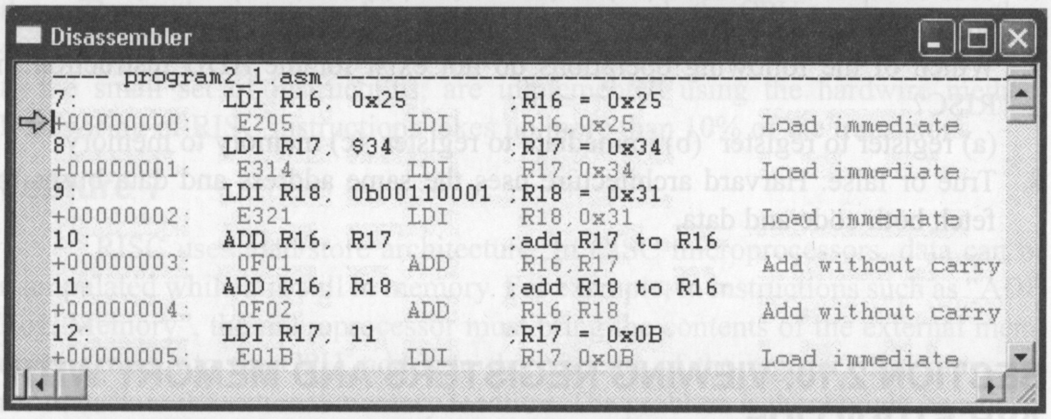
See the following website for a tutorial on using AVR Studio:

http://www.MicroDigitalEd.com



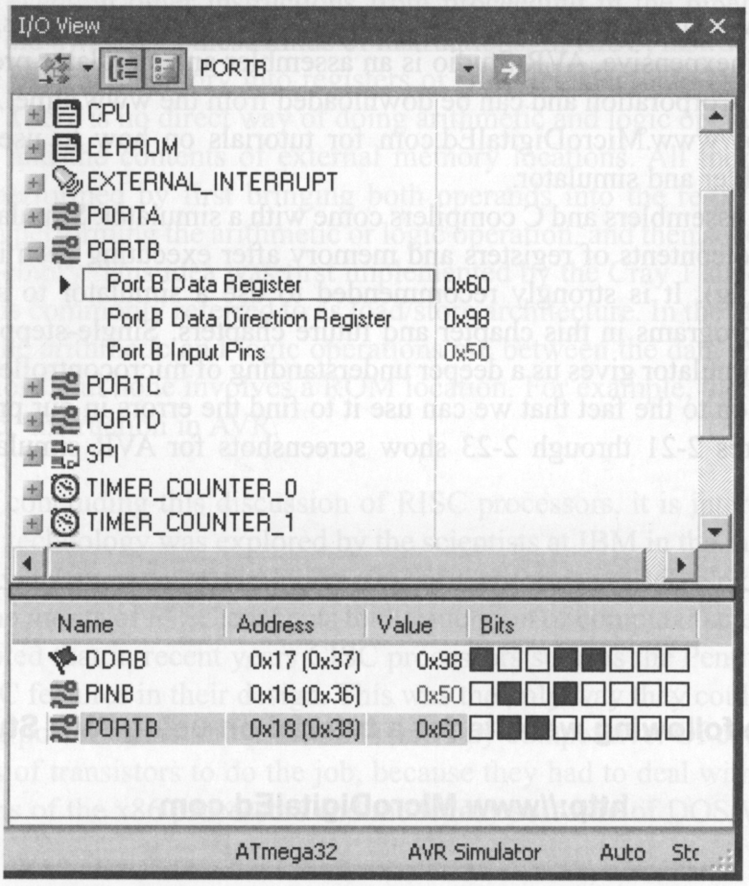
```
Memory
Program   8/16  abc.   Address: 0x00        Cols: Auto
000000 05 E2 14 E3 21 E3 01 0F 02 0F 1B E0 01 0F 00 93  .â.ã!ã.....à...“
000008 00 03 0C 94 09 00 FF FF FF FF FF FF FF FF FF FF  ...”..ÿÿÿÿÿÿÿÿÿÿ
000010 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
000018 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
000020 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
000028 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
000030 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
```

Figure 2-21. Data Memory Window in AVR Studio IDE



```
Disassembler
--- program2_1.asm -----------------------------------------------
7:          LDI R16, 0x25       ;R16 = 0x25
+00000000:    E205      LDI      R16,0x25          Load immediate
8:          LDI R17, $34        ;R17 = 0x34
+00000001:    E314      LDI      R17,0x34          Load immediate
9:          LDI R18, 0b00110001 ;R18 = 0x31
+00000002:    E321      LDI      R18,0x31          Load immediate
10:         ADD R16, R17        ;add R17 to R16
+00000003:    0F01      ADD      R16,R17           Add without carry
11:         ADD R16, R18        ;add R18 to R16
+00000004:    0F02      ADD      R16,R18           Add without carry
12:         LDI R17  11         ;R17 = 0x0B
+00000005:    E01B      LDI      R17,0x0B          Load immediate
```

Figure 2-22. Program ROM (Disassembler) Window in AVR Studio IDE



```
I/O View
PORTB
+ CPU
+ EEPROM
+ EXTERNAL_INTERRUPT
+ PORTA
- PORTB
    Port B Data Register            0x60
    Port B Data Direction Register  0x98
    Port B Input Pins               0x50
+ PORTC
+ PORTD
+ SPI
+ TIMER_COUNTER_0
+ TIMER_COUNTER_1
```

Name	Address	Value	Bits
DDRB	0x17 (0x37)	0x98	■□□■■□□□
PINB	0x16 (0x36)	0x50	□■□■□□□□
PORTB	0x18 (0x38)	0x60	□■■□□□□□

ATmega32   AVR Simulator   Auto   Stc

Figure 2-23. I/O View Window in AVR Studio IDE

SUMMARY

This chapter began with an exploration of the major registers of the AVR, including general purpose registers, I/O registers, and internal SRAM, and the program counter. The use of these registers was demonstrated in the context of programming examples. The process of creating an Assembly language program was described from writing the source file, to assembling it, linking, and executing the program. The PC (program counter) register always points to the next instruction to be executed. The way the AVR uses program Flash ROM space was explored because AVR Assembly language programmers must be aware of where programs are placed in ROM, and how much memory is available.

An Assembly language program is composed of a series of statements that are either instructions or pseudo-instructions, also called *directives*. Instructions are translated by the assembler into machine code. Pseudo-instructions are not translated into machine code: They direct the assembler in how to translate instructions into machine code. Some pseudo-instructions, called *data directives*, are used to define data. Data is allocated in byte-size increments. The data can be in binary, hex, decimal, or ASCII formats.

Flags are useful to programmers because they indicate certain conditions, such as carry or zero, that result from execution of instructions. The concepts of the RISC and Harvard architectures were also explored.

The RISC architecture allows the design of much more powerful microcontrollers. It has a simple instruction set and uses a large number of registers. Harvard architecture allows us to bring more code and data to the CPU faster. The use of a wider data bus in the AVR allows us to fetch an instruction every cycle because the AVR instructions are typically 2 bytes.

PROBLEMS

SECTION 2.1: THE GENERAL PURPOSE REGISTERS IN THE AVR

1. AVR is a(n) _____-bit microcontroller.
2. The general purpose registers are _____ bits wide.
3. The value in LDI is _____ bits wide.
4. The largest number that can be loaded into the GPRs is _____ in hex.
5. What is the result of the following code and where is it kept?

```
        LDI       R20,$15
        LDI       R21,$13
        ADD       R20,R21
```

6. Which of the following is (are) illegal?

   (a) `LDI R20,500`   (b) `LDI  R23,50`   (c) `LDI  R1,00`
   (d) `LDI R16,$255`   (e) `LDI  R42,$25`   (f) `LDI R23,0xF5`
   (g) `LDI 123,0x50`

7. Which of the following is (are) illegal?

   (a) `ADD R20,R11`   (b) `ADD R16,R1`   (c) `ADD R52,R16`

8. What is the result of the following code and where is it kept?

```
          LDI   R19,$25
          ADD   R19,$1F
```

9. What is the result of the following code and where is it kept?

```
          LDI   R21,0x15
          ADD   R21,0xEA
```

10. True or false. We have 32 general purpose registers in the AVR.

SECTION 2.2: THE AVR DATA MEMORY

11. AVR data memory consists of ____________ (Flash ROM, internal SRAM).
12. True or false. The special function register in AVR is called the I/O register.
13. True or false. The I/O registers are part of the data memory space.
14. True or false. The general-purpose registers are not part of the data memory space.
15. True or false. The data memory is the same size in all members of AVR.
16. If we add the I/O registers, internal RAM, and general purpose register sizes together we should get the total space for the ______.
17. Find the data memory size for the following AVR chips:
    (a) ATmega32 (b) ATmega16 (c) ATtiny44
18. What is the difference between the EEPROM and data RAM space in the AVR?
19. Can we have an AVR chip with no EEPROM?
20. Can we have an AVR chip with no data memory?
21. What is the address range for the internal RAM?
22. What is the maximum number of bytes that the AVR can have for the data memory?

SECTION 2.3: USING INSTRUCTIONS WITH THE DATA MEMORY

23. Show a simple code to load values $30 and $97 into locations $105 and $106, respectively.
24. Show a simple code to load the value $55 into locations $300–$308.
25. Show a simple code to load the value $5F into the PORTB I/O register.
26. True or false. We cannot load immediate values into the internal RAM directly.
27. Show a simple code to (a) load the value $11 into locations $100–$105, and (b) add the values together and place the result in R20 as they are added.
28. Repeat Problem 27, except place the result in location $105 after the addition is done.
29. Show a simple code to (a) load the value $15 into location $67, and (b) add it to R19 five times and place the result in R19 as the values are added. R19 should be zero before the addition starts.
30. Repeat Problem 29, except place the result in location $67.
31. Write a simple code to complement the contents of location $68 and place the result in R27.
32. Write a simple code to copy data from location $68 to PORTC using R19.

SECTION 2.4: AVR STATUS REGISTER

33. The status register is a(n) ______ -bit register.
34. Which bits of the status register are used for the C and H flag bits, respectively?
35. Which bits of the status register are used for the V and N flag bits, respectively?
36. In the ADD instruction, when is C raised?
37. In the ADD instruction, when is H raised?
38. What is the status of the C and Z flags after the following code?
```
    LDI R20,0xFF
    LDI R21,1
    ADD R20,R21
```
39. Find the C flag value after each of the following codes:
```
(a)LDI R20,0x54    (b)LDI R23,0      (c) LDI R30,0xFF
   LDI R25,0xC4       LDI R16,0xFF       LDI R18,0x05
   ADD R20,R25        ADD R23,R16        ADD R30,R18
```
40. Write a simple program in which the value 0x55 is added 5 times.

SECTION 2.5: AVR DATA FORMAT AND DIRECTIVES

41. State the value (in hex) used for each of the following data:
```
    .EQU MYDAT_1 = 55
    .EQU MYDAT_2 = 98
    .EQU MYDAT_3 = 'G'
    .EQU MYDAT_4 = 0x50
    .EQU MYDAT_5 = 200
    .EQU MYDAT_6 = 'A'
    .EQU MYDAT_7 = 0xAA
    .EQU MYDAT_8 = 255
    .EQU MYDAT_9 = 0B10010000
    .EQU MYDAT_10 = 0b01111110
    .EQU MYDAT_11 = 10
    .EQU MYDAT_12 = 15
```
42. State the value (in hex) for each of the following data:
```
    .EQU DAT_1 = 22
    .EQU DAT_2 = $56
    .EQU DAT_3 = 0b10011001
    .EQU DAT_4 = 32
    .EQU DAT_5 = 0xF6
    .EQU DAT_6 = 0B11111011
```

43. Show a simple code to (a) load the value $11 into locations $60–$65, and (b) add them together and place the result in R29 as the values are added. Use .EQU to assign the names TEMP0–TEMP5 to locations $60–$65.

SECTION 2.6: INTRODUCTION TO AVR ASSEMBLY PROGRAMMING and SECTION 2.7: ASSEMBLING AN AVR PROGRAM

44. Assembly language is a ________ (low, high)-level language while C is a ________ (low, high)-level language.
45. Of C and Assembly language, which is more efficient in terms of code generation (i.e., the amount of ROM space it uses)?
46. Which program produces the obj file?
47. True or false. The source file has the extension "asm".
48. True or false. The source code file can be a non-ASCII file.
49. True or false. Every source file must have .ORG and .EQU directives.
50. Do the .ORG and .SET directives produce opcodes?
51. Why are the directives also called pseudocode?
52. True or false. The .ORG directive appears in the ".lst" file.
53. The file with the _____ extension is downloaded into AVR Flash ROM.
54. Give three file extensions produced by AVR Studio.

SECTION 2.8: THE PROGRAM COUNTER AND PROGRAM ROM SPACE IN THE AVR

55. Every AVR family member wakes up at address _____ when it is powered up.
56. A programmer puts the first opcode at address $100. What happens when the microcontroller is powered up?
57. Find the number of bytes each of the following instructions takes:
    (a) `LDI R19,0x5` (b) `LDI R30,$9F` (c) `ADD R20,R21`
    (d) `ADD R22,R20` (e) `LDI R18,0x41` (f) `LDI R28,20`
    (g) `ADD R1,R3` (h) `JMP`
58. Write a program to (a) place each of your 5-digit ID numbers into a RAM locations starting at address 0x100, (b) add each digit to R19 and store the sum in RAM location 0x306, and (c) use the program listing to show the ROM memory addresses and their contents.
59. Find the address of the last location of on-chip program ROM for each of the following:
    (a) AVR with 32 KB (b) AVR with 8 KB
    (c) AVR with 64 KB (d) AVR with 16 KB
    (f) AVR with 128 KB
60. Show the lowest and highest values (in hex) that the ATmega32 program counter can take.
61. A given AVR has $7FFF as the address of the last location of its on-chip ROM. What is the size of on-chip ROM for this AVR?
62. Repeat Question 61 for $3FF.
63. Find the on-chip program ROM size in K for the AVR chip with the following address ranges:
    (a) $0000–$1FFF (b) $0000–$3FFF
    (c) $0000–$7FFF (d) $0000–$FFFF
    (e) $0000–$1FFFF (f) $00000–$3FFFF
    (g) $00000–$FFF (h) $00000–$1FF

64. Find the on-chip program ROM size in K for the AVR chips with the following address ranges:

(a) \$00000–\$3FF	(b) \$00000–\$7FF
(c) \$00000–\$7FFFF	(d) \$00000–\$FFFFF
(e) \$00000–\$1FFFFF	(f) \$00000–\$3FFFFF
(g) \$00000–\$5FFF	(h) \$00000–\$BFFFF

(Some of the above might not be in production yet.)

65. How wide is the program ROM in the AVR chip?
66. How wide is the data bus between the CPU and the program ROM in the AVR chip?
67. In instruction "`LDI R21,K`" explain why the K value cannot be larger than 255 decimal.
68. \$0C01 is the machine code for the _____ (LDI, STS, JMP, ADD) instruction.
69. In "STS memLocation,R22", explain what the size of the instruction is and how it allows one to cover the entire range of the data memory in the AVR chip.
70. In "LDS Rd, memLocation", explain what the size of the instruction is and how it allows one to cover the entire range of the data memory in the AVR chip.
71. Explain how the instruction "JMP target-addr" is able to cover the entire 4M address space of the AVR chip.

SECTION 2.9: RISC ARCHITECTURE IN THE AVR

72. What do RISC and CISC stand for?
73. In ______ (RISC, CISC) architecture we can have 1-, 2-, 3-, or 4-byte instructions.
74. In ______ (RISC, CISC) architecture instructions are fixed in size.
75. In ______ (RISC, CISC) architecture instructions are mostly executed in one or two cycles.
76. In ______ (RISC, CISC) architecture we can have an instruction to ADD a register to external memory.
77. True or false. Most instructions in CISC are executed in one or two cycles.

ANSWERS TO REVIEW QUESTIONS

SECTION 2.1: THE GENERAL PURPOSE REGISTERS IN THE AVR

1. `LDI R29,0x34`
2.
```
LDI R18,0x16
LDI R19,0xCD
ADD R19,R18
```
3. False
4. FF hex and 255 in decimal
5. 8

SECTION 2.2: THE AVR DATA MEMORY

1. False
2. Data memory
3. 8
4. 3
5. 64K
6. 64

SECTION 2.3: USING INSTRUCTIONS WITH THE DATA MEMORY

1. True
2. LDI R20,0x95
   OUT SPL,R20
3. LDI R19,2
   ADD R18,R19
4. LDI R20,0x16
   LDI R21,0xCD
   ADD R20,R21
5. FF in hex or 255 in decimal
6. R16
7. It copies the contents of R23 into the OCR0 I/O register.
8. The OCR0 presents the I/O address of the OCR0 register ($3C); but we should use the data memory address while using the STS instruction. The instruction copies the contents of R23 into the location with the data memory address of $3C (the EECR I/O register).

SECTION 2.4: AVR STATUS REGISTER

1. SREG
2. 8 bits
3.

```
    Hex         binary
     9F         1001 1111
   + 61       + 0110 0001
    100        10000 0000
```
This leads to C = 1, H = 1, and Z = 1.

4.

```
    Hex         binary
     82         1000 0010
   + 22      +  0010 0010
     A4         1010 0100
```
This leads to C = 0, H = 0, and Z = 0.

5.

```
    Hex         binary
     67         0110 0111
   + 99      +  1001 1001
    100        10000 0000
```
This leads to C = 1, H = 1, and Z = 1.

SECTION 2.5: AVR DATA FORMAT AND DIRECTIVES

1.
```
.EQU DATA1 = 0x9F
.EQU DATA2 = $9F
```

2.
```
.EQU DATA1 = 0x99
.EQU DATA2 = 99
.EQU DATA3 = 0b10011001
```

3. If the value is to be changed later, it can be done once in one place instead of at every occurrence.

4. (a) $34 (b) $1F
5. 15 in decimal (0x0F in hex)
6. Value of location 0x200 = (0x95)
7. $0C + $10 = $1C will be in data memory location $63.

SECTION 2.6: INTRODUCTION TO AVR ASSEMBLY PROGRAMMING

1. The real work is performed by instructions such as LDI and ADD. Pseudo-instructions, also called assembly directives, instruct the assembler in doing its job.
2. The instruction mnemonics, pseudo-instructions
3. False
4. All except (c)
5. Assembler directives
6. True
7. (c)

SECTION 2.7: ASSEMBLING AN AVR PROGRAM

1. True
2. True
3. (a)
4. (b), (c), and (d)

SECTION 2.8: THE PROGRAM COUNTER AND PROGRAM ROM SPACE IN THE AVR

1. 22
2. True
3. 0000H
4. 2
5. 4
6. True

SECTION 2.9: RISC ARCHITECTURE IN THE AVR

1. RISC is reduced instruction set computer; CISC stands for complex instruction set computer.
2. True
3. Large
4. True
5. Around 130
6. True
7. (c)
8. False